**Title**
Dataflow analysis for concurrent programs using data-race detection



**Author**
Voung, Jan Wen

**Publication Date**
2010

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Dataflow Analysis for Concurrent Programs using Data-race Detection

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

in

Computer Science and Engineering

by

Jan Wen Voung

Committee in charge:

Professor Ranjit Jhala, Co-Chair
Professor Sorin Lerner, Co-Chair
Professor William G. Griswold
Professor Andrew B. Kahng
Professor Todd Millstein

2010

The dissertation of Jan Wen Voung is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____
Co-Chair

_____
Co-Chair

University of California, San Diego

2010

DEDICATION

For baba, mama, and gor gor (big brother).

# EPIGRAPH

*Greg Ostertag is one of the finest centers*
*in the history of Western Civilization.*
— Bill Walton

*Making something variable is easy.*
*Controlling duration of constancy is the trick.*
— Alan J. Perlis

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I must thank my two advisers, Ranjit and Sorin, for their many years of many things. First, I want to thank you for giving me guidance and support throughout projects that faced many difficulties along the way. I still remember some of the weeks where I would bring my laptop into your offices (almost) daily to show you the latest numbers and discuss where to go from there. Thank you also for the gastronomical support (cakes, cookies, pies, and potato products), even when deadlines were months away. The mix of energy, humor, and critical thought that you two show daily has been inspiring.

I also want to thank Ranjit and Sorin for the amazing job they have done in assembling the previously-non-existent programming systems / PL group in the few years that they have been at UCSD. These are some of the brightest and most energetic group of peers that I have had the pleasure of sharing (two!) offices with. Thank you all for the great discussions in and out of class. Thanks for sitting through those embarrassingly terrible first few practice talks, and inviting me to your own practice talks. Preparing talks was one of the areas I really needed to work on as a graduate student.

Individually, there are so many students I would like to thank with so many words, so please excuse me if the mention appears brief. Thank you Ravi for all the nights of burgers and burritos, and of course all the nights working on getting RADAR up and running for the first time. Ming, thanks for staying alive (that pancreas or whatever thing ;) ). Nathan, I hope you can keep your rasterbation habits under control. Pat, I will miss your razor sharp wit. Thank you Ross having the power to bring us Halloween in November, after the PLDI deadline. Thanks to Zach and Anne for the nickname "Crusher" that has given me confidence in these last years. Alden, Anshuman, Ganesh, Jack, Jeff, Joel, KV, Matus, Mike, Roy, and Steve, it has been a pleasure sharing the occasional dinner table with you all, among many other things. Thanks go out to the Ultimate players for keeping my heart healthy. Finally, thank you Daniel for being a great roommate and for all of your support, especially during year two. I hope to see you all again someday.

Thanks go out to my committee members Andrew, Bill, and Todd for the pointers and the great questions that have made this work more complete.

Outside of UCSD, I have to thank Franjo Ivancic, Vineet Kahlon, Sriram Sankaranarayanan, and Aarti Gupta at NEC Labs and Shuvendu Lahiri, and Shaz Qadeer at Microsoft for sharing insights related to my work as well as exposing me to broader issues and research in the fields of concurrency, software verification and reliability. It was a pleasure and joy working with you and learning from you.

Finally, I must thank my family for all of their love and support!

## Papers included in this dissertation

Chapter 2 and Chapter 5 in part, have been published as "Dataflow Analysis for Concurrent Programs using Datarace Detection" by Ravi Chugh, Jan Voung, Ranjit Jhala, and Sorin Lerner in *PLDI 08: Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation* [CVJL08]. The dissertation author was the primary investigator and author of this paper.

Chapter 3 in part, has been published as "Relay: Static Race Detection on Millions of Lines of Code" by Jan Voung, Ranjit Jhala, and Sorin Lerner in *ESEC/FSE 07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering* [VJL07]. The dissertation author was the primary investigator and author of this paper.

## VITA

| | |
|---|---|
| 2010 | Ph. D. in Computer Science and Engineering,<br>University of California, San Diego<br>Dis: *Dataflow Analysis for Concurrent Programs using Data-race Detection*<br>Advisers: Prof. Ranjit Jhala and Prof. Sorin Lerner |
| 2007 | M. S. in Computer Science and Engineering,<br>University of California, San Diego |
| 2004 | B. S. in Electrical Engineering and Computer Science,<br>University of California, Berkeley |

## PUBLICATIONS

S. Lahiri, S. Qadeer, J.P. Galeotti, **J.W. Young**, T. Wies, "Intra-module Inference", *Computer Aided Verification*, 493-508, 2009.

R. Chugh, **J.W. Young**, R. Jhala, S. Lerner, "Dataflow Analysis for Concurrent Programs Using Datarace Detection", *Programming Language Design and Implementation*, 316-326, 2008.

**J.W. Young**, R. Jhala, S. Lerner, "Static Race Detection on Millions of Lines of Code", *Foundations of Software Engineering*, 205-214, 2007.

D.C. Glaser, O. Feng, **J.W. Young**, L. Xiao, "Towards an Algebra for Lighting Simulation", *Building and Environment*, 895-903, 2004.

D.C. Glaser, **J.W. Young**, L. Xiao, B. Tai, S. Ubbelohde, J. Canny, E.Y. Do, "Lightsketch: A Sketch-modelling Program for Lighting Analysis", *CAAD Futures*. 371-382, 2003.

ABSTRACT OF THE DISSERTATION

**Dataflow Analysis for Concurrent Programs using Data-race Detection**

by

Jan Wen Voung

Doctor of Philosophy in Computer Science and Engineering

University of California, San Diego, 2010

Professor Ranjit Jhala, Co-Chair
Professor Sorin Lerner, Co-Chair

Dataflow analyses are a critical part of many optimizing compilers as well as bug-finding and program-understanding tools. However, many dataflow analyses are not designed for concurrent programs.

Dataflow analyses for concurrent programs differ from their single-threaded counterparts in that they must account for shared memory locations being overwritten by concurrent threads. Existing dataflow analysis techniques for concurrent programs typically fall at either end of a spectrum: at one end, the analysis conservatively kills facts about all data that might possibly be shared by multiple threads; at the other end, a precise thread-interleaving analysis determines which data may be shared, and thus which dataflow facts must be invalidated. The former approach can suffer from imprecision, whereas the latter does not scale.

This dissertation presents a framework called Radar, which automatically converts a dataflow analysis for sequential programs into one that is correct for

concurrent programs. With RADAR, the vast body of work in designing dataflow analyses for sequential programs can be reused in the concurrent setting.

RADAR uses a race detection engine to kill the dataflow facts – generated and propagated by the sequential analysis – that become invalid due to concurrent writes. This approach of factoring all reasoning about concurrency into a race detection engine yields two benefits. First, to obtain analyses for code using new concurrency constructs, there is no need to update each individual analysis; instead, one need only design a suitable race detection engine for the constructs. Second, it gives analysis designers an easy way to tune the scalability and precision of the overall analysis by only modifying the race detection engine or swapping in a different one.

This dissertation describes the RADAR framework and its implementation using a race detection engine called RELAY. RELAY is designed to be a push-button solution, and is designed to analyze independent portions of programs in parallel. Combined with RELAY, RADAR is capable of analyzing large C programs, including the Apache web server and a subset of the Linux kernel, in a reasonable time budget. We compare the results of RADAR to reasonable upper and lower bounds, and show that it is effective in generating concurrent versions of a null-pointer dereference analysis as well as several traditional compiler-oriented dataflow analyses.

# Chapter 1

# Introduction

Concurrency is a prevalent feature of almost all critical computing infrastructure, including operating systems, databases, internet-routing tables, and banking systems. Furthermore, several recent trends in technology, such as the widespread adoption of multi-core processors, web-service-oriented architectures, and peer-to-peer systems, are making concurrency more important than ever in mainstream programming.

Unfortunately, concurrent programming is hard. There are many ways for multiple threads of execution to interleave, and it is difficult for a programmer to mentally account for all these interleavings while coding. Furthermore, the execution paths that concurrent programs take are often affected by non-deterministic factors like the choices made by the scheduler. As a result, it is hard to systematically generate and reproduce dynamic executions. This not only makes testing especially ineffective in the face of concurrency, but it also makes concurrency problems hard to debug. Finally, concurrent programming models, such as those involving locks, require that all programmers follow a discipline throughout the entire program. Such disciplines are too easily violated as features are added or bugs are "fixed".

Meanwhile, advances in static algorithms for program optimization and error detection have shown that compiler technology can dramatically improve the performance and reliability of computer systems. Optimizations like array bounds checking elimination have allowed high-level languages to compete with

low-level languages while providing the increased security and benefits of memory safety ([Gup93, WWM07]). At the same time, error detection techniques complement testing and code reviews to find vulnerabilities involving null-pointer dereferences [DDA07], protocol violations [DLS02, BR02], SQL injection and cross site scripting [XA06], integer overflow [BCC+03], and array bounds violations [CH78, BCC+03, DRS03, WFBA00]. The success of this research can be seen in the many commercialized offerings and its application to improving safety-critical and foundational software.

Unfortunately, most of these algorithmic advances are limited to *sequential* programs and often ignore the challenges introduced by *concurrency*, where the need for static checking and potential for optimization are the greatest. Previous attempts at adapting these techniques to concurrent programs have been limited.

The simplest method of handling concurrent programs, which avoids expensive analysis, is to either make conservative or optimistic assumptions about concurrent threads. Optimizing compilers, which must preserve the semantics of the input program, take the conservative approach. As an example, compilers for memory safe languages like Java and C# do not attempt to eliminate array bounds checks for global arrays [Det], assuming that their lengths can be updated by concurrent threads. On the other hand, static code analyses, whose goal is to report bugs to programmers at compile time, may take the optimistic approach. Although this approach may miss concurrency-related bugs, it does not introduce the false alarms that come with conservative assumptions and overwhelm the programmer.

Alternatives to performing zero concurrency analysis include the following. First, the programmer can perform the "analysis" and provide annotations (such as `volatile`) that alert the compiler to the pieces of data that can be modified by concurrent threads. The compiler can then safely ignore these pieces of data – *i.e.,* not perform any optimizations that depend on them. Unfortunately, this solution is error-prone as the programmer can mistakenly forget annotations. Secondly, one can perform *escape analysis* to determine if a piece of data is modified by multiple threads [SR01]. However, the precision of these analyses is inherently limited – even if the escape information is perfect, there are some *critical pieces* of shared

data about which the analysis can infer nothing with respect to synchronization, making it impossible, for example, to statically prove the safety of dereferences of shared pointers or the access of shared arrays. Third, to overcome this imprecision, one can use custom concurrent analyses – tailored to specific models of concurrency and synchronization – to infer specific kinds of information [GS93, KSV96, RR99]. These analyses can be precise, but one must painstakingly retool a new analysis for each concurrent setting. Finally, one could use model checking to infer facts by exhaustively exploring all thread interleavings [CCO02, FQ03, HJM04]. While this is an extremely precise and generic approach, such analyses are unlikely to scale due to the combinatorial explosion in the number of interleavings.

## 1.1 Approach of this Dissertation

This dissertation presents a framework for converting program analyses that are designed for sequential programs to be sound in the concurrent setting. The conversion process is simple, requiring little change to the original sequentially-minded analysis. The framework contains a component for reasoning about concurrency, and this component is a tunable parameter. Thus, users can decide on the right balance between the opposing forces of precision and scalability. Through experiments we show that this framework can scale to large code bases.

The analyses considered for conversion to the concurrent setting are those that can be formulated as a dataflow analysis. Many analyses used in compiler optimizations [ASU86] and static code analysis can be formulated as a dataflow analysis.

## 1.2 Factoring out the Concurrency Analysis

The framework is based on two insights. First, the most common way for a programmer to ensure a fact about a piece of shared data at any given point in a thread is to ensure that no other thread can *modify* the data while the first thread is still at that point. Our second insight consists of a way of using *race detection* to

determine when dataflow facts may be invalidated or *killed* by the actions of other threads. A *data-race* occurs when multiple threads are about to access the same piece of memory and at least one of those accesses is a write. Since data-races are a common source of tricky bugs, several static analyses have been developed to find races, or show their absence. Our insight is that to determine whether the actions of other threads can kill a fact inferred about some data at some point, it suffices to determine whether an *imaginary read* of the data at the point can race with a write to that data by another thread.

We combine these insights in a framework called RADAR that takes as input a *sequential* dataflow analysis and a *race detection* engine, and returns as output a version of the sequential analysis that is sound for multiple threads (Figure 1.1). RADAR combines our insights as follows. It first runs the sequential analysis. At each program point, after the transfer function for the sequential dataflow analysis has propagated facts to the point, RADAR queries the race detector to determine which facts must be killed due to concurrency. More precisely, for each propagated fact, RADAR asks the detector if an imaginary read, *at that program point*, of the memory locations that the fact depends on can race with writes performed by other threads. If the answer is yes (that is, if another thread may be concurrently writing to one of the locations), then the dataflow fact is killed. If the answer is no (that is, if no other threads can possibly be writing to these locations), then the dataflow fact remains valid in the concurrent setting.

RADAR's approach of factoring all reasoning about concurrency into a race detection engine is less precise than a custom analysis where the concurrency analysis may gain information from the dataflow analysis and vice versa. However, the looser coupling of RADAR yields two concrete benefits. First, to obtain analyses for code using new concurrency constructs, one need only design a suitable race detection engine for the constructs. Second, it gives analysis designers an easy way to tune the scalability and precision of the overall analysis by only modifying the race detection engine.

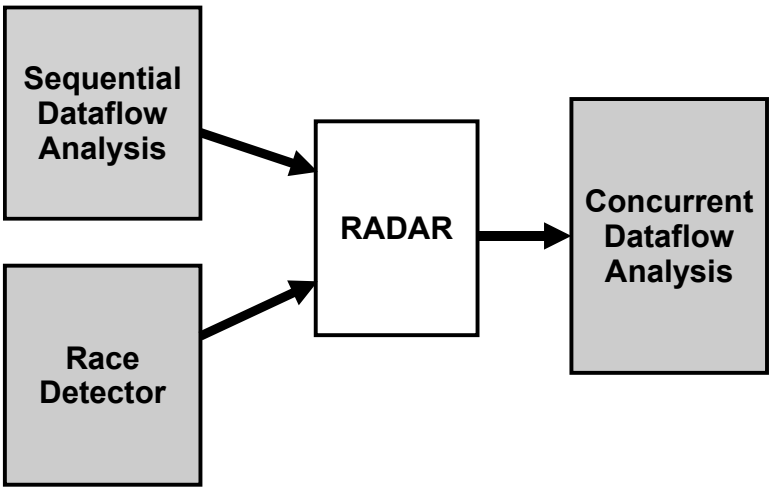



Figure 1.1: Radar inputs / outputs

## 1.3 Scalable Data Race Detection

An important ingredient of Radar is a race detection engine. However, we are not interested in just any race detector. We require that it be (1) *static*, in that it runs before the program is executed, (2) *sound*, in that it should guarantee that it finds all races, and (3) *scalable*, in that it should be effective on programs comprising millions of lines of code.

The above three goals have previously never been achieved all at once. In particular, while sound and static race detection techniques have proven to be effective, the largest programs they have ever been applied to are on the order of tens of thousands of lines of C code [PFH06, Ter08] and little over a hundred thousand lines of Java code [NAW06]. Furthermore, while some static race detection algorithms run on millions of lines of code [EA03], they are extremely unsound, and therefore would cause Radar to produce an unsound client analysis.

We take a step towards achieving all three goals by developing Relay, a static and scalable algorithm that can perform race detection on programs as large and complicated as the Linux kernel (which comprises 4.5 million lines of C code). In Relay unsoundness is *modularized* to the following sources: (1) Relay ignores reads and writes that occur inside blocks of assembly code; (2) Relay does not

handle corner cases of pointer arithmetic correctly; and in some cases (3) Relay uses a per-file alias analysis to optimistically resolve function pointers. Sources (1) and (2) are shared by many other research-level race detectors ([PFH06, Ter08]). Unsoundness from function pointers can be made eliminated – Relay has hooks to sound function pointer analyses (call-graph construction algorithms).

Besides serving as a component within the Radar framework, the evaluation of Relay has also lead to a better understanding of the kinds of program patterns that can make a race detector imprecise. More importantly, we have designed several filters to estimate the prevalence of each pattern. Each filter is targeted to a specific pattern, and so the number of warnings removed by each filter is an estimate. This evaluation of Relay allows designers of future data-race detectors to focus their efforts on aspects with the greatest impact.

## 1.4 Evaluation of Call-graph Construction

Relay and many other inter-procedural analyses are structured such that mutually recursive functions – strongly connected components (SCCs) in a program's call-graph– are analyzed together. A problem is that imprecise function pointer analysis can create artificially large SCCs in a program's call-graph. This can force Relay and Radar to analyze tens of thousands of functions at once, in large million-line programs like the Linux kernel.

This scalability wall due to function pointers led to further investigation of call-graph construction algorithms, in an attempt to answer the following three questions. First, what is the best precision that can be achieved from known call-graph construction algorithms? Although call-graphs constructed using Steensgaard are imprecise, we do not know how much more precise of a call-graph can be obtained while still being capable of processing large code bases. Second, if one call-graph construction algorithm is in fact more precise than another, what aspect of the analysis gives the greatest improvement? Finally, the third question is how does the precision of a call-graph affect the precision of clients such 1Relay?

This dissertation contains an experimental evaluation of a wide spectrum

of call-graph construction algorithms on a number of large C benchmarks with the goal of answering the above questions. In particular, we are interested in measuring how different analysis choices, such as unification vs. inclusion, flow-insensitive vs. sensitive, field-insensitive vs. sensitive, context-insensitive vs. sensitive, affect the quality of the computed call-graph, the quality of the results produced by a client analysis, and the scalability of the client analysis. In the process, we also devised a method of restricting a context-sensitive, flow-sensitive, and field-sensitive pointer analysis [WL95] to function values, in the context of C programs, so that it is applicable to larger code bases.

## 1.5 Contributions and Outline

To sum up, the main contributions of this dissertation are as follows.

- We have designed a framework called RADAR that automatically converts a sequential dataflow analysis into a concurrent one using a race detection engine (Chapter 2).
- We have developed a race detection engine called RELAY that scales to millions of lines of C code by exploiting modularity and parallelism (Chapter 3).
- Using RELAY we were able to find code patterns on which race detectors may find difficult to analyze precisely. We then developed filters to remove false warnings related to each pattern as well as estimate the prevalence of each pattern (Chapter 3).
- We have compared a large range of call-graph construction algorithms (based on different pointer analyses). This is done to better understand which of the trade-offs made by the different pointer analyses are most important to the precision of the generated call-graph (Chapter 4). This study is done on larger code-bases (where function pointers may be used more heavily) and with more advanced pointer analyses than previous studies.

- We have measured how the precision of a generated call-graph affects the precision and scalability of inter-procedural dataflow analyses including RELAY and clients of RADAR (Chapter 4).

- We have instantiated the RADAR framework with RELAY, the result we call RADAR(RELAY). We then use RADAR(RELAY) to transform several sequential dataflow analyses into concurrent dataflow analyses. The analyses converted by RADAR(RELAY) achieve good precision relative to some appropriate upper and lower bounds. Furthermore, we find that RADAR(RELAY) easily scales to hundreds of thousands of lines of code (Chapter 5).

Finally, Chapter 6 covers related work for RADAR and RELAY while Chapter 7 concludes the paper and discusses possible future work.

**Appendices.** Details on the monotonicity and distributivity of dataflow analyses used in this dissertation are presented in Appendix A. An approach to restricting the Wilson and Lam pointer analysis [WL95] to function pointer values (for the purposes of call-graph construction) is discussed in Appendix B.

# Chapter 2

# Radar: Dataflow Analysis for Concurrent Programs

This chapter describes RADAR, a framework for converting dataflow analyses from the sequential setting to the concurrent setting. We begin with an overview built around an example to give some intuition behind RADAR and how it works. We also use this example to compare RADAR to other approaches, at a high level. Section 2.2 describes the framework more formally and in more detail. The framework is parameterized by several components, but how the framework can be instantiated is left to a later chapter (5).

## 2.1 Overview of Radar

We begin with an overview of our technique using some simple examples. First, consider the multithreaded program shown in Figure 2.1, which executes a single copy of the `Producer` thread and a single copy of the `Consumer` thread. There is a shared, acyclic list of structures named `bufs`, and a shared performance counter `perf_ctr`. To enable mutually exclusive list access, there is a lock `buf_lock` which is initially "unlocked", *i.e.,* not held by any thread. The `Producer` (respectively `Consumer`) thread has a local reference `px` (respectively `cx`) used to iterate over the list `bufs`.

The `Producer` thread iterates over the cells in the list `bufs`. In each iter-

| *Adjusted Analysis* | |
|---|---|
| | `buffer_list *bufs;` |
| | `lock buf_lock;` |
| | `int perf_ctr;` |
| | `thread producer1(){` |
| ∅ | `P0: px = bufs;` |
| ∅ | `P1: while (px != NULL){` |
| *px* | `P2:  lock(buf_lock);` |
| *px* | `P3:  px->data = new();` |
| | |
| | |
| *px->data,px* | `P5:  perf_ctr++;` |
| *px->data,px* | `P6:  t=produce();` |
| | |
| | |
| *px->data,px* | `P8:  *px->data = t;` |
| *px->data,px* | `P9:  unlock(buf_lock);` |
| ~~px->data~~,px | `PA:  px = px->next;` |
| | `}` |
| | `thread consumer1(){` |
| ∅ | `perf_ctr = 0;` |
| ∅ | `C0: cx = bufs;` |
| ∅ | `C1: while(cx != NULL){` |
| *cx* | `C2:  lock(buf_lock);` |
| *cx* | `C3:  if(cx->data != NULL){` |
| *cx->data,cx* | `C4:  consume(*cx->data);` |
| *cx->data,cx* | `C5:  cx->data = NULL;` |
| *cx* | `C6:  cx = cx->next;` |
| | `}` |
| ∅ | `C7:  unlock(buf_lock);` |
| | `}` |

Figure 2.1: `Producer-Consumer` example.

ation, it acquires the lock `buf_lock` protecting the cell, and resets the `px->data` to a new buffer initialized with the value `0` that will hold the data that will be produced. Next, it obtains the value, via a call to `produce` and once it is ready, the producer writes the value into `*px->data` and moves onto the next cell.

The `Consumer` thread iterates over the cells in the list `bufs`. In each iteration, it acquires the lock `buf_lock` and if the pointer `cx->data` is non-null, it consumes the data, resets the pointer to free the buffer, and moves to the next cell in the list. Finally, the consumer releases the lock.

We assume that the shared list contains no cycles and that it starts off with all the data fields set to `null`. Thus, the net effect of having the `Producer` and `Consumer` running in parallel is that the producer walks through the list setting the data field of each individual cell, and the consumer trails behind the producer, using the data field in each cell and resetting it to `null`. Finally, notice the `Consumer` thread initializes `perf_ctr` without holding any locks, and so the initialization races with the increment operation at `P5` in the `Producer` thread. However, we shall assume that the programmer has deemed that the race is benign as it is on an irrelevant performance counter.

### 2.1.1 Sequential Non-Null Analysis

Suppose we wish to statically determine whether any null-pointer dereference occurs during the execution of the program in Figure 2.1. To this end, we could perform a standard sequential dataflow analysis, using flow facts of the form $NonNull(l)$ stating that the lvalue $l$ is non-null, and flow functions that appropriately generate, propagate, and kill such facts using guards and assignments.

Let us assume that `new`() returns a non-null pointer to a cell initialized with `0`. The set of facts in the fixpoint solution computed by this analysis is shown on the left in Figure 2.1 (for the moment, ignore the line crossing out a fact on the last line). At points `P0` and `P1`, every lvalue may be null. At `P2` and `P3` `px` is non-null, and everywhere else, the sequential analysis determines that the lvalues `px` and `px->data` are non-null. Thus, the analysis determines that at each program point where a pointer is dereferenced, the pointer is non-null, and so all of the dereferences are safe.

**Sequential analysis is unsound, even without data-races.** In this case, the above conclusion is sound: there are indeed no null-pointer dereferences in the program. In general, however, a sequential non-null dataflow analysis may actually miss some null-pointer dereferences. As an example, consider a scenario where the programmer who wrote Figure 2.2 mistakenly uses the intuition that the system is safe as long as there are no *data-races* on any of the shared cells. Thus, to improve the performance of the program from Figure 2.1, the programmer writes the modified version of the `Producer` thread shown in Figure 2.2, where `buf_lock` is temporarily released while the data is being produced to allow the `Consumer` thread to concurrently make progress. The resulting system has no races, and so the programmer may think that the system is safe.

However, this intuition turns out to be incorrect, and in fact the programmer has actually introduced a null-pointer bug, even though there are no races. This is because *after* the producer thread has initialized `px->data` and released the lock, the consumer can acquire the lock and reset the pointer. When the producer thread re-acquires the lock after storing the data temporarily in `t`, the dereference at `P8` can cause a crash because the pointer is null.

| *Adjusted Analysis* | |
|---|---|
| | `buffer_list *bufs;` |
| | `lock buf_lock;` |
| | `int perf_ctr;` |
| | `thread producer1(){` |
| ∅ | `P0: px = bufs;` |
| ∅ | `P1: while (px != NULL){` |
| px | `P2:  lock(buf_lock);` |
| px | `P3:  px->data = new();` |
| px->data,px | `P4:  unlock(buf_lock);` |
| | |
| ~~px->data~~,px | `P5:  perf_ctr++;` |
| ~~px->data~~,px | `P6:  t=produce();` |
| | |
| ~~px->data~~,px | `P7:  lock(buf_lock);` |
| ~~px->data~~,px | `P8:  *px->data = t;` |
| ~~px->data~~,px | `P9:  unlock(buf_lock);` |
| ~~px->data~~,px | `PA:  px = px->next;` |
| | `}` |
| | `thread consumer1(){` |
| ∅ | `perf_ctr = 0;` |
| ∅ | `C0: cx = bufs;` |
| ∅ | `C1: while(cx != NULL){` |
| cx | `C2:  lock(buf_lock);` |
| cx | `C3:  if(cx->data != NULL){` |
| cx->data,cx | `C4:  consume(*cx->data);` |
| cx->data,cx | `C5:  cx->data = NULL;` |
| cx | `C6:  cx = cx->next;` |
| | `}` |
| ∅ | `C7:  unlock(buf_lock);` |
| | `}` |

Figure 2.2: Buggy version of `Producer-Consumer`.

Unfortunately, even though the new program has a null-pointer bug, the sequential analysis returns exactly the same solution as for the original program (shown on the left in Figure 2.2). That is, at each point the same lvalues are deemed to be non-null, and thus the sequential analysis would not discover the null-pointer bug.

### 2.1.2 The Problem: Adjusting for Multiple Threads

To address the above problem, we want to *automatically* convert a sequential dataflow analysis into one that is sound in the presence of multiple threads. Doing this in a way that is also precise is not trivial.

Consider for example the simple solution of running an escape analysis, and keeping only *NonNull* facts for those lvalues that do not escape the current thread. The `px->data` field escapes the current thread, so the analysis would never infer *NonNull*(`px->data`). Although this solution makes the analysis sound in the face of concurrency (and in particular, it would find the bug in Figure 2.2), it also

makes the analysis imprecise: the resulting concurrent analysis would not even be able to show that the original program from Figure 2.1 is free of null-pointer bugs.

Alternatively, one may be tempted to run independent sequential analyses over blocks that are *atomic* in the sense of [Lip75, FQ03] and conservatively *kill* facts over shared variables at atomic block exit points [CR06]. Intuitively, a block is atomic if for each execution of the operations of the block where the operations are interleaved with those of other threads, there exists a semantically equivalent execution where the operations of the block are run without interleaving. Unfortunately, the body of the `Producer` loop from Figure 2.1 is not atomic because of the benign race on the performance counter `perf_ctr`. This race splits the body of `Producer` into multiple atomic blocks – the statements before, at, and after the racy increment. Thus, such an analysis would kill the *NonNull*(`px->data`) fact at `P5`, and would be too imprecise to prove that the program from Figure 2.1 is free of null-pointer bugs.

### 2.1.3 Our Solution: Pseudo-Race Detection

We now describe our solution to this problem, which allows us to leverage existing race detection engines to build a sound concurrent analysis that is strictly more precise than the previously mentioned simple approaches. As discussed in Section 6, race detection is a well-studied problem. For programs using lock-based synchronization, there are scalable race detectors that infer the sets of locks that protect each shared memory location and produce a race warning if two threads access a shared cell without holding a common lock.

**Adjust.** Our first insight is that the facts that can soundly be inferred to hold in the presence of multiple threads are the *subset* of facts established via sequential analysis which are not killed by operations of other threads. Thus, the multi-threaded dataflow analysis can be reduced to determining which facts inferred at a particular point by the sequential analysis are killed by other threads. To determine if a fact can be killed by a concurrently executing operation of another thread, it suffices to check if another thread can *concurrently write* any lvalue appearing in the flow fact.

**Pseudo-Races.** Our second insight is that to perform this check we can insert *pseudo-reads* corresponding to the lvalues in the flow fact at the program point, and query a *race detection* engine to determine if any of the pseudo-reads can race with a write to the same memory location. If such a *pseudo-race* occurs, then the fact is killed; otherwise, the analysis deduces that the fact continues to hold at the point even in the presence of other threads.

We have designed a framework called RADAR that combines these two insights to convert an arbitrary dataflow analysis for sequential programs into one that is sound for multithreaded programs. During analysis, RADAR uses the sequential flow function, but at each program point, it kills the facts over lvalues that have pseudo-races at that point. This mechanism captures the following informal idiomatic manner in which the programmer reasons about multiple threads. Each thread performs some operation that establishes a certain fact in the programmer's head, *e.g.* a null check or initialization or an array bounds check. The programmer can only expect that the fact continues to hold as long as other threads cannot modify the memory locations. As a result, the programmer uses synchronization mechanisms to "protect" the memory locations from writes by other threads as long as the information is needed. Our technique of adjusting preserves only those facts that the race detection engine deems to be protected from modification.

### 2.1.4 Multithreaded Non-Null Analysis

Let us consider the result of running the non-null analysis adjusted using RADAR to account for multiple threads. The lines in Figures 2.1 and 2.2 show the facts generated by the sequential analysis that get killed during the adjusting because of pseudo-races.

For both the correct and the buggy programs, in the `Consumer` thread the adjusting has no effect because `cx` is thread-local, and due to the held lock `buf_lock`, there are no races on the pseudo-reads of `cx->data` at program points `C4` and `C5`.

**Safety without Atomicity.** In the correct `Producer` thread of Figure 2.1, the adjusting process has no effect on facts over (only) the thread-local, and hence,

race-free lvalue `px`. The initialization at `P3` causes the fact *NonNull*(`px->data`) to get generated at program point `P5`. The adjusting does not kill this fact because the lock `buf_lock` held at `P5` ensures there is no pseudo-race on `px->data`. Similarly, the fact *NonNull*(`px->data`) generated at `P3` is not killed by the adjusting at `P6` − `P9`, as the held lock `buf_lock` ensures there are no races with the pseudo-read on `px->data` at any of these points. As the lock is released at `P9`, the fact *NonNull*(`px->data`) is killed by the adjusting at `PA`, as the pseudo-read can race with the write in the `Consumer` thread. The adjusted analysis shows that the dereferences in the program are safe, as `px->data` is soundly inferred to be non-null at `P8`, where the dereference takes place. Notice the adjusted analysis can soundly show that the program does not cause any null-pointer dereferences, even though the producer thread, even the loop-body, is not atomic, due to `perf_ctr` which may be accessed without any synchronization. By preserving the facts that are over protected lvalues, our adjusting technique can ignore atomicity "breaks" caused by *benign races* on irrelevant entities like `perf_ctr`. Thus, our RADAR adjusting technique is strictly more precise than running independent sequential analyses over semantically atomic blocks.

**Concurrency Errors without Races.** In the buggy `Producer` thread of Figure 2.2, the facts over the thread-local lvalues are not killed by adjusting. However, notice that although the sequential analysis propagates fact *NonNull*(`px->data`) to `P5`, the adjusted version kills the fact since once the protecting lock is released at `P4`, the pseudo-read of `px->data` at `P5` can race with the write at `C5` in a `Consumer` thread. As this fact is killed at `P5`, it does not propagate in the adjusted analysis to `P6` − `P9`, as happens in the sequential analysis. Thus, as a result of the adjusting, the dereference at `P8` is no longer inferred to be safe as `px->data` may be null at this point! Thus, our technique finds an error caused by multithreading that is absent from the sequential version of the program, *even though there are no data-races* in the program except on the irrelevant `perf_ctr`.

**Beyond lock-based synchronization.** Although we have used lock-based synchronization to show how RADAR works, our adjusting technique is applicable to any synchronization regime for which race detection techniques exist, not just those

| *Adjusted Analysis* | |
|---|---|
| | `buffer_list *bufs;` |
| | `int flag;` |
| | `int perf_ctr;` |
| | `thread producer2(){` |
| $\phi$ | `P0: px = bufs;` |
| $\phi$ | `P1: while (px != NULL){` |
| *px* | `P2:  while(px->flag !=0){};` |
| *px* | `P3:  px->data = new();` |
| | |
| | |
| *px->data,px* | `P5:  perf_ctr++;` |
| *px->data,px* | `P6:  t=produce();` |
| | |
| | |
| *px->data,px* | `P8:  *px->data = t;` |
| *px->data,px* | `P9:  px->flag = 1;` |
| ~~*px->data*~~*,px* | `PA:  px = px->next;` |
| | `}` |
| | `thread consumer2(){` |
| $\phi$ | `perf_ctr = 0;` |
| $\phi$ | `C0: cx = bufs;` |
| $\phi$ | `C1: while(cx != NULL){` |
| *cx* | `C2:  while(cx->flag==0){};` |
| *cx* | `C3:  if(cx->data != NULL){` |
| *cx->data,cx* | `C4:  consume(*cx->data);` |
| *cx->data,cx* | `C5:  cx->data = NULL;` |
| *cx* | `C6:  cx = cx->next;` |
| | `}` |
| $\phi$ | `C7:  cx->flag = 0;` |
| | `}` |

Figure 2.3: Flag-based version

based on locks. Consider a version of the `Producer-Consumer` example, shown in Figure 2.3, which has finer-grained synchronization done with a flag field in each of the structures in the cyclic list. Now, instead of acquiring the lock, the `Producer` thread spins in a loop while `px->flag` is non-zero, which indicates that the data in the structure has not yet been consumed. Once the flag is zero, the producer, initializes the `px->data` field, writes the new data into it, and sets the `px->flag` field to `1` indicating the data is ready. Dually, the `Consumer` thread spins while the `cx->flag` field is zero, at which point it consumes the data and resets the `cx->data` field. The result of the adjusted analysis for this program is identical to the result for the fixed program of Figure 2.2, as a more general race detection engine (*e.g.* one based on model checking [HJM04]) would deduce that there were no pseudo-races on `px->data` in the locations `P5` – `P9`. Once the flag is set at `P9`, the pseudo-read of `px->data` at `PA` can race with the write at `C5` in a `Consumer` thread, and so the adjust kills the fact *NonNull*(`px->data`) at `PA`.

In the rest of the chapter, we formalize the RADAR framework and show how it converts sequential analyses into concurrent ones.

## 2.2 The Radar Framework

This section presents the RADAR framework for concurrent dataflow analysis in several steps. We start by presenting (in Section 2.2.1) a basic version of RADAR. Although this basic version lacks certain important features and optimizations (such as support for function calls), it illustrates the foundation of our approach. We then gradually refine the basic framework (in Sections 2.2.3 and 2.2.4) by adding various optimizations and features. Section 2.2.5 discusses known limitations of the approach. Section 2.2.6 describes in further detail how RADAR handles may analyses and backwards analyses. Finally, section 2.2.7 contains proofs that RADAR enjoys a few properties (*e.g.* it preserves monotonicity of flow functions). Experimental evaluation of RADAR is delayed to chapter 5, after components of the framework are described in further detail.

**Assumptions.** We make two standard assumptions about the program being analyzed and the system it is to be run on. First, we assume that for each procedure, either we have its code or we have a summary that soundly approximates its behaviors. As a result, our framework can analyze incomplete programs (*e.g.* programs that use libraries), since we can model the missing procedures using summaries. Second, we assume that the shared memory system is sequentially consistent [Lam77] in that memory operations are executed in the order in which they appear in the program.

### 2.2.1 Intra-procedural Framework

We start by presenting a basic framework for generating an intra-procedural concurrent dataflow analysis from an intra-procedural sequential dataflow analysis.

**Sequential Dataflow Analysis.** We assume a Control Flow Graph (CFG) representation of programs, where each node represents a statement, and edges between nodes are program points where dataflow information is computed. We use *Node* to represent the set of all CFG nodes and *PPoint* the set of all program points. In the program shown in Figure 2.1, the program point `P3` is the point just before executing the statement at `P3`.

We assume that the sequential dataflow analysis computes a set of dataflow facts at each program point, and *DataflowFact* represents the set of all possible dataflow facts. We assume that dataflow facts in *DataflowFact* are must facts, which means that may information, if needed, has to be encoded by the absence of must information. Thus, the domain of the sequential dataflow analysis is $D = 2^{DataflowFact}$, ordered as a lattice $(D, \sqsubseteq, \top, \bot, \sqcap, \sqcup)$, where $\sqsubseteq$ is $\supseteq$, $\top$ is $\emptyset$, $\bot$ is *DataflowFact*, $\sqcap$ is $\cup$, and $\sqcup$ is $\cap$.

We also assume that the flow function is given as:

$$F : Node \times D \times PPoint \rightarrow D$$

Given a node $n$, some incoming dataflow information $d$, and an outgoing program point $p$ for node $n$, $F(n, d, p)$ returns the outgoing dataflow information. We assume that if a node $n$ has more than one incoming program point, the dataflow information is merged using $\sqcup$ before being passed to the flow function.

Examples of dataflow facts include: $ConstValue(\mathtt{x}, \mathtt{5})$, which states that `x` has the value of `5`, $MustPointTo(\mathtt{x}, \mathtt{y})$, which states that `x` points to `y`, and $NonNull(\mathtt{p})$, which states that `p` is safe to dereference.

**Requirement on Dataflow Information.** As our framework does not depend on the exact details of the *DataflowFact* set, analysis writers have the freedom to choose the way in which they encode dataflow information. However, we do place a requirement on the *DataflowFact* set: we assume the existence of a function *Lvals* that returns the set of lvalues that a fact depends on. Intuitively, given a dataflow fact $f \in DataflowFact$, $Lvals(f)$ returns the set of lvalues that, if written to with arbitrary values, would invalidate the fact $f$. We denote the set of all lvalues by *Lvals*, and so the function *Lvals* has type $DataflowFact \rightarrow 2^{Lvals}$. As an example, for the dataflow facts mentioned above, we would have:

$$Lvals(ConstValue(\mathtt{x}, \mathtt{5})) = \{\mathtt{x}\}$$
$$Lvals(MustPointTo(\mathtt{x}, \mathtt{y})) = \{\mathtt{x}\}$$
$$Lvals(NonNull(\mathtt{p})) = \{\mathtt{p}\}$$

Although we assume that the *Lvals* function is given, it can easily be computed from the sequential flow function $F$ if $F$ handles "havoc" CFG nodes of the form

"$l := \bot$". In particular:

$$Lvals(f) = \{l \mid l \in Lvals \wedge f \notin F(\text{“}l := \bot\text{”}, \{f\}, \_)\}$$

**Concurrent Dataflow Analysis.** We capture the way in which concurrency affects the sequential dataflow analysis through a function $ThreadKill : PPoint \times Lvals \rightarrow Bool$. Given a program point $p$ and an lvalue $l$, $ThreadKill(p, l)$ returns whether or not $l$ may be written to by concurrent threads when the current thread is at program point $p$. The *ThreadKill* function, which we will define later in terms of a race detection engine, is at the core of our technique: it allows RADAR to kill dataflow facts that are made invalid by concurrent writes.

Given the sequential flow function $F$ and the *ThreadKill* function, we define $F_{Adj}$, the flow function for the concurrent analysis:

$$\begin{aligned} F_{Adj}(n, d, p) = \{f \mid \quad & f \in F(n, d, p) \wedge \\ & \forall l \in Lvals(f) \ . \ \neg ThreadKill(p, l)\} \end{aligned}$$

This adjusted flow function essentially propagates dataflow facts that are produced by the original flow function $F$ and that are not killed by any concurrent writes.

**Adjusting via Race Detection.** The key contribution of our work lies in the way in which we use a race detection engine to compute *ThreadKill*. As a result, we need a way to abstract the race detection engine. We achieve this through a function $RacyRead : PPoint \times Lvals \rightarrow Bool$, which behaves as follows: given a program point $p$ and an lvalue $l$, $RacyRead(p, l)$ returns *true* if a read of $l$ at program point $p$ would cause a race.

**Soundness.** For our framework to be sound, the race detection engine must be sound, in the sense that if there really is a race, then *RacyRead* must return *true* (but *RacyRead* can also return *true* in cases where there is no race). To formalize this soundness property, we assume a perfect race detection oracle $RealRace : PPoint \times Lvals \rightarrow Bool$, such that $RealRace(p, l)$ returns *true* exactly when there is an execution in which a read of $l$ at $p$ would cause a race. The following requirement states that the race detection engine *RacyRead* must approximate

the oracle *RealRace*:

$$\begin{aligned} &\forall p \in PPoint, l \in Lvals \ . \\ &\quad RealRace(p, l) \Rightarrow RacyRead(p, l) \end{aligned} \tag{2.1}$$

Having a sound race detection engine, the *ThreadKill* function can then be defined as:

$$ThreadKill(p, l) = RacyRead(p, l)$$

This basic definition of *ThreadKill* expresses the key insight behind the RADAR framework, which is that pseudo-races can be used as a way of determining when concurrent writes could happen.

**Instantiation Requirements:** To instantiate the basic RADAR framework, one needs to provide a race detection engine $RacyRead : PPoint \times Lvals \rightarrow Bool$ that satisfies the soundness property (2.1).

### 2.2.2 Impact of Unsoundness from the Race Detector

In practice there may be unsoundness in the data-race detector due to errors in implementation, designs based on assumptions over the target language that hold for most programs but not all, or trade-offs made between soundness and run-time performance. We would like to understand how RADAR transfers unsoundness in the data-race detector to the generated concurrent dataflow analysis, if at all. A useful measure of unsoundness in the dataflow would be based on the client of the dataflow analysis. For example, instead of counting the existence of $NonNull(l)$ fact that is untrue, we only count that existence as an unsoundness if it results in a mis-classification of a pointer dereference as safe versus unsafe.

How RADAR transforms unsoundness in the data-race detector to the converted concurrent dataflow analysis depends on the sequential dataflow analysis and the program being analyzed. One cannot say that RADAR will uniformly "amplify" or "dampen" unsoundness.

A data-race detector may be unsound in many ways. Instead of enumerating all possible ways that it can be unsound, we can measure unsoundness by

comparing the result of *RacyRead* against the oracle *RealRace*. To see how translation of unsoundness depends the sequential dataflow analysis we must consider whether or not a pseudo-read is inserted at all. A race detector is unsound on a particular program location $p$ and lvalue $l$ if *whenever* a pseudo-read is queried $RealRace(p, l)$ and result is true, $RacyRead(p, l)$ responds with false. Given this unsoundness we can perform the following case analysis:

- **Case No Pseudo-Read:** This is the case where the dataflow analysis does not insert a pseudo-read for an lvalue $l$ at location $p$. An example of this is in Figure 2.2 where `perf_ctr` is an integer variable and therefore irrelevant to a Non-Null client dataflow analysis. In this case, RADAR will dampen the unsoundness.

- **Case Pseudo-Read:** This is the case where the dataflow analysis *will* insert a pseudo-read for lvalue $l$ at location $p$. The result depends on the program. Consider the following sub-cases.

- **Sub-Case Irrelevant:** In this sub-case, the program is structured such that the client of the dataflow analysis *does not* rely on the soundness of the dataflow-fact. A missed race is irrelevant and again RADAR will dampen the unsoundness. An example of this is shown on the left of Figure 2.4. In the example, it is possible that there is a race at location `L`, but this is irrelevant since the program will check again if the pointer is  or not, at location `L2`.

- **Sub-Case Relevant:** In this sub-case, the program is structured such that the client of the dataflow analysis *does* rely on the dataflow-fact. A missed race is relevant and, depending on the data-dependencies present in the program, RADAR may amplify the unsoundness. The example on the right of Figure 2.4 shows that a single missed data race will result in $n$ mis-categorized dereferences (in the example, each `foo_i` dereference `t`).

```
if (p != null) {
L:  lock(l);
L2: if (p != null) {
        use (*p);
    }
    unlock(l);
}
```

```
if (p != null) {
L:  t = p;
    foo_1(t);
    foo_2(t);
    foo_3(t);
    ...
    foo_n(t);
}
```

Figure 2.4: RADAR can dampen or amplify unsoundness in the race detector

### 2.2.3 Optimization: Race Equivalence Regions

The basic RADAR framework from Section 2.2.1 performs a race check at each program point for each lvalue that the dataflow facts depend on. This can lead to a large number of race checks, in the worst case $n \times m$, where $n$ is the number of program points and $m$ is the number of lvalues. To reduce this large number of race checks, we partition program points into race equivalence regions.

Intuitively, a race equivalence region is a set of program points that have the same raciness behavior: for each lvalue, either the lvalue is racy throughout the entire region, or it is not racy throughout the entire region. It is not possible for an lvalue to be racy in one part of the region and not racy in another part. Race equivalence regions reduce the number of race checks because by checking the raciness of an lvalue at (any) one program point in a region, RADAR can know the raciness of the lvalue throughout the entire region.

**Race Equivalence Regions.** Formally, we define a partitioning of program points into race equivalence regions as a pair $(R, \mathit{Reg})$, where $R$ is a set of regions, and $\mathit{Reg} : \mathit{PPoint} \rightarrow R$ is a function mapping each program point to a region. Two program points $p$ and $p'$ are *race equivalent* if $\mathit{Reg}(p) = \mathit{Reg}(p')$. We say a program point $p$ *belongs to* a region $r$ if $\mathit{Reg}(p) = r$. Since all program points belonging to a region are equivalent in terms of race detection, we change the interface to the race detection engine to take a race equivalence region rather than a program point:

$$\mathit{RacyRead} : R \times \mathit{Lvals} \rightarrow \mathit{Bool}$$

One possible implementation for this new *RacyRead* is to choose a unique representative program point for each region, and when queried with a particular region $r$ and lvalue $l$, to return the result of the old *RacyRead* on $r$'s representative point and $l$.

**Soundness.** Instead of imposing a particular way of implementing *RacyRead*, we define a soundness requirement for the new *RacyRead* engine and the *Reg* function:

$$\begin{aligned} &\forall p \in PPoint, l \in Lvals \ . \\ &\qquad RealRace(p, l) \Rightarrow RacyRead(Reg(p), l) \end{aligned} \tag{2.2}$$

With this new abstraction for the race detection engine, the *ThreadKill* function becomes:

$$ThreadKill(p, l) = RacyRead(Reg(p), l)$$

This new definition of *ThreadKill* reduces the number of race checks for each lvalue from at most once per program point to at most once per region.

**Instantiation Requirements:** To instantiate the region-based RADAR framework, one needs to provide:

1. A **race detection engine** $RacyRead : R \times Lvals \rightarrow Bool$
2. A **region map** $Reg : PPoint \rightarrow R$

such that *RacyRead* and *Reg* satisfy property (2.2). We now give two examples to illustrate the idea of race equivalence regions.

**Example: Global Locksets.** One possible instantiation of regions uses locksets. If we assume that there is a global set of locks *Locks*, we can define $R = 2^{Locks}$, which means that a race equivalence region is simply a set of locks, and the program points in the region are those program points at which the region's locks are held.

Consider the buggy example from Figure 2.2. The *Reg* map is:

$$Reg(p) = \begin{cases} \{\texttt{buf\_lock}\} & \text{if } p \in \{\texttt{P3}, \texttt{P4}, \texttt{P8}, \texttt{P9}\} \\ \{\texttt{buf\_lock}\} & \text{if } p \in \{\texttt{C3}, \texttt{C4}, \texttt{C5}, \texttt{C6}, \texttt{C7}\} \\ \emptyset & \text{otherwise} \end{cases}$$

which captures the set of program points where `buf_lock` is held. The *RacyRead* function is defined as:

$$\begin{aligned} \mathit{RacyRead}(r, l) \quad = \quad & (l = \texttt{px->data} \wedge \texttt{buf\_lock} \notin r) \vee \\ & (l = \texttt{cx->data} \wedge \texttt{buf\_lock} \notin r) \end{aligned}$$

which captures the fact that an access of `px->data` in `Producer` or `cx->data` in `Consumer` is racy at any point where the lock `buf_lock` is not held, and that all other accesses are non-racy. It is easy to check that the *Reg* and *RacyRead* functions soundly approximate the possible races and pseudo-races. The adjusting at program point `P5` kills the fact *NonNull*(`px->data`) since

$$\mathit{RacyRead}(\mathit{Reg}(\texttt{P5}), \texttt{px->data}) = \mathit{true}$$

In the correct version from Figure 2.1, the absence of the unsafe lock operations changes the *Reg* map to:

$$\mathit{Reg}(p) = \begin{cases} \{\texttt{buf\_lock}\} & \text{if } p \in \{\texttt{P3}, \texttt{P5}, \texttt{P6}, \texttt{P8}, \texttt{P9}\} \\ \{\texttt{buf\_lock}\} & \text{if } p \in \{\texttt{C3}, \texttt{C4}, \texttt{C5}, \texttt{C6}, \texttt{C7}\} \\ \emptyset & \text{otherwise} \end{cases}$$

reflecting the fact that in this program, `buf_lock` is held throughout from `P3` to `P9`. The *RacyRead* function remains the same as before, as the synchronization discipline is unchanged: as in Figure 2.1, `buf_lock` is held at all points where the buffer cells are written, namely `P3`, `P8`, and `C5`. In the fixed program, the adjusting at $p \in \{\texttt{P5}, \texttt{P6}, \texttt{P8}\}$ does not kill the fact *NonNull*(`px->data`), as for each of these $p$

$$\mathit{RacyRead}(\mathit{Reg}(p), \texttt{px->data}) = \mathit{false}$$

**Example: Predicates.** Another possible instantiation of regions consists of having $R$ be a set of predicates *Pred*. A race equivalence region is then a predicate from *Pred*, and the set of program points in the region are those program points at which the predicate holds. The predicate instantiation is more general than the lockset instantiation because we can encode a set of locks as a predicate stating that all the locks in the set are held.

Recall the version of the `Producer-Consumer` program shown in Figure 2.3, where the synchronization is performed via a `flag` field and not explicitly declared locks. As shown in [HJM04], one can generalize race regions to an *access predicate* describing the thread's state. For the example in Figure 2.3, the *Reg* map is:

$$Reg(p) = \begin{cases} \texttt{px->flag} = 0 & \text{if } p \in \{\texttt{P3}, \texttt{P5}, \texttt{P6}, \texttt{P8}, \texttt{P9}\} \\ \texttt{cx->flag} \neq 0 & \text{if } p \in \{\texttt{C3}, \texttt{C4}, \texttt{C5}, \texttt{C6}, \texttt{C7}\} \\ \mathit{true} & \text{otherwise} \end{cases}$$

The *RacyRead* function is defined as:

$$\begin{aligned} RacyRead(\varphi, l) = \quad & (l = \texttt{px->data} \wedge \varphi \not\Rightarrow \texttt{px->flag} = 0) \vee \\ & (l = \texttt{cx->data} \wedge \varphi \not\Rightarrow \texttt{cx->flag} \neq 0) \end{aligned}$$

Together, *Reg* and *RacyRead* capture the intuition that a read of `px->data` in `Producer` (respectively `cx->data` in `Consumer`) is racy at any point where the `px->flag` is zero (respectively `cx->flag` is zero).

### 2.2.4 Inter-procedural Framework

The RADAR framework presented so far does not take function calls into account. To understand how function calls affect our basic framework, consider the example from Figure 2.5, which is a version of the `Producer-Consumer` example where there is a call to a function `foo` right before the increment of `perf_ctr`.

Let us assume that `foo` itself does not modify `px->data`, and that the sequential dataflow analysis uses a simple "modifies" analysis to determine this. As a result, the sequential dataflow analysis is able to propagate *NonNull*(`px->data`) from `P4` to `P5` (that is to say, through the call to `foo`). However, in the face of concurrency, even if `foo` itself does not modify `px->data`, a call to `foo` while other threads are running can in fact lead to `px->data` being modified. In particular, calling `foo` has the effect of unlocking `buf_lock` and then re-locking it, which gives concurrent threads an opportunity to modify `px->data`. As a result, the adjusted flow function needs to kill *NonNull*(`px->data`) at `P5`, as well as any dataflow information about `px->data`.

| *Adjusted Analysis* | |
|---|---|
| | `buffer_list *bufs;` |
| | `lock buf_lock;` |
| | `int perf_ctr;` |
| | `thread producer1(){` |
| ∅ | `P0: px = bufs;` |
| ∅ | `P1: while (px != NULL){` |
| *px* | `P2:  lock(buf_lock);` |
| *px* | `P3:  px->data = new();` |
| | |
| *px->data,px* | `P4:  foo();` |
| ~~*px->data*~~*,px* | `P5:  perf_ctr++;` |
| ~~*px->data*~~*,px* | `P6:  t=produce();` |
| | |
| | |
| ~~*px->data*~~*,px* | `P8:  *px->data = t;` |
| ~~*px->data*~~*,px* | `P9:  unlock(buf_lock);` |
| ~~*px->data*~~*,px* | `PA:  px = px->next;` |
| | `}` |
| | `foo(){` |
| ∅ | `G0: unlock(buf_lock);` |
| ∅ | `//do work` |
| ∅ | `G1: lock(buf_lock);` |
| | `thread consumer1(){` |
| ∅ | `perf_ctr = 0;` |
| ∅ | `C0: cx = bufs;` |
| ∅ | `C1: while(cx != NULL){` |
| *cx* | `C2:  lock(buf_lock);` |
| *cx* | `C3:  if(cx->data != NULL){` |
| *cx->data,cx* | `C4:  consume(*cx->data);` |
| *cx->data,cx* | `C5:  cx->data = NULL;` |
| *cx* | `C6:  cx = cx->next;` |
| | `}` |
| ∅ | `C7:  unlock(buf_lock);` |
| | `}` |

Figure 2.5: `Producer-Consumer` with function calls

Unfortunately, with the definition of *ThreadKill* given so far, the adjusted flow function would not do this. In particular, the adjusted flow function would ask *ThreadKill* if `px->data` could be written concurrently at the program point right after `foo` returns (which is `P5`). *ThreadKill* in turn would ask the race detection engine if a read of `px->data` would cause a race at `P5`. The race detection engine would answer back saying "no race" since by the time `foo` returns, the lock protecting `px->data` would already have been re-acquired. As a result, the information about `px->data` being non-null would incorrectly survive the adjusting process.

The problem in the example above is that the execution of `foo` passes through a region that does not hold `buf_lock`, which allows concurrent threads to modify `px->data`, and callers of `foo` must take this into account. More broadly, the problem is that the execution of a function can pass through a *variety* of race equivalence regions, and callers need a way to summarize the effect of having passed through all of the callee's regions.

**Summary Regions.** To address this problem, we add to RADAR a new function called $SumReg : CS \rightarrow R$, which returns for each call-site an *inter-procedural summary region*. This call-site-specific summary region is meant to approximate the possible regions that the callee can go through when invoked at the given call-site. We denote by $CS$ the set of all call-sites, and we define the call-site of a call node to be the CFG edge that immediately follows the call node (so that $CS \subseteq PPoint$).

**Soundness.** Intuitively, for soundness we require that for every lvalue $l$, if a read of $l$ is racy at *some* program point transitively reachable during the call made at $cs$, then the summary region for the call-site $cs$ must be a region that is racy for $l$. Thus, to formalize soundness in a context-sensitive manner, we extend the perfect race detection oracle to $RealRace : CS \times PPoint \times Lvals \rightarrow Bool$, such that $RealRace(cs, p, l)$ returns *true* exactly when there is an execution in which a read of $l$ at $p$ while $cs$ is on the callstack would cause a race. Let $cs^*$ be the set of program points in the function being called at call-site $cs$ and any of its transitive callees. The soundness requirement for $SumReg$ can be formally stated as:

$$\begin{aligned} &\forall cs \in CS, l \in Lvals \ . \\ &\quad [\exists p \in cs^* \ . \ RealRace(cs, p, l)] \Rightarrow \\ &\qquad RacyRead(SumReg(cs), l) \end{aligned} \tag{2.3}$$

Having defined $SumReg$ and its soundness property, we can now define the $ThreadKill$ function as follows:

$$\begin{aligned} ThreadKill(p, l) = \ & RacyRead(Reg(p), l) \ \vee \\ & (RacyRead(SumReg(p), l) \wedge p \in CS) \end{aligned}$$

**Instantiation Requirements:** To instantiate the inter-procedural version of the RADAR framework, one needs to provide:

1. A **race detection engine** $RacyRead : R \times Lvals \rightarrow Bool$
2. A **region map** $Reg : PPoint \rightarrow R$
3. A **summary map** $SumReg : CS \rightarrow R$

such that *RacyRead* and *Reg* satisfy property (2.2), and *SumReg* satisfies property (2.3). We now go through the same two instantiation from Section 2.2.3, and show how *SumReg* can be defined.

**Example: Global Locksets.** If we instantiate regions as locksets over a global set of locks *Locks*, the summary of a function is the intersection of all the locksets that the function goes through. As a result, in the example from Figure 2.5, the *SumReg* function is defined as:

$$SumReg(\texttt{P5}) = \emptyset$$

since the unlock in `foo` causes the lockset to be empty at `G1`, and thus the intersection of all locksets in `foo` is empty. The *Reg* and *RacyRead* functions are the same as in Section 2.2.3.

With these definitions, the adjusting process now correctly kills the fact *NonNull*(`px->data`) at `P5` , since, even though

$$RacyRead(Reg(\texttt{P5}), \texttt{px->data}) = false$$

we have

$$RacyRead(SumReg(\texttt{P5}), \texttt{px->data}) = true$$

which, combined with $\texttt{P5} \in CS$, means

$$ThreadKill(\texttt{P5}, \texttt{px->data}) = true$$

**Example: Predicates.** If we use a set *Pred* of predicates for the race equivalence regions, the summary of a function is the disjunction of all the predicates in the function.

### 2.2.5 Limitations

The extension for inter-procedural analysis and the optimization of race equivalence regions are certainly important for RADAR to be realistic. One area

that has not yet been discussed is the precision of the framework itself (as opposed to its components).

**Non-semantic** *ThreadKill***.** It is possible that in some cases, *ThreadKill* is conservative even when *RacyRead* is perfectly precise (it is equivalent to *RealRace*). Consider a slight change in the buggy `Producer-Consumer` example (Figure 2.2), where the racy write to `cx->data` does not involve the assignment of `null`, but is instead assigned the non-null address of a dummy sentinel. There is no longer a `null` pointer violation because only non-null values are written to the data field. However, the example is still buggy, as values can now be inadvertently written through `*px->data` into the dummy sentinel.

For the case of the non-null dataflow problem it is possible to use the non-nullness dataflow information that is present at the site of the write to improve precision. However it is unclear if such increase in precision is useful, as shown in the above example it can mask other bugs. Furthermore, the approach of using the client's dataflow information to better interpret racy writes does not apply to all clients. For example, not all dataflow problems compute information that can be interpreted when taken from the site of the racing write. For example, the dataflow facts computed by the available expressions analysis would not clarify how a possible racing write in one procedure may affect available expressions in another procedure (for one, the analysis is intra-procedural).

**Cross-thread Propagation of Facts.** In addition ignoring semantic information when deciding how to handle racing writes, RADAR does not allow one thread to *generate* dataflow facts for another thread. Figure 2.6 contains a few examples. The first example shows that if a parent thread `join`s a child thread which sets a global variable `p` to a non-null value before exiting, it is more precise to say that `p` is non-null in the parent thread after the `join`. In general, this communication can happen at any point in the lifetime of each thread (*e.g.* through the use of condition variables). Such cases are not handled by RADAR but are handled by other frameworks that are limited to more structured concurrent programs (see related work in chapter 6). However, it is not hard to imagine how to extend RADAR to at least handle the cases in figure 2.6.

```
int *p = null;

void foo(){
  tid = spawn(&bar);
  join (tid);
  use(*p);
}

void bar(){
  p = new int(1);
}
```

```
int *p = null;

void foo(){
  p = new int(2);
  spawn(&bar);
  //...
}

void bar(){
  use(*p);
}
```

Figure 2.6: Simple cases where one thread may generate facts for another.

**Dataflow Facts for Pruning Races.** The two previous limitations involve information that is available from a client dataflow analysis not being fully used or propagated by part of the framework. This third limitation is similar and involves the concurrency analysis, the race detection engine, not using all of the information that may be available from the client dataflow analysis. Figure 2.7 contains such an example, where we are concerned about the dereference on line `p4`. The programmer intends that the allocation on line `p2` should guarantee that `p` is non-null. The sequential dataflow analysis will assert that fact and Radar will insert a pseudo-read of `p` at lines `p3` and `p4`. However, if Radar uses a lockset-based data-race detector it will find that the pseudo-reads race with the write at line `q3`. However, in no execution of the program does the write at line `q3` actually occur. If a dataflow analysis is able to derive a sound invariant that at the branch on `q2`, the global variable is always non-null, and the data-race detector is able to use that information, then it will be able to prune out that write and therefore prune the race. However, deriving sound dataflow facts is the very problem that we are trying to solve in Radar.

This lack of communication between the dataflow analysis and the concurrency analysis in Radar is only a limitation if the data-race detector itself is unable to prune such impossible execution paths or program interleavings. For example, the non-lockset-based data-race engine based on model checking [HJM04]

```
Thread 1                    Thread 2
 p1: void foo(){             q1: void bar(){
 p2:   p = new int(3);       q2:   if (p == null) {
 p3:   spawn(&bar);          q3:     p = new int(3);
 p4:   use(*p);              q4:   }
 p5: }                       q5: }
```

Figure 2.7: Dataflow information can refine data-race detector.

```
Thread 1                    Thread 2
 p1: void foo(){
 p2:   x = 1;                q1: void bar(){
 p3:   spawn(&bar);          q2:   x = 0;
 p4:   print(10/x);          q3: }
 p5: }
```

Figure 2.8: Example *May* analysis: May be zero.

would not need to rely on the dataflow analysis.

### 2.2.6 May Analyses and Backwards Analyses

**May Analyses.** In the introduction of the basic framework, it was suggested that may analyses be computed by encoding the analysis as a must-not analysis and checking for the absence of facts. There are two reasons for this besides clarity of exposition. First, formulating a may analysis as the negation of a must-not analysis makes it clear how to insert pseudo-reads based the dataflow facts at a program point without implicitly encoding those lvalues into the *lvals* function of the sequential dataflow analysis. Second, depending on how the analysis's flow function $F$ handles "havoc" cfg nodes ($l := \perp$), it may be the case that the definition of $F_{Adj}$ would need to change such that racy writes generate facts instead of killing facts.

Consider a "May be Zero" analysis in Figure 2.8, which is used to check if there is a possible divide by zero at line `p4`. If one is to design such a may analysis, there are at least two options.

The first option is based on a may analysis with a domain of *MayBeZero*($l$) facts. Suppose in this case that the analysis starts with a *MayBeZero*($x$) (for the uninitialized $x$). The flow function for the assignment $x = 1$ in thread one will kill the *MayBeZero*($x$) fact, and the analysis will be left with an empty set of facts. In this case *Lvals* must be capable of conjuring up $x$ absent any facts concerning $x$. It is only if *lvals* can instruct RADAR to insert a pseudo-read that a race can be found with the write in thread two. Finally, based on the racing write, $F_{Adj}$ must be capable of introducing the *MayBeZero*($x$) fact prior to program point `p4`.

The second option is to use must not facts. In this case the assignment $x = 1$ introduces a *MustNotBeZero*($x$) fact after line `p2`. Then, *Lvals* will see the *MustNotBeZero*($x$) fact and return $x$ to insert a pseudo-read for $x$. The earlier definition of $F_{Adj}$ based on *ThreadKill* will be work as intended, and the *MustNotBeZero*($x$) fact will be killed.

**Backwards Analysis**. Two potential issues arise from the fact that RADAR can only detect *potential* concurrent *writes*. First, the writes are only potential, as it is possible that the write will not occur in any execution of the program, due to imprecision in the race detection engine. Second, RADAR does not detect concurrent reads. These two issues may be of extra concern when designing a backwards directional analysis.

Let us take Liveness Analysis as an example of a may backwards analysis. Liveness introduces facts whenever a variable is read and kills facts whenever they are (surely) written to.

First, because the writes are only potential, a variable that may be live can be inadvertently killed by RADAR. To remedy this, RADAR must model the program as *branching* at each program point towards either the possible racy write or into a path that skips that write. Given this more accurate model, the potential racy write will kill a live variable along one branch, but along the other branch the variable will still be live. Finally, because the confluence operator is $\cup$, the analysis can retain the live variable. Thus, $F_{Adj}$ must be defined to use the confluence operator supplied by the particular sequential analysis. Essentially, in the case of may analyses $F_{Adj} = F$, and in the case of must analyses $F_{Adj}$ is as defined earlier.

The second concern is that a variable may be live because of a read from another thread. Luckily, Liveness and other dataflow analyses are only concerned with reads executed by the current thread.

### 2.2.7 Preservation of Monotonicity and Distributivity

We conclude this chapter discussing some properties that the RADAR framework *does* enjoy.

*Monotonicity* is essential to the correctness and proof of termination for any iterative dataflow algorithm. Monotonicity alone, however, does not guarantee that the fixpoint solution is the precise meet-over-all-paths (MOP) solution. The MOP solution is precise in the sense that the final information at a program point is exactly the combination along all individual paths that can be taken to that program point. *Distributivity* is the property that ensures that the fixpoint solution from an iterative dataflow algorithm is in fact the MOP solution [KU77].

This section shows that RADAR preserves any monotonicity and distributivity present in the client's sequential flow function. In other words, if the sequential flow function is monotonic (or distributive) then the adjusted flow function is still monotonic (or distributive).

**Alternate Notation.** First, recall from section 2.2.1 that we assume that dataflow facts in *DataflowFact* are must facts, giving us the domain $D = 2^{DataflowFact}$ and the lattice $(D, \sqsubseteq, \top, \bot, \sqcap, \sqcup)$, where $\sqsubseteq$ is $\supseteq$, $\top$ is $\emptyset$, $\bot$ is *DataflowFact*, $\sqcap$ is $\cup$, and $\sqcup$ is $\cap$.

Recall that given the sequential flow function $F$, the basic adjusted flow function is defined as:

$$\begin{aligned} F_{Adj}(n, d, p) = \{f \mid \; & f \in F(n, d, p) \wedge \\ & \forall l \in Lvals(f) \; . \; \neg ThreadKill(p, l)\} \end{aligned}$$

For convenience, we will work with a curried version of the flow functions where $p$ and $n$ are fixed ($F^{(p,n)}$ is the curried sequential flow function and ${F_{Adj}}^{(p,n)}$ is the adjusted flow function). We can also alternatively define ${F_{Adj}}^{(p,n)}$ as a

composition of the functions $F^{(p,n)}$ and $FilterRacy^p : 2^{DataflowFact} \rightarrow 2^{DataflowFacts}$.

$$F_{Adj}{}^{(p,n)}(d) = FilterRacy^p(F^{(p,n)}(d))$$

$$FilterRacy^p(d) = \{f \mid f \in d \wedge \forall l \in Lvals(f)\ .\ \neg ThreadKill(p,l)\}$$

**Preservation of Monotonicity.** Now to show preservation of monotonicity, we want to prove that given sets of dataflow facts $A$ and $B$, if $A \sqsubseteq B \Rightarrow F(A) \sqsubseteq F(B)$, then $F_{Adj}(A) \sqsubseteq F_{Adj}(B)$. If we show that $FilterRacy^p$ is a monotone function as well, then we are done. Spelled out, we know $F^{(p,n)}(A) \sqsubseteq F^{(p,n)}(B)$ from the sequential TF's monotonicity, and if $FilterRacy^p$ is also monotone, then $FilterRacy^p(F^{(p,n)}(A)) \sqsubseteq FilterRacy^p(F^{(p,n)}(B))$, thus showing that $F_{Adj}{}^{(p,n)}(A) \sqsubseteq F_{Adj}{}^{(p,n)}(B)$.

Monotonicity of $FilterRacy^p$ can be proved by contradiction. Assume $A \sqsubseteq B$ and $FilterRacy^p(A) \not\sqsubseteq FilterRacy^p(B)$. Recall that $\sqsubseteq$ is defined as $\supseteq$. Then there exists a dataflow fact $f$ in $FilterRacy^p(B)$ but not in $FilterRacy^p(A)$. If $f$ is in $FilterRacy^p(B)$, we know two things:

(a) $f \in B$ and

(b) $\nexists l \in Lvals(f)$ such that $ThreadKill(p,l)$ is true.

For $f$ to not be in $FilterRacy^p(A)$ means at least one of the two propositions is true: (1) $f \notin A$, or (2) some lval $l \in Lvals(f)$ satisfies $ThreadKill(p,l)$. We know that (2) cannot be true from $f$ surviving $FilterRacy^p(B)$ (fact (b)). Thus (1) must be true – $f \notin A$. However, this contradicts the assumptions (fact (a) and the assumption that $A \sqsubseteq B$ implies that $f \in A$). $\square$

**Preservation of Distributivity.** To prove preservation of distributivity, we must prove that for any sets of dataflow facts $A$ and $B$, if $F^{(p,n)}(A \sqcup B) = F^{(p,n)}(A) \sqcup F^{(p,n)}(B)$, then $F_{Adj}{}^{(p,n)}(A \sqcup B) = F_{Adj}{}^{(p,n)}(A) \sqcup F_{Adj}{}^{(p,n)}(B)$. The proof is simple

if we prove that $FilterRacy^p$ is distributive.

$$
\begin{aligned}
& FilterRacy^p(A \sqcup B) \\
&= FilterRacy^p(A \cap B) \\
&= \{f \mid f \in (A \cap B) \wedge \forall l \in Lvals(f) \ . \ \neg ThreadKill(p, l\} \\
&= \{f \mid f \in A \wedge \forall l \in Lvals(f) \ . \ \neg ThreadKill(p, l\} \\
&\quad \cap \{f \mid f \in B \wedge \forall l \in Lvals(f) \ . \ \neg ThreadKill(p, l\} \\
&= FilterRacy^p(A) \cap FilterRacy^p(B) \\
&= FilterRacy^p(A) \sqcup FilterRacy^p(B). \quad \square
\end{aligned}
$$

Given that $FilterRacy^p$ is distributive we can then show that Radar preserves distributivity.

$$
\begin{aligned}
& {F_{Adj}}^{(p,n)}(A \sqcup B) \\
&= FilterRacy^p(F^{(p,n)}(A \sqcup B)) \\
&= FilterRacy^p(F^{(p,n)}(A) \sqcup F^{(p,n)}(B)) \\
&= FilterRacy^p(F^{(p,n)}(A)) \sqcup FilterRacy^p(F^{(p,n)}(B)) \\
&= {F_{Adj}}^{(p,n)}(A) \sqcup {F_{Adj}}^{(p,n)}(B). \quad \square
\end{aligned}
$$

## 2.3 Summary

This chapter presented an overview of the Radar framework for converting a sequential dataflow analysis into one that is sound in a concurrent setting. The basic algorithm, an inter-procedural algorithm, and an optimization involving race equivalence regions were all presented.

The main benefit of this approach is that all reasoning about concurrency is cleanly separated out into a race detection engine. The race detection engine and the sequential dataflow analysis modules can then be independently fine-tuned; each module can be improved in the areas of precision or scalability, without changing the other. We show how Radar can work with a scalable but imprecise lockset-based race detector as well as a precise but non-scalable race detector based on predicates. Furthermore, the sequential dataflow analysis does not need to be modified if new concurrency constructs are introduced to the programming language or environment – one need only modify the race detection engine.

A trade-off of this clean separation is that beneficial information from one module (say, the sequential dataflow analysis) may not be fully communicated to and utilized by the other module (say, the race detection engine).

Experimental evaluation is deferred to Chapter 5, after the other components required for RADAR are introduced.

**Acknowledgments:** Chapter 2 in part, has been published as "Dataflow Analysis for Concurrent Programs using Datarace Detection" by Ravi Chugh, Jan Voung, Ranjit Jhala, and Sorin Lerner in *PLDI 08: Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation* [CVJL08]. The dissertation author was the primary investigator and author of this paper.

# Chapter 3

# The Relay Data-Race Detector

This chapter begins with an overview of the different components of Relay and how they work, as illustrated through an example (Section 3.2). Section 3.3 presents the basic algorithm in more detail. Details on how to optimize the basic algorithm are presented in Section 3.4. Finally, Section 3.5 evaluates Relay as a standalone race detector. Part of the evaluation involves determining coding idioms for which Relay is imprecise and estimating the prevalence of each idiom. Such estimates can help prioritize the different areas in which to improve Relay. In a later chapter (5), we evaluate the combination of Relay and the Radar framework.

## 3.1 Motivation and Contributions

One of the requirements of Radar is a sound, static, and scalable data-race detector. These three goals have previously never been achieved all at once. In particular, while sound and static race detection techniques have proven to be effective, the largest programs they have ever been applied to are on the order of tens of thousands of lines of C code [PFH06, Ter08] and little over a hundred thousand lines of Java code [NAW06]. Furthermore, while some static race detection algorithms run on millions of lines of code [EA03], they are extremely unsound, and therefore miss many errors.

We take a step towards achieving all three goals by developing Relay, a

static and scalable algorithm that can perform race detection on programs as large and complicated as the Linux kernel (which comprises 4.5 million lines of C code). In Relay, unsoundness is *modularized* to the following sources: (1) it does not interpret assembly code (2) it does not handle corner cases of pointer arithmetic, and (3) its inter-procedural analysis is dependent on a call-graph that may be unsound (does not resolve all function pointer targets). Factors (1) and (2) are shared by other data-race detectors for C code [PFH06, Ter08]. The third factor is important but can be made sound – Relay does have hooks to sound call-graph construction algorithms. We revisit the issue of call-graphs in Chapter 4.

The standard mechanism for preventing data races is to ensure that for each shared lvalue, there exists a unique *lock* that is held whenever a thread accesses the lvalue. Since only one thread can hold a lock at any given time, we can ensure the absence of races. One family of algorithms for inferring whether such a lock exists is that based on computing *locksets*. These algorithms determine either statically [DS91, EA03, Ste93] or dynamically [SBN+97, CLL+02, YRC05] the set of locks held by the program at every program point. If the intersection of the locksets at each point the lvalue is accessed is non-empty then there are no races on the lvalue. If the intersection is empty, the analysis conservatively reports that there may be a race on the lvalue.

While static lockset based techniques have proven to be effective for race analysis, there are significant hurdles that must be crossed to scale them to millions of lines of systems code. First, it is difficult to tell which piece of memory a given operation will actually affect. The lvalue being accessed need not be in global scope — it may have been passed into the function as a parameter, and thus, the actual memory accessed can only be determined by a careful, calling-context-sensitive analysis. Second, for similar reasons, it is difficult to tell which locks are held at each point, as it is hard to tell exactly what locks are acquired and released by various operations as the locks may be derived from structures passed in as parameters. As a result, it becomes hard to tell if the set of locks held at two different accesses are the same, as the locks may have very different syntactic names. Third, in low-level systems code, the acquisition and release of locks is

not syntactically nested (as is the case in Java). A lock may be acquired in one function, the access may happen in a second function, and the lock may be released in a third function. As a result, many modular type-based [FF00, BLR02] and flow-insensitive [NAW06, NA07] approaches cannot be applied in this setting, and instead a precise flow-sensitive approach is required.

The technical contribution of Relay is a technique for addressing the above limitations. In particular, we introduce the concept of a *relative lockset*, which describes the changes in the locks being held relative to the function entry point. These relative locksets allow us to summarize the behavior of a function independent of the calling context. For example, the summary of a function whose formal is `x` may say that the field `x->f` is accessed inside the function while holding all locks that were held on entry, plus `x->lock1`, and minus `x->lock2`. The information about this guarded access is not absolute — it is relative to the locks held at the entry point, which allows Relay to plug the summary in while analyzing any callers.

This switch to relative locksets, rather than absolute locksets, is the key to scalability: relative locksets allow us to aggressively exploit modularity. In particular, Relay analyzes functions in isolation to compute summaries that capture the behavior of a function for any calling context, and then it composes these summaries to determine whether races exist. This leads to a bottom-up context-sensitive analysis over the call-graph that scales to programs as large as the Linux kernel. The modularity also enables easy parallelization. Because functions can be analyzed independently, we can run our analysis of the Linux kernel using a cluster of machines in about 5 hours, as opposed to the 72 hours it takes without parallelization.

## 3.2 Overview of Relay

This section presents an overview of the Relay race detection algorithm using the example code from Figures 3.1 and 3.2. The code in the figures are simplified versions of two functions from the Linux kernel. Figure 3.1 shows the

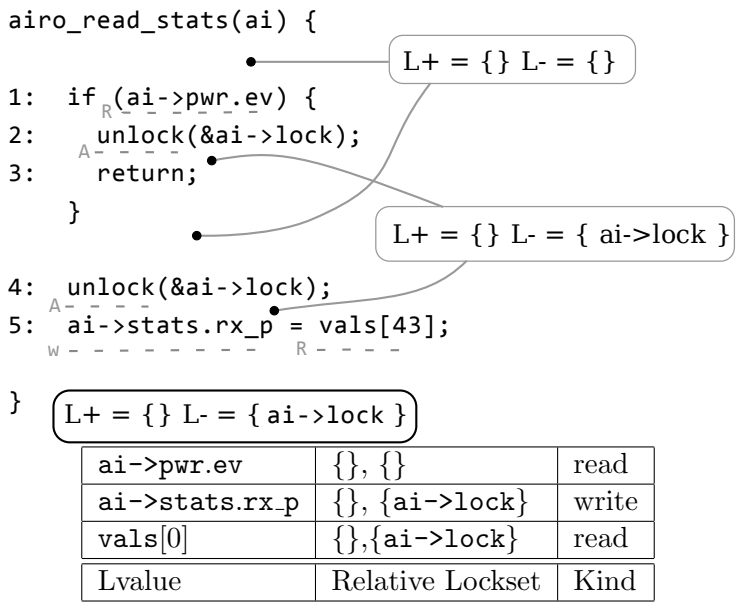



Figure 3.1: Callee expects an acquired lock and releases the lock (from Linux).

function `airo_read_stats`, which takes as input a single parameter `ai` that is a reference to a complex structure. Figure 3.2 shows the function `airo_thread`, which takes a pointer to a device structure `d`. The latter function is a *thread entry point*, *i.e.,* it is a function where some thread begins execution. In the sequel, suppose that multiple threads can begin executing concurrently from this entry point. The structure that `d` refers to is a shared structure — it may be accessed outside the thread running `airo_thread`, and thus, it can possibly be accessed by multiple concurrently running threads. Similarly, because the `vals` array used on line `5` is declared to be global, it can be accessed by different threads, and so it is shared.

Locks are acquired and released by calling appropriate functions on the lock arguments. Thus, in the code from Figures 3.1 and 3.2, we can deduce (assuming that `airo_read_stats` is called only from within `airo_thread`), that the lock `d->priv->lock` is held (acquired in `airo_thread`) whenever the lvalue `d->priv->pwr.ev` is accessed (in `airo_read_stats`). Since line `1` is the only place the lvalue is accessed, we can therefore conclude there are no races on the lvalue. On the other hand, when `d->priv->stats.rx_p` is written to on line `5`, there is

```
airo_thread(d) {
                                L+ = {} L- = {}
6: dev = d;
7: ai = dev->priv;
8: lock(&ai->lock);             L+ = { d->priv->lock } L- = {}
9: airo_read_stats(ai);
                                L+ = {} L- = { d->priv->lock }
}
```

L+ = {} L- = { d->priv->lock }

| d->priv | {}, {} | read |
|---|---|---|
| d->priv->pwr.ev | {d->priv->lock}, {} | read |
| d->priv->stats.rx_p | {}, {d->priv->lock} | write |
| vals[0] | {}, {d->priv->lock} | read |
| Lvalue | Relative Lockset | Kind |

Figure 3.2: Caller acquires locks before calling function (continued example).

no lock that is held, and so the write may cause a race, since two or more threads could simultaneously perform the write.

### 3.2.1 Ingredients that Enable Modular Analysis

We devised a precise analysis that scales to millions of lines of code by aggressively exploiting modularity *i.e.,* by analyzing functions in isolation to compute summaries that capture the behavior of the function independent of the calling context, and then composing the summaries to determine whether races exist. In particular, our algorithm for race analysis is built using the following ingredients.

**1. Relative Locksets:** A *relative lockset* at a location is a disjoint pair of locksets $(L_+, L_-)$ (resp. called the positive and negative locksets), which encodes the *difference* between the locks held at the given location and the locks held at the function entry location. Intuitively, the set $L_+$ is the set of additional locks that are definitely acquired on *all* executions from the entry to the location. The set $L_-$ is the set of all locks that may have been released on *some* execution from the entry to the location. It is important to remember that $L_+$ is a *must* set and that $L_-$ is a *may* set.

**2. Guarded Accesses:** A *guarded access* is a triple of an *lvalue*, the *relative lockset* at the program location where the access takes place and the *kind* of access, either a *read* or a *write*. The set of guarded accesses of a function is the set of triples corresponding to accesses that may occur during the execution of the function. RELAY works by computing an over-approximation of the set of guarded accesses of each thread entry point. Once this set is computed, RELAY compares pairs of guarded accesses whose lvalues may be aliased. For each such pair, RELAY determines if the intersection of the positive locksets is empty, and if so, reports a race warning.

**3. Function Summaries:** To compute the guarded accesses for each thread entry point, RELAY builds the call-graph and traverses it in a bottom up manner, computing the guarded accesses for each function along the way. To this end RELAY computes *two* summaries for each function. The first is a *relative lockset summary*, which is the relative lockset of the exit location of the function. This summary soundly approximates the effect the function has on the set of locks held by the thread just *before* calling the function. The second is a *guarded access summary*, which is a set of guarded accesses that includes the guarded accesses that may occur during the execution of the function. RELAY computes the summaries in a bottom-up manner, plugging in the summaries of the *callees* at call-sites to compute the guarded access summaries and relative lockset of the *callers*.

**4. Symbolic Execution:** In order for a summary to capture the behavior of a function regardless of the calling context, the summary must be expressed in terms of the formals of the function, or in terms of globals. In this way, the summary can be instantiated at a call site by replacing formals in the summary with the actuals passed in at the call site, thus producing information in the caller's context. As an example, for the `airo_thread` function, we want to compute a summary stating that `d->priv->lock` is held when `d->priv->pwr.ev` is accessed, not that `ai->lock` is held when `ai->pwr.ev` is accessed. To build such summaries, one must re-express accesses inside a function in terms of the globals and the formals. To this end, RELAY performs an intra-procedural symbolic execution that maps each local lvalue to a value expressed in terms of the incoming values of formals, and

the incoming values of globals. With appropriate join operators to handle merge nodes, we can handle loops while preserving termination.

### 3.2.2 Putting the Ingredients Together

The above are combined in the RELAY modular race analysis tool as follows. RELAY processes functions bottom-up in the call-graph, starting with leaves, and working its way up the call-graph. RELAY repeatedly picks a function to analyze amongst all the functions whose callees have been analyzed.

For each function being analyzed, RELAY performs three analyses: first a symbolic execution, second a relative lockset analysis, and third a guarded access analysis. The symbolic execution is used to express the values contained in memory locations in terms of the incoming values of the formals and the globals. This symbolic information is required by the two subsequent analyses. The relative lockset analysis is an iterative dataflow analysis that maintains a relative lockset at each program point. For call sites, the analysis uses the summaries of callees to compute the lockset after the call. Once a fixed point is reached, the relative lockset computed for the function's exit point becomes the relative lockset summary of the function. After performing the relative lockset analysis on a function, RELAY runs a guarded access analysis on that same function. The guarded access analysis maintains a monotonically increasing set $A$ of guarded accesses. The analysis iterates through all the statements in the function (in a flow-insensitive way), accumulating in $A$ the locations being accessed, along with the locks being held during those accesses. The information about which locks are held at the access points is provided by the results of the relative lockset analysis.

Consider the program comprising the two functions shown in Figures 3.1 and 3.2. RELAY begins with the leaf function `airo_read_stats`. Initially, the relative lockset at the entry point is the pair $(\{\}, \{\})$. The calls to `unlock` result in the addition of `ai->lock` to the negative lockset of program points `3` and `5`. Because negative locksets are *may* sets, the negative lockset of the exit point is the *union* of those of its predecessor program points `3` and `5`, namely `ai->lock`. Because positive locksets are *must* sets, the positive lockset is the *intersection*

of those of the predecessors, which is the empty set. Thus, the relative lockset summary of the function is the pair: $(\{\}, \{\texttt{ai->lock}\})$. This summary states that the `airo_read_stats` function does not acquire any locks, and it may release (in fact in this case, it definitely does release) the `ai->lock` lock.

After computing the relative lockset information for `airo_read_stats`, Relay iterates through the statements of `airo_read_stats` to find the guarded access set of the function. There are three accesses in this function: the read of `ai->pwr.ev` on line `1`, with a relative lockset of $(\{\}, \{\})$; the read of the `vals` array on line `5`, with a relative lockset of $(\{\}, \{\texttt{ai->lock}\})$; and the write to `ai->stats.rx_p` with the same lockset. This information is collected in the guarded access summary of the function, which is shown in the table of Figure 3.1 (the index 0 in `vals`[0] represents all array indices).

Next, Relay picks the function `airo_thread`. The relative lockset for the entry location is the same as for `airo_read_stats`, namely $(\{\}, \{\})$. The symbolic execution tracks that at `7` the lvalue `dev` refers to the formal `d`, and that at `8` the lvalue `ai->lock` refers to `d->priv->lock`. As a result, the relative lockset at `8` (just before the call to `airo_read_stats`) is $(\{\texttt{d->priv->lock}\}, \{\})$, indicating that `d->priv->lock` was added since the entry point of the function. Now the call to `airo_read_stats` is analyzed. The relative lockset summary for `airo_read_stats` is $(\{\}, \{\texttt{ai->lock}\})$, and since the symbolic execution tells us that `ai` is in fact `d->priv`, the instantiated summary in the caller's context is $(\{\}, \{\texttt{d->priv->lock}\})$. Updating the information from before the call $(\{\texttt{d->priv->lock}\}, \{\})$, with the effect of the call $(\{\}, \{\texttt{d->priv->lock}\})$ gives us the information $(\{\}, \{\texttt{d->priv->lock}\})$ after the call. In particular, after the call, we have lost the information we had previously about `d->priv->lock` being held, because `airo_read_stats` releases that lock. Furthermore, we gain the information that since the beginning of execution of `airo_thread`, the lock `d->priv->lock` may end up being released because of the call to `airo_read_stats`. Since the call is the last statement in `airo_thread`, the information after the call becomes the relative lockset summary for `airo_thread`.

Next Relay processes all the statements in `airo_thread` to compute its

guarded access set. There is only one access in `airo_thread` itself, namely the read of `dev->priv` at line `7` (the write to `ai` is not recorded because `ai` is a local stack variable). Using the symbolic information, this read access is recorded in the guarded access summary (shown in the table of Figure 3.2) as an access to `d->priv` with relative lockset $(\{\}, \{\})$. The other accesses in the guarded access summary of `airo_thread` are added when the call to `airo_read_stats` is processed. Since the relative lockset of the `ai->pwr.ev` entry in the summary is $(\{\}, \{\})$ (*i.e.,* no locks were added or removed), the relative lockset for this access in the caller is just the relative lockset at the call-site, $(\{\texttt{d->priv->lock}\}, \{\})$. For the other two accesses in the summary of `airo_read_stats`, the negative lockset contains `ai->lock`, which, after plugging in the actuals corresponds to `d->priv->lock` as shown in the last two rows of the guarded access set shown in the figure.

**Race Warnings.** Once Relay has computed the guarded access summaries for all functions that are thread entry points, it reports warnings for all pairs of accesses, where the lvalues may be aliases, and whose positive locksets have an empty intersection, and where at least one of the accesses is a write. Suppose that a sound alias analysis shows that the lvalues corresponding to the accesses shown on the table of Figure 3.2 have no other aliases. In this case, Relay reports that:

1. There are no races due to concurrent accesses to `vals` or `d->priv` as both accesses would be reads.

2. There are no races due to concurrent accesses to `d->priv->pwr.ev` as the intersection of the positive locksets is non-empty.

3. There may be a race involving concurrent accesses to `d->priv->stats.rx_p` in different threads, as the intersection of the positive locksets is empty. The accesses involved in this race are the access on line 5, but from two different threads.

## 3.3 The Relay Algorithm

This section describes the race detection algorithm in detail. As outlined in Section 3.2, our algorithm performs a bottom-up analysis that has three interacting components: a symbolic execution (Section 3.3.1), an analysis that computes lockset changes (Section 3.3.2), and an analysis that computes guarded accesses (Section 3.3.3). After the bottom-up analysis has finished running, the results are used to generate warnings (Section 3.3.4).

### 3.3.1 Symbolic Execution

Our symbolic execution analysis is a dataflow analysis that keeps track of the values contained in memory locations in terms of the incoming values of the formals and the globals. This analysis is fairly standard, and the details of the symbolic execution are orthogonal to the contribution of our work, so we only present an overview of our analysis here. The domains of the symbolic execution are shown Figure 3.3. We use metavariable $x \in \mathit{Vars}$ to denote formals and globals, and metavariable $p \in \mathit{Reps}$ to denote representative nodes which will be explained later. The set $\mathit{Lvals}$ of symbolic lvalues denotes the locations that our symbolic execution analysis keeps track of; this includes formals, globals and field/pointer accesses through these. We use $ls \in 2^{\mathit{Lvals}}$ to represent a set of lvalues. The set $\mathit{SymVals}$ of symbolic values denotes the values that our symbolic analysis computes, and these include: $\bot$, which means "not assigned yet"; $\top$, which means "any possible value"; $i$, which represents a constant integer; $\mathit{init}(l)$, which denotes the incoming value of lvalue $l$; $\mathit{must}(l)$, which represents a value that must point to lvalue $l$; and $\mathit{may}(ls)$, which represents a value that may point to any of the lvalues in $ls$. Finally, a symbolic execution map $\sigma \in \Sigma$ is a function from symbolic lvalues to symbolic values.

The symbolic execution keeps track of a symbolic map at each program point, and this symbolic map is updated using flow functions. The flow function for a simple assignment $x := e$ evaluates $e$ in the current map to a symbolic value, and then updates $x$ in the map. For assignments through pointers, namely $*x := e$,

the flow function evaluates $x$ to a symbolic value $v_1$ and $e$ to a symbolic value $v_2$. Which lvalues are updated in the store depends on the value $v_1$. For example, if $v_1$ is $must(o)$, then only $o$ is updated to the value $v_2$. As another example, if $v_1$ is $may(ls)$, then all the lvalues in $ls$ are updated to the value $v_2$. The remaining cases, not shown here, are very similar in nature.

**Concurrent Modifications.** When performing symbolic execution on a given function, we must account for the effect that other threads may have on the state of shared memory (the very problem we are trying to solve with RADAR!). Instead of using RADAR at this point, we use a flow-insensitive pointer analysis (PTA) to compute the set of locations that may escape the current thread, which means they could potentially be accessed by another thread. Because PTA is flow-insensitive, it already accounts for concurrent writes and doesn't suffer from the same problem. These shared lvalues are havocked (set to $\top$) after each invocation of the symbolic flow function. Essentially, we have manually performed the work of RADAR to obtain $F_{Adj}$ by using an escape analysis to answer *RacyRead* queries. This approach to handling inter-thread interference can be very conservative. However, it retains full precision on non-escaping local variables which are the most important locations to keep track of when re-expressing accesses in a function in terms of its formal parameters. For example, on line `8` of Figure 3.2, our symbolic execution is able to conclude that the lock being acquired is `d->priv->lock` because the variables involved, namely `dev` on line `6` and `ai` on line `7`, are non-escaping local variables.

**Handling Function Calls and $\top$.** Rather than generate function summaries for the symbolic execution, we handle function calls in the same way as concurrent modifications – by mapping modified lvalues to $\top$ (based on a mods analysis).

One issue is that, eventually, lvalues that are mapped to $\top$ will be dereferenced. If a pointer $l$ with symbolic value $\top$ is ever dereferenced, we ask the flow-insensitive pointer analysis (PTA) for a "representative node" for $*l$, which can be used in later alias queries (during the symbolic execution pass as well as during the data-race warning pass of RELAY). Our implementation supports both a field-insensitive Steensgaard's analysis [Ste96a] and field-insensitive Andersen's

| | | |
|---|---|---|
| formals, globals | $x \in$ | $Vars$ |
| PTA reps | $p \in$ | $Reps$ |
| symbolic lvalues | $l \in Lvals ::=$ | $x \mid x.f \mid p.f \mid (*l).f$ |
| symbolic values | $v \in SymVals ::=$ | $\top \mid \bot \mid i \mid init(l) \mid$ |
| | | $must(l) \mid may(ls)$ |
| symbolic map | $\sigma \in$ | $\Sigma = Lvals \rightarrow SymVals$ |

Figure 3.3: Symbolic analysis domain

analysis [And94] as the PTA.

### 3.3.2 Lockset Analysis

After the symbolic execution has finished, Relay runs an analysis to compute *relative* locksets held by a thread at each program point. A relative lockset $L$ in a function $f$ is a pair $(L_+, L_-)$, where the set $L_+ \subseteq Lvals$ represents the locks that have definitely been acquired since the beginning of $f$, and the set $L_- \subseteq Lvals$ represents the locks that may have been released since the beginning of the function $f$. We denote by $Locks = 2^{Lvals \times Lvals}$ the set of all relative locksets.

The lockset analysis is a dataflow analysis whose domain is the lattice $(Locks, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$, where the ordering is defined as:

- $\bot = (Lvals, \emptyset)$, $\top = (\emptyset, Lvals)$
- $(L_+, L_-) \sqsubseteq (L'_+, L'_-)$ iff $L'_+ \subseteq L_+ \wedge L_- \subseteq L'_-$
- $(L_+, L_-) \sqcup (L'_+, L'_-) = (L_+ \cap L'_+, L_- \cup L'_-)$.
- $(L_+, L_-) \sqcap (L'_+, L'_-) = (L_+ \cup L'_+, L_- \cap L'_-)$

The analysis runs bottom-up on the call-graph. After a function $f$ has been analyzed, its effect on locksets is stored as a summary $\mathsf{LockSummary}(f) \in Locks$ that represents the relative lockset at the end of the function. For simplicity of exposition, we assume that functions take only one parameter.

$$\mathsf{lockUpdate} : \mathit{Locks} \times \mathit{Locks} \rightarrow \mathit{Locks}$$

$$\mathsf{lockUpdate}\big((L_+, L_-), (L'_+, L'_-)\big) = \big((L_+ \cup L'_+) - L'_-, (L_- \cup L'_-) - L'_+\big)$$

Figure 3.4: Relative lockset update

$$\mathsf{F_{locks}} : s \times \mathit{Locks} \times \mathit{PPoint} \rightarrow \mathit{Locks}$$

$$\mathsf{F_{locks}}(\mathit{call}(e, a), L, p) = \bigsqcup_{f \in \mathsf{targets(e)}} \mathsf{lockUpdate}(L, \mathsf{Rebind}(\mathsf{LockSummary}(f), f, a, p))$$

$$\mathsf{F_{locks}}(s, L) = L$$

Figure 3.5: Lockset flow function

The flow function for the lockset analysis is shown in Figure 3.5. Because we model lock and unlock operations as function calls, the only statements that modify locksets are function calls $e(a)$. In particular, the `lock(l)` function is modeled as having a relative lockset summary of $(\{\texttt{l}\}, \{\})$ and the `unlock(l)` function is modeled as having a relative lockset summary $(\{\}, \{\texttt{l}\})$. Given a function call $e(a)$, for each possible function $f$ that $e$ may represent (which may be greater than one due to function pointers), the flow function first retrieves the summary $\mathsf{LockSummary}(f)$, and then, using the $\mathsf{Rebind}$ function shown in Figure 3.6, it replaces all occurrences of $f$'s formal in the summary with the actual being passed in. The resulting rebound summary represents the changes in the lockset that occur from the moment $f$ starts executing until it reaches a return. To find the relative lockset after the call to $f$ (relative to the caller's entry point), we apply the changes indicated by the summary to the incoming relative lockset. This is done using the $\mathsf{lockUpdate}$ function shown in Figure 3.4. In particular, the positive differences are added together and so are the negative differences, with the following post-processing: the locks that may have been released in $f$ are removed from the final must-have-acquired lockset, and the locks that must have been acquired in $f$ are removed from the final may-have-been-released lockset.

| $\mathsf{Rebind} : T \times \mathit{Funs} \times \mathit{Exprs} \times \mathit{PPoint} \to T$ |
|---|
| $\mathsf{Rebind}(q, f, e, p) = q[\mathit{formal}(f) \mapsto \mathsf{eval}(e, \mathsf{symStoreAt}(p))]$ |

Figure 3.6: Rebinding formals to actuals.

### 3.3.3 Guarded Access Analysis

Once the lockset analysis from Section 3.3.2 has finished computing the relative locksets for all program points of a given function, the guarded access analysis uses this information to compute the guarded accesses performed by the function.

A guarded access is a triple $\mathsf{A} = (l, L, k)$, where $l \in \mathit{Lvals}$ is the lvalue being accessed, $L \in \mathit{Locks}$ is the relative lockset at the point where the access is made, and $k \in \mathit{AccKind} = \{\mathsf{Read}, \mathsf{Write}\}$ is the kind of access being made (either a read or a write). The set of all guarded accesses is denoted by $\mathit{GuardedAccs} = \mathit{Lvals} \times \mathit{Locks} \times \mathit{AccKind}$.

For each function, our guarded access analysis maintains a guarded access set $\mathsf{A} \subseteq \mathit{GuardedAccs}$ for the entire function. After the lockset analysis has reached a fixed point for a given function, the guarded access analysis starts out by initializing the function's guarded set to the empty set. Then, for each statement $s$ in the function, the access set is updated by calling $\mathsf{UpdateAccessSet}(s, L)$, where $L$ is the relative lockset computed by the lockset analysis at the program point right before $s$. As statements are being processed, the guarded access set increases monotonically; when all statements in the function have been processed, the final guarded access set becomes the access summary of the function. For a function $f$, we denote the access summary of $f$ by $\mathsf{AccessSummary}(f)$.

The most important cases of the UpdateAccessSet function are shown in Figure 3.7. For a function call $e(a)$, UpdateAccessSet copies all guarded accesses from the callee, re-expressing them in the caller's context. In particular, for each possible function $f$ that $e$ may represent, we look up the access summary of $f$, and for each guarded access $(l, L_f, k)$ in the summary, we use Rebind to re-express $l$ and $L$ in terms of the caller's actuals. We also use lockUpdate to plug the rebound

```
UpdateAccessSet : Stmts × Locks × PPoint → void
UpdateAccessSet(x := e, L, p) =
  let σ = symStoreAt(p) in
  A := A ∪ {(eval(e, σ), L, Read)} ∪ {(x, L, Write)}

UpdateAccessSet(call(e, a), L, p) =
  let σ = symStoreAt(p) in
  A := A ∪ {(eval(e, σ), L, Read), (eval(a, σ), L, Read)}
  foreach f in targets(e) do
    foreach (l, L_f, k) in AccessSummary(f) do
      let L' = lockUpdate(L, Rebind(L_f, f, a, p)) in
      let l' = Rebind(l, f, a, p) in
      if isAccessible(l') :
        A := A ∪ {(l', L', k)}
```

Figure 3.7: Guarded access update.

$L$ into the caller's context.

The resulting lvalue $l'$ and lockset $L'$ are added to the guarded access set $A$ only if $l'$ is accessible from globals or from the formals of the function being analyzed. The $isAccessible(l')$ call performs this pruning by running a reachability query in the flow-insensitive points-to graph from the globals and formals to the node representing $l'$.

### 3.3.4 Warning Generation

Once the bottom-up guarded access analysis from Section 3.3.3 has finished running on all functions, the GenerateWarnings function from Figure 3.8 uses the resulting guarded access summaries to generate warnings. The GenerateWarnings function takes as a parameter the set *ThreadEnts* of thread entry points, which are tuples of thread creation sites and functions used to start the thread ($ThreadEnts \subset 2^{PPoint \times Funs \times Funs}$). This includes the "main" thread that starts the program which has a special external creator.

For each pair of thread entry points, GenerateWarnings retrieves the guarded access sets for the two entry points, and then it searches for two guarded accesses

$\mathsf{GenerateWarnings} : 2^{PPoint \times Funs \times Funs} \rightarrow \texttt{void}$

$\mathsf{GenerateWarnings}(ThreadEnts)$
  **foreach** $((p, c, f), (p', c', f'))$ **in** $ThreadEnts^2$ **do**
    **foreach** $\mathsf{A}$ **in** $\mathsf{AccessSummary}(f)$ **do**
      **foreach** $\mathsf{A}'$ **in** $\mathsf{AccessSummary}(f')$ **do**
        **let** $(l, (L_+, L_-), k) = \mathsf{A}$ **in**
        **let** $(l', (L'_+, L'_-), k') = \mathsf{A}'$ **in**
        **if** $mayEqual(l, l') \wedge (L_+ \cap L'_+ = \emptyset) \wedge (k = \mathsf{Write} \vee k' = \mathsf{Write})$ **:**
          $\mathsf{WarnRace}(\mathsf{A}, \mathsf{A}')$

Figure 3.8: Producing data-race warnings

such that the lvalues may be equal, the must-hold locksets do not overlap, and one of the accesses is a write. If two such accesses are found, a warning is generated.

The *mayEqual* function determines if two lvalues could be the same (that is to say, could alias). Nominally, *mayEqual* looks up the representative node of the two lvalues in the flow-insensitive points-to graph, and returns true if the two representative nodes are the same.

At the moment, Relay does not filter out accesses that cannot happen in parallel due to reasons other than locking (*e.g.* happens-before relationships induced by fork-join and condition-variables). In practice, Relay could be augmented with a may-happen-in-parallel analysis (*e.g.* [NAC99]), but perhaps limited to fork-join happens-before relations to avoid the problems of aliasing. Given such an analysis, GenerateWarnings could filter out some pairs of guarded accesses (the inner loops of GenerateWarnings), or even some pairs of thread entry points (the outer loop).

## 3.4 Optimizations

This section presents a few optimizations that, in addition to the modularity afforded by relative locksets and guarded accesses, enable Relay to find races in large programs.

### 3.4.1 SCC-wide Summaries for Accesses to Globals

As presented, each function $f$ has individual guarded access summaries, $\mathsf{AccessSummary}(f)$. We can do better in the case of programs with mutually recursive functions forming non-trivial strongly connected components (SCC) in its call-graph. It is possible for every function in an SCC to *share* a single guarded access summary, for reads and writes to *globals*.

This sharing has several benefits. The most obvious is that it reduces overall memory usage as well as bytes transferred in the parallel grid computing setting. Secondly, it reduces the amount of work done by UpdateAccessSet (Figure 3.7), when handling function pointer calls where some of the target functions are in the same SCC. The outer loop in UpdateAccessSet must be rewritten to take advantage of the SCC-wide global guarded accesses. Finally, it reduces the number of iterations required to reach a fixpoint, since the modifies summaries used by the symbolic execution are actually gleaned from the guarded access writes, and a shared global access summary will present the writes to functions earlier.

Sharing accesses among functions of an SCC does not introduce additional imprecision in the analysis, as long as the input program follows certain patterns.

First, this approach does not introduce accesses that would not already exist in the fixpoint solution. If every function call in the SCC is determined to be reachable, then the set of reads and writes would be exactly the same across the SCC anyway. Because UpdateAccessSet pulls all global accesses from the callee into a caller, without renaming, function calls (say $f$ calls $f'$) induce an inclusion constraint $\mathsf{ProjGlobals}(\mathsf{AccessSummary}(f)) \supseteq \mathsf{ProjGlobals}(\mathsf{AccessSummary}(f'))$. Additionally, for every pair of functions $f$ and $f'$ in the same SCC, there are call-paths from $f$ to $f'$ and $f'$ to $f$. Therefore, by transitivity of the inclusion constraints, along those call-paths, there are inclusion constraints in both directions between $f$ and $f'$. This shows that the sets of accesses themselves are exactly the same.

Another issue is that a guarded access is a triple with a relative lockset, captured from the time of the access, as a component. From this issue two problems arise. First, if the relative lockset involves a lock named by a formal, then the guarded access requires specialization at call-sites. In this case, we simply do not

factor the guarded access out into the shared SCC-wide summary. Fortunately, most global variables are protected by global locks.

The second and bigger problem is that when applying the guarded access summary at function calls, the relative lockset of the guarded access must be composed with the lockset just before the call, using lockUpdate. If guarded accesses are shared within an SCC, then it is possible that a guarded access will be applied at additional call-sites.

This is sound. For calls from functions within the SCC, it may be the case that lockUpdate is applied at a call-site with additional locks held. However, a guarded access with a weaker relative lockset will already exist in the shared summary (the one without those additional locks). For calls from functions outside the SCC, this sharing of guarded accesses changes nothing; the summary is only used after reaching a fixpoint that has these weaker relative locksets.

From the precision angle, precision is retained in two cases. The first case is where the function containing the actual access acquires locks locally. The second case is where a caller outside of the SCC acquires locks prior to the call. *i.e.,* the access cannot rely on locking from within the SCC, since those locks would then be irrelevant according to the same logic used in the discussion about soundness.

### 3.4.2 Optimized Warning Generation

Most programs have only a few thread entries, so the quadratic outer loop in GenerateWarnings is not a bottleneck. However, programs like the Linux kernel may have a non-trivial number of threads. If interrupt-handlers are modeled as separate threads then $|ThreadEnts|^2$ can exceed 100,000. This can be a serious bottleneck.

For some special cases, we could do better with two strategies: cluster guarded accesses, and form a hierarchy of combined thread-entry summaries.

**Clustering Guarded Accesses.** Functions may access the same variables with the same level of synchronization. If one were to join the guarded access summaries of two functions with such an access, the access is represented only once in the combined summary (more details in section 3.5.2).

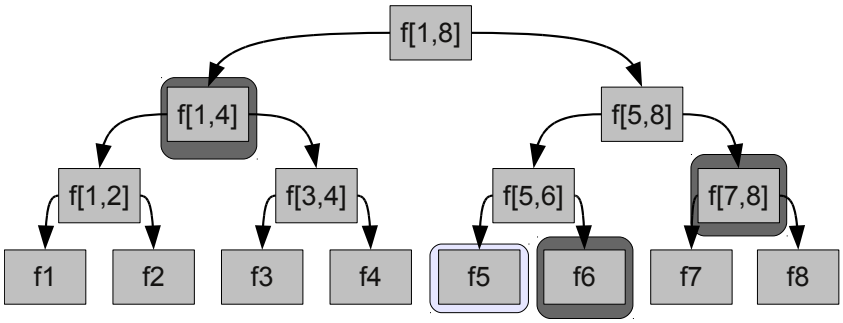



Figure 3.9: Hierarchy of joined thread entry summaries.

**Hierarchy of Combined Thread Entries.** If a function $h$ is known to run in parallel with $f$ and $g$, then the same warnings would be generated if (a) the summary of $h$ is compared with the summaries of $f$ and $g$ individually, or (b) the summary of $h$ is compared with a joint summary for $f$ and $g$. Option (b) may be more efficient because of guarded access clustering.

**Special Case 1: Reflexive All Parallel.** Taking the idea further, if every function in *ThreadEnts* may happen in parallel with each other including itself, then one can construct a single joint summary $F$ covering every function with $O(n)$ joins. Then GenerateWarnings need only compare pairs of guarded accesses in $F$. Thus, it reduces the summary-to-summary comparisons to one, at the cost of joins (involving larger summaries). This is the case with Relay, where all threads are assumed to happen in parallel and threads may be spawned more than once, except main.

**Special Case 2: Non-Reflexive All Parallel.** For the special case where we assume that all thread root functions may run in parallel with each other, but not with itself, we can create a binary tree of joint summaries, like the one in Figure 3.9. This takes $O(n)$ joins. Now, for each function $f_i$ in *ThreadEnts*, GenerateWarnings we need only compare $f_i$ with a joint summary from each level of the tree (except for the root). Thus, this reduces the number of summary-to-summary comparisons to $O(n \log(n))$, at the cost of $O(n)$ joins.

## 3.5 Evaluation

Before applying RELAY to the RADAR, it is useful to have some understanding of RELAY's perfomance as a standalone data-race detector. In actuality, RELAY was developed before RADAR, and the experiments presented here are based on an earlier version of RELAY.

Our experiments involve running RELAY on a large software base, the Linux kernel version 2.6.15. The kernel was configured with the `makeallyes` option and with loadable module support turned off so as to maximize the code included in the build. This choice serves to demonstrate the scalability of our techniques as well as to obtain a close understanding of the variety of idioms used in systems code for synchronizing and avoiding data-races. Under this configuration, the analysis must process 4.5 million lines of code, spanning 46872 functions, scattered across 18042 files.

We begin in Section 3.5.1 with some details about the implementation of RELAY. We then describe the results of running RELAY on the Linux kernel. In particular, RELAY's context- and flow- sensitive analysis resulted in the generation of 5022 warnings (over a 4.5 million line code base). We performed a close analysis of a randomly chosen subset of the warnings and found that most of these warnings were in fact false positives. We categorized the false positives based on the coding idioms used to prevent races, and present a summary of common idioms in Section 3.5.3. Our categorization reveals that to soundly remove the false positives would require sophisticated analyses that are path- and shape-sensitive, and handle non-lock-based synchronization while also scaling to millions of lines, a challenging task that we leave to future work.

Instead, we used our categorization of the sample warnings to devise post-processing *warning filters* capable of automatically placing every warning into one of the categories (Section 3.5.4). After applying the filters we were left with 161 warnings, 31 of which we again carefully categorized. 25 of this subset (80%) were real data-races. Although the filters may remove genuine races, and using it with RADAR would give unsound results, these filters approximate analyses that would be implemented in a more precise race detector, and the fraction of warnings

removed by each filter estimate the relative impact of these more precise analyses.

Finally, we compare RELAY to other data-race detectors built for C programs, in Section 3.5.7.

### 3.5.1 Implementation

RELAY is implemented in OCAML and uses CIL [NMRW02] as a front-end. To build the call-graph, RELAY processes the kernel one file at a time. It traverses each function's body, adding call edges to each function called within the body. For indirect calls, RELAY consults a pointer analysis. For reasons discussed in the introduction of this chapter, we use a per-file pointer analysis. The bottom-up analysis can process a function as soon as summaries of the callees have been computed. Thus, it is possible to analyze multiple functions concurrently, as long as the summaries for their callees have been computed. RELAY exploits this by distributing the summary computations across a grid of 32 nodes each equipped with 2.8Ghz Xeons and 4Gb of RAM. RELAY took 72 hours to perform the whole analysis on a single machine. By distributing the computation, we were able reduce the analysis time to 5 hours.

**Structuring Parallel Computation.** The computation is structured as a single server process that keeps track of the locations of summaries, and worker processes that compute the summaries, request summaries from the server, and report completed summaries to the server. Each SCC of the call-graph is analyzed by a separate worker process. Worker processes begin by requesting an SCC to analyze. Next the worker downloads summaries of the callees to the local file system if they are not found locally. Once summaries of callees are available, the worker computes the new summaries for the SCC functions and then informs a server of the whereabouts of the new summaries. Warning generation is also done in parallel, in a similar manner. A single server process stores warnings generated by the workers and clusters the warnings. Each worker takes a different pair of thread functions, requests their guarded access summaries (if necessary), performs the data-race check, and reports new warnings to the server.

### 3.5.2 Clustering Warnings and Counting Warnings

Before discussing warnings and counting warnings, let us first discuss how they are counted. As RELAY was originally designed as a tool for finding bugs (and not simply as an analysis within RADAR), summarizing or clustering warnings to avoid overwhelming users was a necessary feature. However, clustering can make warning counts deceptively low, or it can complicate comparison if the underlying PTA is changed to one that has a finer set of abstract *Lvals* (a more precise PTA may have *more* clusters of warnings). In this section, we do not compare different versions of RELAY using different PTAs, but we do count the number of clusters of warnings manually inspected and filtered out.

RELAY clusters two warnings in two phases. First, it may cluster the guarded accesses in summaries for each function $g$ if the same memory location is read or written more than once along any path through $g$, with the same relative locksets. To cluster, we union the *PPoint*s that are involved to create a single guarded access. Second, we may cluster two warnings $w_1 = (\mathsf{A}_1, \mathsf{A}'_1)$ and $w_2 = (\mathsf{A}_2, \mathsf{A}'_2)$ based on the program points of the accesses. If the set *PPoint*s of the accesses are the same between $w_1$ and $w_2$ we cluster the two warnings and count them as one warning, even though they may have different thread roots.

Many other strategies exist for clustering warnings. Choi et al. [CLL+02] use a *weaker than* relation to decide when a race involving one access $p$ and $r$ implies that another access $q$ will also race with $r$. This relation can be approximated by checking if (1) the memory locations are the same, and (2) the lockset for $p$ is a subset of the lockset for $q$. In that case, they only track access $q$ to reduce time and space overhead in their dynamic data-race detector. Kahlon et al [KYSG07] use this in the static setting. Chord [NAW06] offers two modes of clustering – clustering by field accessed or abstract lvalue. The field-based view allows users to quickly ignore fields that are intentionally left racy (*e.g.* may only be tracking statistics) and the abstract lvalue-based view is useful for identifying false warnings due to imprecision in the alias analysis.

### 3.5.3 Warning Categorization

RELAY uses the guarded access sets to generate 5022 warnings using the method described in Section 3.3.4. Rather than undertake the herculean task of sifting through all these warnings, we chose to randomly sample and classify 90 of the warnings. This sample contained some races, but the vast majority of the warnings were false positives. However, it turns out that most of the false positives in the sample fell into one of a handful of categories described below. Each of these patterns appears to require a somewhat specialized analysis as they require careful reasoning about path-sensitivity, non-lock-based synchronization, and the shape of the heap, neither of which is easy to scale.

**1. Initialization:** A common idiom is to allocate a structure within a thread, and perform some initialization *without* any synchronization while the structure is still local to the thread, and then to make the structure accessible to other threads, by adding it to a global data structure. Even though subsequent accesses happen while holding a lock, RELAY will report a warning due to the first unprotected access. Figure 3.10 shows a simple code fragment from the kernel that illustrates this pattern. The lower function calls a helper to allocate a structure. The structure `conn` is allocated on line `1` and passed back to the caller. At this point, the structure is not shared and so on line `4` some fields of the structure get initialized, and then on line `5` the structure gets added to a global queue after which it can be accessed by multiple threads. RELAY would warn about subsequent accesses being a race with the access on line `4`.

**2. Unlikely Aliasing:** Many of the warnings reported are false positives because of the flow-, field- and arithmetic- insensitivity of the alias analysis. For example, our alias analysis reports that there is a single "blob" representative node that represents over 10000 variables and heap allocated objects. Race conditions reported on objects within this blob are most likely false positives.

**3. Unsharing:** RELAY reported many warnings on objects that are indeed shared, but which are not shared *during* the time they were accessed. A common situation where this happens is that the object belongs in a shared list, and therefore can be

```
__rxrpc_create_connection(..., **_conn){
1: conn = kmalloc(sizeof(...), ...);
2: timer_init(&conn->timeout, ...);
3: *_conn = conn;
}

rxrpc_create_connection(*trans, ...){
   __rxrpc_create_connection(&conn);
   /* fill in the specific bits */
4: conn->addr.sin_family = AF_INET;
   write_lock(&peer->conn_idlock);
5: list_add(&conn->id_link, _p);
   //...
}
```

Figure 3.10: Unprotected initialization before sharing.

accessed by multiple threads. However, just *before* a thread performs the access, it *removes* the object from the shared list, and then safely accesses the object without any lock. Figure 3.11 illustrates this pattern. `pam` is a reference to the first element of the `page_addr_pool`, and this element is removed from the list in line `1` (after acquiring the appropriate locks for the list). Then, the list lock is released and on line `2` the previously shared object referred to by `pam` is written to without any synchronization.

**4. Re-entrant Locks:** A significant fraction of the false warnings we analyzed were because some data structures were protected with the kernel semaphore, which is a re-entrant lock. For such locks, acquires and releases can be nested, and after $k$ nested acquires, the lock is actually released only after $k$ successive releases. Relay conservatively models these locks, by treating them as released after the very first release call, and thus, finds several unsynchronized shared accesses, even though they are protected by previous acquires.

**5. Non-parallel Threads:** Many false warnings arose due to unsynchronized accesses that take place at instances when the kernel has ensured, using one of

```
set_page_address(..., *virtual){
   spin_lock_irq(&pool_lock);
   pam = list_entry(page_addr_pool);
1: list_del(&pam->list);
   spin_unlock_irq(&pool_lock);
2: pam->virtual = virtual;
   spin_lock_irq(&pas->lock);
3: list_add(&pam->list, &pas->lh);
   spin_unlock_irq(&pas->lock);
}
```

Figure 3.11: Protected unsharing, followed by unprotected access

several mechanisms, that there is only a *single* active thread that can access the shared object. The most common case is when an object is accessed from multiple threads, but the threads use program logic, including signals and other mechanisms, to order operations in such a way that the threads in essence never run in parallel. One such example is shown in Figure 3.12. On line `1` the function `start_sync_thread` checks the shared variable `state` to see if the thread already exists. If not, on line `2` it attempts to create the thread by looping until the thread gets created. After the creation succeeds, the parent thread waits for the child to set the `state` variable on line `4` and then signal completion `6`, at which point, on line `3` the parent returns. This code essentially ensures that only one copy of the `sync_thread` ever runs, and so the access on line `4` is safe, even though Relay will warn that two copies of `sync_thread` may write to `state` at the same time. There are other mechanisms that, like the above, require a very precise thread interleaving analysis of blocking primitives like `wait_for_completion` (illustrated in the example).

**6. Conditional Locking:** Several false warnings generated by Relay were because the program checks some condition to determine whether to acquire locks, and later, checks a correlated condition to determine whether the access should occur. Unfortunately, the acquisition of the lock and the actual access occur in different blocks or functions thereby introducing a path-sensitivity problem. The

```
static sync_thread(*startup){
4: state = IP_VS_STATE_MASTER;
5: set_sync_mesg_maxlen(state);
6: complete(startup);
   //...
}

start_sync_thread(state, ...){
1: if (state == IP_VS_STATE_MASTER)
     return -EEXIST;
repeat:
2: if (kernel_thread(sync_thread,&startup) < 0)
      goto repeat;
3: wait_for_completion(&startup);
   return 0;
}
```

Figure 3.12: Non-lock-based synchronization

example in Figure 3.13 exhibits this pattern. The upper function either returns `NULL` without holding the lock if the condition on line `1` holds, or acquires the lock on line `2` and returns a non-null value. This return value is checked on line `4` before performing the access on line `5`.

### 3.5.4 Filters

We have devised simple syntactic filters based on the above categorization to automatically categorize the warnings thereby yielding a subset of the warnings that are very likely genuine races. The design of these filters was guided by finding common patterns among the warnings in a given category.

These filters are very aggressive, and they are unsound, in the sense that they can remove real races too. However, since this source of unsoundness is confined to a post-processing pass, it can easily be removed or replaced with sound analyses.

We now describe the filters. In each case, in parentheses we list the categories that the filter targets.

```
static * swap_info_get(entry){
1: if (!entry.val)
      goto 3;
   p = &swap_info[type];
2: spin_lock(&swap_lock);
   return p;
3: return NULL;
}

swap_free(entry){
   p = swap_info_get(entry);
4: if (p) {
5:    swap_entry_free(p, ...);
      spin_unlock(&swap_lock);
   }
}
```

Figure 3.13: Conditional locking

**1. Thread-local Allocation (Initialization)**: To handle the initialization false-positives, we filter out warnings on objects that are allocated inside the thread within which the conflicting access occurs.

**2. Large Points-To Reps. (Unlikely Aliasing, Unsharing)**: For unlikely aliasing we can filter warnings where the flow-insensitive alias analysis is asked to compare lvalues whose representative nodes represent more than $k$ lvalues for a parameter $k$ ($k = 1$ for our results). Typically, these are nodes where different data-structures are mixed at a common function (*e.g.* , two different lists merging at a node removal function). As this filter captures warnings involving data-structures, it also applies to the "unsharing" pattern.

**3. Bootup Thread (Re-entrant Locks)**: The most heavily used re-entrant lock is `kernel_sem`. This lock is mainly used by the boot-up thread which holds it for most of its execution. Thus, to filter warnings about re-entrant locks, it sufficed to filter warnings where one of the accesses was in the boot-up thread.

**4. Same Root (Non-parallel Threads)**: We noticed that many false positives involving threads that cannot execute concurrently were warnings where the two

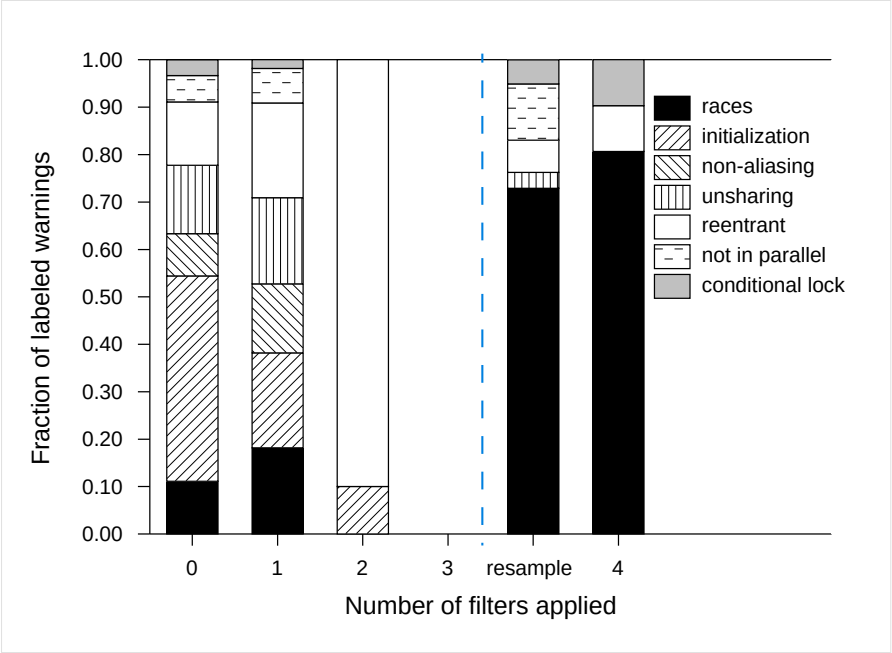



Figure 3.14: Relative proportions of categories in the manually labeled warnings after each filter is applied.

accesses originated from the same thread. We therefore designed a filter that removes such warnings.

### 3.5.5 Other Filters Considered

The above filters were sufficient for reducing the set of warnings generated for Linux to a manageable size. Although measurements are not shown, we did consider the following additional filters:

**No Locks (Non-parallel Threads or Thread-local Data)**: This filter removes warnings on pairs of accesses where *no locks* are held at *either* access. In other words, this filter expresses the heuristic that if the programmer believes an object is not shared with other active threads they will not protect accesses to the object. This filter is also likely to be effective for removing benign races.

**Changing** lockUpdate **or Shadowing Locksets (Re-entrant Locks)**: To handle re-entrant locks we can change the analysis slightly, and change the lockUpdate

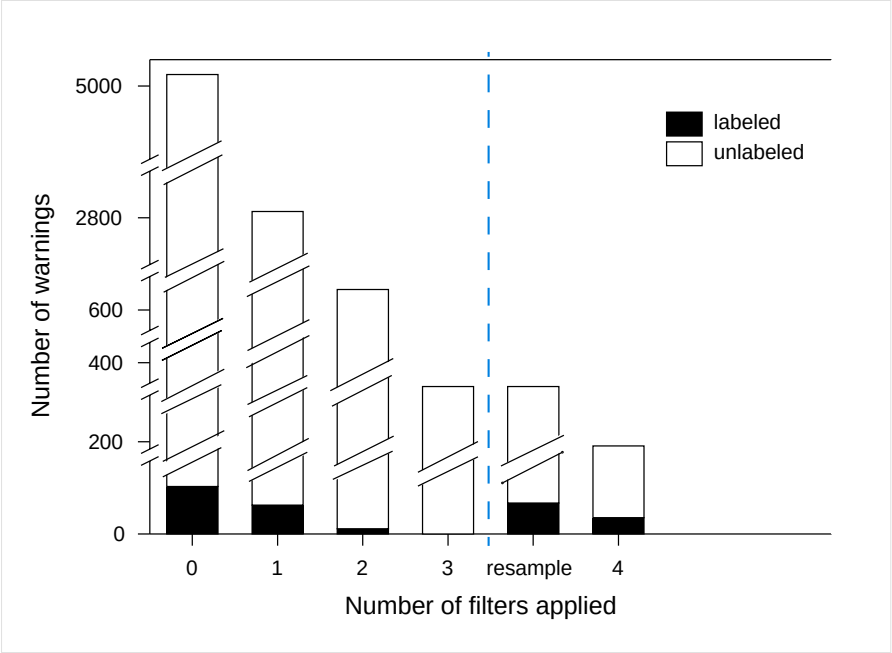



Figure 3.15: Absolute number of manually labeled vs. unlabeled warnings after each filter is applied.

$$\mathsf{lockUpdate}' : \mathit{Locks} \times \mathit{Locks} \rightarrow \mathit{Locks}$$

$$\mathsf{lockUpdate}'\big((L_+, L_-), (L'_+, L'_-)\big) = \\ ((L_+ \cup L'_+) - L'_-, \big(L_- \cup (L'_- - L_+)\big) - L'_+)$$

Figure 3.16: Alternative relative lockset update (for re-entrant locks)

function to leave no effects (Figure 3.16) in the case that a re-entrant lock is released within the same function in which it is acquired, and the release is after the acquire. Rather than leave the lock in the $L_-$ set after the release (which occurs after the acquire), we can omit the lock from both sets. The release balances the acquire, and, to the caller, this has no effect. An alternative to actually changing lockUpdate to lockUpdate′ is to track an additional shadow relative lockset which is updated using lockUpdate′. Warnings can be generated with the original locksets, or warnings can be filtered by checking for common locks in this shadow lockset.

### 3.5.6 Results

We evaluate the result of applying the filters using two criteria. First, they should remove false positives, *i.e.,* after applying the filters, the *fraction* of real races left in the warning should increase (Figure 3.14). Second, they should not remove too many races, *i.e.,* after applying the filters we should still be left with a pool of warnings large enough to contain many real races (Figure 3.15).

We applied the filters to the warnings as follows. First, we drew a random sample of 90 warnings from the 5022 warnings generated by RELAY. We manually placed each of the warnings into one of the six categories described in Section 3.5.3. When a warning fell in multiple categories, as was often the case, we placed the warning into the first (according to the order shown) category.

Next, we applied the filters in the order described in Section 3.5.4. Figure 3.15 shows how after each filter is applied, the total number of warnings as well as the number of warnings in the manually categorized sample set decreases. Figure 3.14 shows how the distribution of the categories changes in the remaining set of samples, as we apply more filters. The important thing to note here is that the dark solid bar, which represents the percentage of real races in the sample set increases monotonically as we apply filters, and reaches 80 % after having applied 4 filters.

We now describe the four steps in applying each one of the four filters:

1. After applying the first filter (which is meant to remove initialization false positives), the total number of warnings drops to 2812 and the manually categorized sample set drops to 55 (bar 1). Moreover, the fraction of remaining sampled warnings that are initialization false positives drops from about 43% to 20%, indicating that the filter did in fact remove a larger proportion of initialization false positives than other warnings.

2. After applying the representative node filter (which was meant to remove false positives due to unlikely aliasing or unsharing), the total number of warnings drops to 639 and the sample set drops to 10 (bar 2). Of the remaining samples, there are no more unlikely aliasing or unsharing false positives, indicating that

the filter was a good heuristic for removing these false positives. This filter also had the unintended effect of removing all the non-parallel-threads false positives. Unfortunately, it also removed all the races that we had identified in our first sample set.

3. After applying the third filter, the manually categorized sample set went down to zero (bar 3), and so we *re-sampled* the set of 355 remaining warnings to obtain a new sample set of 59 warnings which we again manually categorized. After resampling, we applied the third filter (bar labeled "resample"), namely the bootup-thread filter, which was meant to remove the re-entrant locks false positives. At this point, the percentage of false positives categorized as re-entrant locks decreases significantly, indicating that the filter is effective at removing these false positives.

4. After applying the same-entry filter (which is meant to remove non-parallel threads false positives), all the non-parallel threads false positives have been removed (bar 4). At this point, the number of remaining warnings is 161, and the size of the manually categorized sample set is 31, of which 25 (80 %) are real races. Note that we have not been able to devise a filter targeted at conditional locks, and therefore the majority of remaining false positives fall in this category.

We conclude from the above that the filters effectively refine the set of warnings and increase the fraction of races from 11% to 80 %, without eliminating an unacceptably large number of races. Counted another way, we manually analyzed 149 warnings in all, and found 53 races.

**Races.** After the application of the filters, the vast majority of warnings are real races. Some races found are possibly benign. For example, some races found involve a flag variable (*e.g.* `thread_finished`) that is written once in a thread, and read in a loop by the other thread, without locks. Figure 3.17 shows a more serious race that survives all filters. By the time we obtained our results, this race had already been reported and fixed.

The race involves the read on line `2` of `p->size` and the write, on line `5` of `t->size`, since `t` and `p` can point to the same object and there are no common

```
iounmap(volatile *addr){
   read_lock(&vmlist_lock);
   for (p = vmlist; p; p = p->next) {
      if (p->addr == addr) break;
   }
1: read_unlock(&vmlist_lock);
   change_page_attr(virt_to_p(p->phys_addr),
2:                  p->size >> PAGE_SHIFT);
}

/* called with write_lock(vmlist_lock) */
__remove_vm_area(*addr){
3: for (t = vmlist; t != NULL; t = t->next) {
       if (t->addr == addr) break;
   }
4: unmap_vm_area(t);
5: t->size -= PAGE_SIZE;
   return t;
}
```

Figure 3.17: A real race found after applying filters.

locks held. This race is serious because the function `change_page_attr` uses the `p->size` parameter that is passed in as the bound for a loop iterating over an array. Due to the race the read of `p->size` can return a stale bound causing the loop inside `change_page_attr` to access the array out of bounds.

### 3.5.7 Comparison to Other Race Detectors

In this Section, we evaluate Relay with a comparison against two other data-race detectors built for C programs, Locksmith and LP-Race.

#### Resource Requirements

Relay scales to handle much larger programs than Locksmith and LP-Race, and analyzes the same programs with much fewer resources. Figure 3.18 charts out a *rough* comparison between the three data-race detectors. Since an implementation of LP-Race is not currently available, times are taken from the

| Benchmark | KLOC | Locksmith | LP-Race | Relay |
|---|---|---|---|---|
| **aget** | 2.2 | 0.85 | 4 | 0.2 |
| **ctrace** | 2.2 | 0.59 | 5 | 0.2 |
| **smtprc** | 8.6 | 5.37 | 145 | 0.8 |
| **plip** | 19.1 | 19.14 | ? | 0.6 |
| **hp100** | 20.4 | 143.23 | ? | 0.8 |
| **synclink** | 24.7 | 1521.07 | ? | 1.5 |
| **retawq** | 40.1 | OOM | 6855 | 18.4 |

Figure 3.18: Comparison of running times for Locksmith, LP-Race, and Relay

paper [Ter08]. Running times for Locksmith are taken from Polyvios's dissertation [Pra08]. Relay's numbers are based on running the analysis on a single 3.2 Ghz Pentium 4 with 2GB of RAM. One issue to consider when examining these running times is that there are slight variations in hardware setup. Finally, variations in running times can be attributed to implementation details or engineering effort. Nevertheless, the data is presented here for reference.

The benchmarks in the table are a sample of drivers and applications from the Locksmith test-suite, plus the Retawq application tested by LP-Race. Benchmarks that were not tested in the LP-Race paper are marked "?". On one benchmark, Locksmith runs out of memory (denoted "OOM").

### Precision

We also compare the precision of each tool qualitatively. There are many aspects that can contribute to differences in precision between each tool. For example, each tool can be parameterized by different alias analyses for checking may-aliasing between memory locations that are accessed in each thread.

**Differences in Formulation and Additional Analyses.** One important high-level difference is that LP-Race is not based on the lockset algorithm. LP-Race's capability-based formulation naturally handles non-lock-based synchronization: signaling, semaphores, read-write locks, and fork-join. Locksmith introduces an additional analysis – on top of correlating accesses and locks – based on *contextual effects* to prune accesses that occur before any threads are spawned. Essentially,

```
int x = 0, y = 0;
void write(p) { *p = 1; }
void read(p) { printf("%d\n", *p); }
void call_fp(fp, x) { *fp(x); }

void thread_1() {
L1: call_fp(&read, &x);
}

int main() {
    spawn (&thread_1);
L2: call_fp(&read, &x);
L3: call_fp(&write, &y);
}
```

Figure 3.19: Limited context-sensitivity of RELAY's guarded access summaries.

this is a way to handle fork but not join synchronization. Finally, Locksmith introduces another additional intra-procedural *uniqueness analysis* to handle some forms of the thread-local initialization idiom outlined in Section 3.5.3. It is possible to incorporate a similar analysis into RELAY. However, because the analysis is intra-procedural, it would not actually handle the example in Figure 3.10, which we found in the Linux kernel. The ability to handle these forms of synchronization and program idioms give LP-Race and Locksmith an edge over RELAY in the area of precision.

**Imprecision of Relative Locksets.** Since one contribution of RELAY is the concept of *relative locksets*, it is important to understand when this abstraction is imprecise. This limited form of context-sensitivity only generates a single summary for each function, which RELAY can specialize at function invocations. Specialization allows two things: (1) locks that are not affected by the function call can be specialized, and (2) value flow that can be handled by renaming. Given this form of summary, RELAY handles the value-flow of the ever-popular `id` function context-sensitively, as well as lock/unlock-wrappers.

An important limitation of this form of context-sensitivity is that RELAY does not handle control-flow context-sensitively. One special case is function-

pointers. Consider the example in figure 3.19. The `call_fp` function is called with either `read` and &`x` at lines `L1` and `L2`, or it is called with `write` and &`y` at `L3`. Since the only common variable between the main thread and `thread_1` is `x`, and `x` is only read, there should be no race. However, only one summary is generated for the function `call_fp`, regardless of the function pointer that it is given. Thus, the summary will be generated assuming that `fp` is both `read` and `write`. This will mean that RELAY will see both a read and write to `x` in the main thread and in `thread_1`, generating a false data-race warning. One workaround for this limitation is to generate an context-sensitive call-graph and have run RELAY on this split call-graph. Then, there will be one version of `call_fp` that exclusively reads and another that exclusively writes, avoiding the false race. One way to generate a context-sensitive call-graph is to use the Wilson-Lam algorithm [WL95] described in Section 4.2. In comparison, Locksmith's context-sensitivity handles function-pointers more directly.

## 3.6 Summary

In this chapter we presented a static race detection analysis that scales to millions of lines of C code. At the heart of our technique is the notion of a relative lockset which allows functions to be summarized independent of the calling context. This, in turn, allows us to perform a modular, bottom-up analysis that is easy to parallelize. We have analyzed 4.5 million lines of C code in 5 hours, and after applying some simple filters, found a total of 53 races.

One of our long-term goals is to soundly eliminate false positives to the point where a large fraction of the remaining warnings, say more than 70%, correspond to real races. To this end, we would like to replace the simple but unsound filters with sound analyses targeted at the coding patterns that we have found to be the leading causes of false positives. Examples of such analyses include a thread-escape analysis for the initialization pattern and a light-weight shape analysis for the unsharing pattern.

Another long-term goal is to address the problem of determining "serious"

races. Some of the races are clearly benign, as deduced from syntactic cues such as variable names like `oops_in_progress`, while others appear to be dangerous. The dangerous races are often those that cause higher-level semantic bugs, such as atomicity violations or unsafe memory accesses like the one shown in Figure 3.17. Combining RELAY and RADAR with sequential dataflow analyses like those for finding null-pointer violations is a start. Its effectiveness, however, will be dependent on the precision of every component, including RELAY.

**Acknowledgments:** Chapter 3 in part, has been published as "Relay: Static Race Detection on Millions of Lines of Code" by Jan Voung, Ranjit Jhala, and Sorin Lerner in *ESEC/FSE 07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering* [VJL07]. The dissertation author was the primary investigator and author of this paper.

# Chapter 4

# Evaluation of Call-graph Construction Algorithms

This chapter considers the role of call-graph construction in the precision and scalability of inter-procedural analyses, including RELAY and input dataflow analyses to the RADAR framework. This is done by comparing several scalable call-graph construction algorithms in conjunction with two inter-procedural analyses that serve as clients of the constructed call-graphs. The collection of call-graph construction algorithms tested here span the most common analysis design choices, allowing us to evaluate the importance of each design choice.

## 4.1 Motivation

Recent years have seen great progress on scalable, flow-sensitive techniques for analyzing memory, security and concurrency properties of low-level C systems code [DLS02, HDWY06a, XA05, VJL07, CDOY09]. This progress is founded on the insight that scalability requires *modular analyses*. Each of the above works by computing *summaries* that succinctly capture the effect of each procedure on its callers. Callers then plug in the results at each callsite to compute summaries for its own callers and so on. RELAY (Chapter 3), and the clients we have developed to test RADAR (Chapter 5) are all modular analyses.

A modular analysis critically depends on a method for precisely determining

which functions can be called at a given callsite. If this determination is imprecise, *i.e.,* if the method greatly exaggerates the number of possible (transitive) callees, then the results computed by modular analysis will be polluted by the spurious callees and hence, will be imprecise. Furthermore, the effects of the spurious callees will propagate to the callsite, thereby causing a blow-up in the information tracked at the caller, which limits the scalability of the modular analysis.

The problem of *call-graph construction* for C, that is, determining the callees at each callsite within a C program, is challenging, as C is decidedly not a first-order programming language. Systems programmers aggressively employ high-level design patterns such as iterators, accumulators, visitors, objects, and callbacks in order to structure large low-level code bases and facilitate code-reuse. In a low-level language like C, which lacks high-level type structure (*e.g.* classes) all these patterns are encoded using function pointers. Thus to precisely analyze large systems code bases, where function pointers are used most aggressively, we need a precise function pointer analysis.

In this chapter, we experimentally evaluate a wide spectrum of call-graph construction algorithms on a number of large C benchmarks with the goal of determining how the precision of the resulting call-graph affects the final results computed by the (modular) client analyses. In particular, we are interested in measuring how different analysis choices, such as unification vs. inclusion, flow-insensitive vs. sensitive, field-insensitive vs. sensitive, context-insensitive vs. sensitive, affect the quality of the computed call-graph, the quality of the results produced by a client analysis, and the scalability of the client analysis.

While such studies have been carried out in the past [MRR04, DLFR01, CDC+04, PKH04a], they have been limited in several ways. First, they study much smaller benchmarks, which typically use function pointers less aggressively. Second, they only study the simpler (and older) value-flow analyses that existed at the time of the study. Third, and most importantly, they do not measure the impact of the call-graph quality on client analyses.

We present an evaluation of call-graphs constructed using the following algorithms:

- Steens-FI [Ste96a] a unification-based, flow-insensitive, field-insensitive, and context-insensitive analysis.
- DSA [LLA07] a unification-based, SSA-flow-insensitive, field-sensitive, and context-sensitive analysis.
- Anders-FI [And94] an inclusion-based, flow-insensitive, context-insensitive, and field-insensitive analysis.
- Anders-FS [HL07a] an inclusion-based, SSA-flow-insensitive, field-sensitive, and context-insensitive analysis.
- WL [WL95] a flow-sensitive, field-sensitive, context-sensitive analysis that is a version of the Wilson-Lam algorithm restricted to function values.

We measure the precision of the analysis using two metrics: the fan-out (number of possible callees at each indirect call-site), and the sizes of SCCs in the call-graph. The first metric is an indication of the imprecision in the client caused by "smearing" induced by spurious callees. Modular inter-procedural analyses compute fixpoints over one SCC at a time, and hence, the second metric is a sign of the scalability of the client analysis.

However, rather than rely solely on the proxy metrics, we directly study the impact of call-graph precision on two clients that use the generated call-graphs to perform an inter-procedural analysis. One client performs a non-null analysis, and the other client is the Relay data-race detector. We measure the time and memory required by these two analyses (a more precise call-graph will make the client analysis more efficient), and the precision of the resulting analysis (a more precise call-graph will be able to prove more dereferences null-safe or race free).

We have carried out our evaluation for a large set of C benchmarks ranging from 12 KLOC to 284 KLOC (Section 4.3). These benchmarks make aggressive use of function pointers. We show both qualitatively (via examples gleaned from the benchmarks) and quantitatively (via metrics of the computed call-graphs and the client analyses) how the different dimensions of sensitivity affect the quality of the resulting call-graph, or the quality of the client analysis.

Our experimental results allow us to make the following conclusions:

- Field-sensitivity and inclusion constraints (vs. unification constraints) matter most for constructing precise call-graphs for C programs. Context-sensitivity, on the other hand, does not provide a substantial amount of additional precision.

- For the client analyses we considered, more precise call-graphs significantly improve the running time and memory usage of the analysis, but they do not significantly improve the precision of the end results. This points to the fact that the primary purpose for getting a more precise call-graph may very well be for scalability of the client rather than precision of the end results.

- SCCs get very large even when ignoring function pointers, meaning that client analyses that work on one SCC at a time may need to decompose the problem further to improve scalability.

## 4.2 Overview of Pointer Analyses

The main goal of our study is to evaluate the effectiveness of various pointer analyses for the purpose of call-graph construction in C. To begin our study, we must therefore pick several well-known inter-procedural pointer analyses to compare. Our criteria for selecting these analyses are two-fold. First, we want our analyses to scale to relatively large benchmarks (on the order of hundreds of thousands of lines of C). Second, we want our analyses to span the spectrum of time/precision tradeoff – from analyses that are quick and imprecise, to analyses that are expensive (yet still runnable) and precise.

To this end, we have selected the following pointer analyses to compare: Steensgaard [Ste96a], field-insensitive Andersen [And94], field-sensitive Andersen, Data-structure analysis (DSA) [LLA07], and an algorithm similar to Wilson and Lam (WL) [WL95]. In addition to these algorithms, we have also selected several optimistic lower bounds to compare against, for example an analysis that builds a call-graph ignoring function pointers altogether.

### 4.2.1 Dimensions of Difference

Before we describe each of the algorithms in more detail, we review the common properties that differentiate these algorithms.

- **Flow-sensitive vs. Flow-insensitive**: A flow-sensitive analysis computes points-to information at each program point, whereas a flow-insensitive analysis computes a single points-to solution for the entire program. Flow sensitivity provides additional precision, because the order of statements is taken into account, but it also uses more memory, and can take longer to fix-point. SSA is used in some of our flow-insensitive algorithms to regain some flow-sensitivity.

- **Unification vs. Inclusion**: These define how assignment and parameter passing are treated in flow-insensitive algorithms. Unification treats an assignment `x := y` as equating $Pts(x)$ and $Pts(y)$ (with each member of $Pts(x)$ and $Pts(y)$ also equated). Solving such equality constraints does not require iteration, and can be done in near linear time using a union-find data structure. Inclusion, on the other hand, treats assignment as a constraint $Pts(x) \supseteq Pts(y)$. Such constraints are more precise because they keep track of the direction of the assignment (and thus the flow of pointer values), but they are also more expensive to solve.

- **Field-sensitive vs. Field-insensitive**: A field-sensitive analysis distinguishes between different fields of the same memory object, whereas a field-insensitive analysis merges all fields of an object together (so that for example `&x.f` becomes the same abstract memory location as `&x.g`). Field-sensitivity adds more precision, at the expense of larger dataflow facts.

- **Context-sensitive vs. Context-insensitive**: A context-sensitive analysis distinguishes between different calling contexts (at differing levels of abstraction, depending on what the definition of a context is), whereas a context-insensitive analysis merges all calling contexts together.

- **Heap-cloning vs. Non-Heap-cloning**: heap cloning affects how calls to `malloc` are analyzed in a context-sensitive analysis. In general, pointer analyses represent the memory returned by a call to `malloc` using an abstract memory location. Heap cloning refers to the technique of generating *different* `malloc` abstract memory locations for *different* calling contexts. On the other hand, without heap cloning, each `malloc` is represented with one abstract memory location (as opposed to one per calling context).

### 4.2.2 List of Algorithms

We now describe each one of the algorithms we compared in more detail.

**Steens-FI:** This is the standard Steensgaard unification-based algorithm [Ste96a]. It requires no iteration leading to an almost-linear time complexity. For our experiments we used a field-insensitive version.

**DSA:** This is the Data-structure Analysis (DSA) algorithm by Lattner, Lenharth and Adve [LLA07]. Like Steensgaard, it is a unification-based algorithm. Unlike Steensgaard it is field-sensitive, context-sensitive and performs heap cloning.

**Anders-FI:** This is the standard Andersen inclusion-based algorithm [And94]. It requires iteration and has worst-case $O(n^3)$ time complexity. Recent work on optimizing the execution of Andersen has made Andersen practical for large programs. These optimizations fall into two categories. First, *representation optimizations* are aimed at reducing the memory cost of storing the points-to-graph using various encoding techniques such as BDDs [HL07a, WL04, ZC04, BLQ+03]. Second, *unification* optimizations aim to reduce the size of the problem by unifying memory locations, but only in cases where the unification does not affect precision. For example, cyclical constraints $Pts(p1) \subseteq Pts(p2) \subseteq Pts(p1)$ mean that the sets $Pts(p1)$ and $Pts(p2)$ are equivalent (even though `&p1` and `&p2` may not be equivalent). Such cycles can be detected either during constraint resolution [FFSA98, HT01, PKH04b, HL07a], or a priori [RC00, HL07b, Sim09]. Much of the work is focused on how to balance the cost of cycle detection and the completeness of search, e.g., using heuristics based on the likelihood of finding cycles.

| | Flow-Sens | Inclusion | Field-Sens | Context-Sens | Heap-Clone |
|---|---|---|---|---|---|
| STEENS-FI | | | | | |
| DSA | SSA | | ✓ | ✓ | ✓ |
| ANDERS-FI | | ✓ | | | |
| ANDERS-FS | SSA | ✓ | ✓ | | |
| WL | ✓ | ✓ | ✓ | ✓ | ✓ |

Figure 4.1: Summary of analyses. SSA indicates that it is flow-insensitive over an SSA representation, buying some level of flow-sensitivity.

**Anders-FS:** We also evaluated a field-sensitive version of Andersen. There are no additional optimizations related to field-sensitivity.

**WL:** This is a context-, field-, and flow-sensitive algorithm that performs heap-cloning. It uses a context-sensitivity abstraction similar to that of Wilson and Lam (WL) [WL95] and Relevant Context Inference [CRL99]. In particular, the context for a call-site is the dataflow information at the call-site, but abstracted away to the pointers/fields that are actually dereferenced in the callee. Thus, contexts that differ on inessential details (*i.e.,* : differ on pointers/fields not used in the callee) are merged together. There is a common belief that the Wilson and Lam context-sensitivity approach does not scale to large C programs, a belief that our experiments confirm. However, if we limit Wilson and Lam to only track pointers that transitively reach function pointers (rather than all pointers), then the analysis does in fact scale to large realistic programs (shown in Appendix B).

The WL context abstraction is useful for constructing context-sensitive call-graphs for clients, as it keeps the number of contexts low (the majority of functions in real C programs end up having a single context). In contrast, context-sensitive approaches like [WL04, ZC04] generate an exponential number of contexts, and they use BDDs to compactly represent redundant information across the many contexts. This requires the client analysis to also be formulated in terms of BDD operations to scale.

A summary of the above pointer analysis algorithms and their properties is show in Figure 4.1. Furthermore, we also consider the following two lower bounds on the precision of call-graphs.

**NoFP:** The NoFP call-graph treats every indirect call as having no targets. The call-graph constructed in this way is clearly unsound (in that it will miss real call edges), but it provides a lower bound showing how much more room there is for improvement in the treatment of indirect calls.

**Dynamic:** The Dynamic call-graph uses calls that are recorded from runs of the program to determine targets of indirect calls. From this information, we generate a context-insensitive call-graph (although it would also be possible to generate a context-sensitive call-graph using a callstring context-abstraction or using other context-abstractions by recording more runtime information). The empirical lower bound provided by the Dynamic call-graph is tighter than the lower bound provided by NoFP. It serves as a great sanity check to make sure that none of our algorithms are missing any real calls.

## 4.3 Results for Call-graph Precision

The goal of our study is to determine how the various pointer analyses described in Section 4.2 affect the precision of the constructed call-graph, and the precision/running-time of client analyses that make use of the call-graph. We start in this section by evaluating the precision of the resulting call-graphs, and then in Section 4.4 we evaluate the effect on client analyses.

### 4.3.1 Experimental setup

**Implementations.** We implemented Steens-FI, Anders-FI, and WL, using CIL 1.3.7 and OCaml. For DSA and Anders-FS, we reused the implementations of these pointer analyses by the authors themselves in LLVM, adding only a call-graph generation pass. In particular, we used the DSA implementation (based on the work in [LLA07]) found in version 2.6 of LLVM, and we used the scalable implementation of Anders-FS (based on the work in [HL07a, HL07b]) found on one of the author's web page: `http://www.cs.ucsb.edu/~benh/downloads.html`.

**Benchmarks.** The C programs that we used as benchmarks are shown in Fig-

| Benchmark | KLOC | #ind | #all | %ind | WL avg ctxts |
|---|---|---|---|---|---|
| **Bzip2 1.0.5** | 12 | 20 | 1185 | 1.7 | 1.06 |
| **Icecast 2.3.2** | 28 | 199 | 4848 | 4.1 | 1.76 |
| **Unzip 6.0** | 31 | 381 | 2271 | 16.8 | 1.01 |
| **OpenSSH 5.2** | 56 | 141 | 11910 | 1.2 | 1.06 |
| **smtpd 2.6.2** | 67 | 337 | 10150 | 3.3 | 1.72 |
| **Mutt 1.5.9** | 83 | 743 | 16833 | 4.4 | 1.08 |
| **Git 1.3.3** | 171 | 91 | 27612 | 0.3 | 1.26 |
| **Vim 7.2** | 230 | 27 | 29305 | 0.1 | 1.06 |
| **Emacs 22.3** | 272 | 288 | 34491 | 0.8 | 1.06 |
| **Nethack 3.4.3** | 284 | 863 | 38432 | 2.2 | 1.02 |

Figure 4.2: Benchmark characteristics including number of indirect calls vs. all calls and average number of contexts for WL

ure 4.2. The first column gives the name and version of the benchmark, the second column the lines of code (in KLOC), and the third column the number of indirect callsites, which are callsites that use function pointers to determine the callee. We don't provide exact numbers for the running times and memory usage of the various analyses because it's not clear that the implementations are directly comparable (some are implemented in OCaml, some in C++). However, the general trends are as follows. Not surprisingly, Steens-FI is the fastest of the analyses, finishing in under 2 minutes for each benchmark. DSA and Andersen's are slightly slower, finishing in under 3 minutes. The slowest analysis by far is WL, finishing in under 100 minutes. In terms of memory usage, all analyses finish within 1 GB of memory, except for DSA on 5 benchmarks, where it did not finish within 3 GB.

### 4.3.2 Call-graph Metrics and Results

**Metrics.** To evaluate precision, we considered several metrics. The first is fan-out, which is the number of targets per indirect call. Fan-out is important because it measures how much unnecessary smearing can occur in a client inter-procedural analysis that uses the constructed call-graph. The second metric is the size of the SCC that each function belongs to. Our previous experience in building bottom-up summary-based inter-procedural analyses [VJL07, CVJL08] is that the maximum

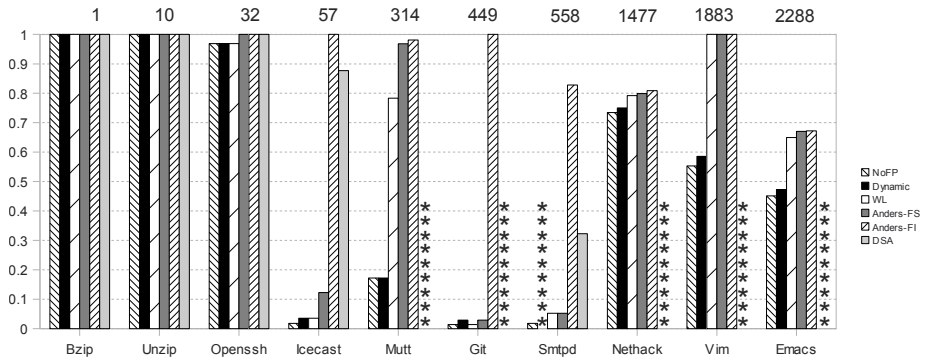



Figure 4.3: Max SCC size (normalized to Steensgaard)

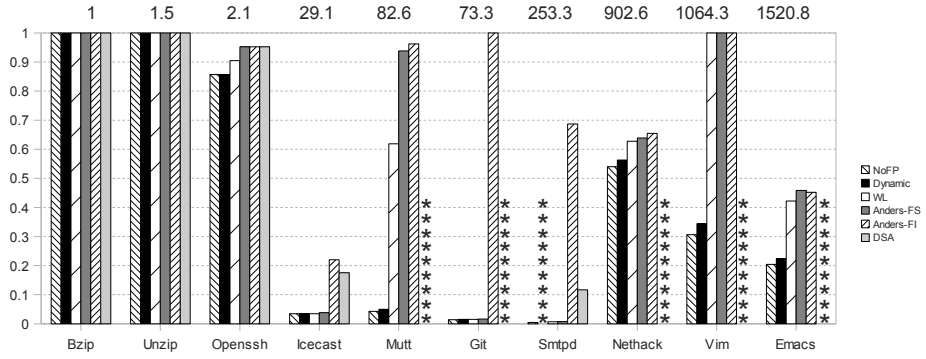



Figure 4.4: Avg. SCC size (normalized to Steensgaard)

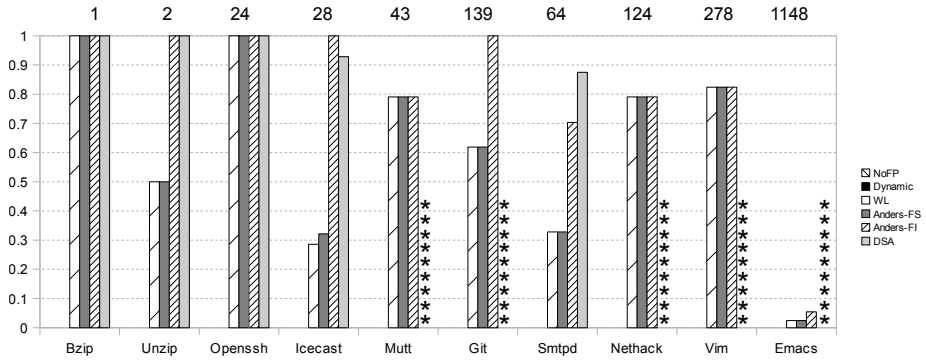



Figure 4.5: Max Fan-out (normalized to Steensgaard)

SCC size is a big determining factor for scalability. This is because many bottom-up analyses compute the result for all functions in a given SCC at once. The

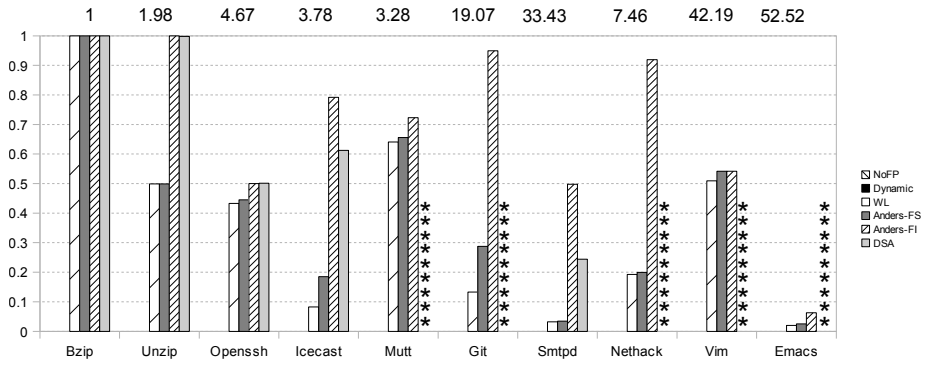



Figure 4.6: Avg. Fan-out (normalized to Steensgaard)

largest SCC is therefore the hardest to analyze, and all other SCCs typically pale in comparison. We measured each of these two metrics (fan-out and SCC size), both as a maximum and as an average. To make the context-sensitive call-graph of WL comparable to the context-insensitive call-graphs, we take the max value (SCC or fan-out) out of all contexts for each particular function.

**Organization of Results.** The results are shown in Figures 4.3, 4.4, 4.5, and 4.6. The benchmarks are arranged left-to-right in increasing size of maximum SCC produced by Steens-FI. For each benchmark, the bars represent different algorithms for constructing the call-graph, all normalized to the results that Steens-FI would produce. We use a bar made out of stars ("$*$") for DSA to indicate that it ran out of memory (the actual size of the "$*$" bars does not mean anything). Because the numbers are normalized to the Steensgaard results for that benchmark (where Steensgaard is therefore 1), it is hard to see the scale of the actual numbers. To give a better feel for the absolute values, we display on top of the bars the Steensgaard result for each benchmark, thus giving an indication of what the "1" represents for that benchmark.

We used the Dynamic measurements as a sanity check to gain confidence that our algorithms would not miss real calls. We only gathered Dynamic measurements for SCC size (not fan-out). Also, we were unable to collect Dynamic measurements for Smtpd because we were unable to run the instrumented version of Smtpd.

| Benchmark | KLOC | #ind | ind Coverage |
|---|---|---|---|
| **Bzip2 1.0.5** | 12 | 20 | 1.0 |
| **Icecast 2.3.2** | 28 | 199 | 0.04 |
| **Unzip 6.0** | 31 | 381 | 0.08 |
| **OpenSSH 5.2** | 56 | 141 | 0.14 |
| **smtpd 2.6.2** | 67 | 337 | * |
| **Mutt 1.5.9** | 83 | 743 | 0.02 |
| **Git 1.3.3** | 171 | 91 | 0.22 |
| **Vim 7.2** | 230 | 27 | 0.16 |
| **Emacs 22.3** | 272 | 288 | 0.08 |
| **Nethack 3.4.3** | 284 | 863 | 0.17 |

Figure 4.7: Indirect call coverage of the dynamic test suite

To estimate the tightness of the DYNAMIC lower bound we also measure the coverage of the tests used to generate these results. The coverage metric is the following. Let $I(b)$ be the number of functions with indirect calls for a benchmark $b$. Let $C(b)$ be the number of functions with indirect calls for benchmark $b$ that are visited in the test suite. Coverage is defined as $C(b)/I(b)$ (a value between 0 and 1). Note that with a value $C(b)/I(b) = 1$ it is still possible that the dynamic lower bound is not tight. This coverage metric only measures the number of indirect call sites, not all of the different possible values that can flow into the function pointer at each of these call sites (because that is unknown!). Figure 4.7 contains coverage data for each benchmark.

**Small Benchmarks: Field-Sensitivity and Context-Sensitivity**. Over the small benchmarks (under 60 KLOC) there is little difference in fan-out and SCC size. The only points that stand out are fan-out for Unzip and Icecast, and SCC sizes for Icecast. First, let's consider Unzip and fan-out. By looking at the code of Unzip, we determined that field-sensitivity is the key property needed to conclude that the max fan-out for Unzip is 1. And indeed, our experiments show that ANDERS-FS and WL, both of which are field-sensitive, compute a max fan-out of 1. However, it is surprising that DSA, which is also field-sensitive, computes a max fan-out of 2. By looking further at this discrepancy, we found that DSA is overly conservative in this case because Unzip stores arrays inside of structures,

```
void avl_tree_free(..., int (*free_key_fun)(void *key )) {
1:   (*free_key_fun)(...);
}

void httpp_clear(...) {
2:  avl_tree_free(..., & _free_vars);
}

void source_free_source(...) {
3:  avl_tree_free(..., & _free_client);
}

static int _free_client(void *key ) {
4:  httpp_clear(...);  // cycle complete
}
```

Figure 4.8: How context-sensitivity reduces max SCC (for **Icecast**)

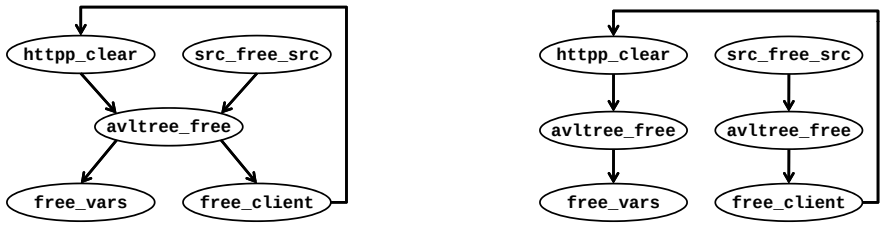



Figure 4.9: (a) Imprecise (b) Precise call-graph for **Icecast**

and DSA essentially loses field-sensitivity in these scenarios. Moving to Icecast, we found that many of the function pointers are stored in different fields of the same structure, and so field-sensitivity is helpful (compare Anders-FS with Anders-FI). Field-sensitivity also gives DSA an edge for Icecast over Anders-FI, but it still performs significantly worse than Anders-FS and WL.

Besides field-sensitivity, context-sensitivity makes a difference (but a small one) in Icecast, as evidenced by the fact that WL has a max SCC of 2 vs. 7. Figure 4.8 is simplified code that illustrates how the larger SCC has been divided by a context-sensitive approach. The function `avl_tree_free` is a generic free routine for AVL trees that is parameterized by a freeing function `free_key_fun`

```
g:228:builtin_merge_options ptsTo g:152:cmd_diff_files ::
  Base: g:168:commands = &g:152:cmd_diff_files @ test.c:58
  ComplexR: t:0:__TEMP__ = g:168:commands @ test.c:79
    l:171:p ptsTo g:168:commands ::
    Base: l:171:p = &g:168:commands @ test.c:77
  Base: l:166:cmd = t:0:__TEMP__ @ test.c:79
  ComplexL: l:162:_a50_1462_test_1 = l:166:cmd @ test.c:54
    l:164:c__heapify__0 ptsTo l:162:_a50_1462_test_1 ::
    Base: l:164:c__heapify__0 = &l:162:_a50_1462_test_1 @ test.c:50
  ComplexR: t:4:__TEMP__ = l:162:_a50_1462_test_1 @ test.c:212
    l:244:num_hits ptsTo l:162:_a50_1462_test_1 ::
    Base: l:164:c__heapify__0 = &l:162:_a50_1462_test_1 @ test.c:50
    Base: l:251:data = l:164:c__heapify__0 @ test.c:55
    Base: l:247:data = l:251:data @ test.c:227
    FP: l:244:num_hits = l:247:data @ test.c:220
      l:246:fn ptsTo g:239:get_remote_group ::
      Base: l:250:fn = &g:239:get_remote_group @ test.c:213
      Base: l:246:fn = l:250:fn @ test.c:227
  ComplexL: l:210:_a154_4011_test_2 = t:4:__TEMP__ @ test.c:212
  \\ more of flow...
```

Figure 4.10: Example value flow witness (for a prefix of example 4.11)

for the keys of the AVL tree. The `avl_tree_free` routine is then called in two contexts (lines 2 and 3), each time passing a different freeing routine for the keys. The freeing routine passed in at one callsite of `avl_tree_free` (line 3) calls back into the function that contains the other callsite to `avl_tree_free`. A context-insensitive call-graph for this example — shown in Figure 4.9(a) — would contain a spurious cycle, whereas the context-sensitive call-graph that WL computes — shown in Figure 4.9(b) — eliminates the cycle.

**Medium Benchmarks and Field-Sensitivity.** The benchmarks between 60 KLOC and 200 KLOC are Mutt, Git, and Smtpd. The largest difference is between field-sensitive Andersen's (Anders-FS) and field-insensitive Andersen's (Anders-FI). For example, the max SCC reported by Anders-FI for Git is 449 while the max SCC reported by Anders-FS is 13. WL only reduces this number slightly, to 6. This points to the fact that context-sensitivity does not provide a

significant amount of additional precision, beyond field-sensitivity.

To investigate how field-sensitivity helps in these cases, we need to find a small code snippet exemplifying the coding patterns that causes the difference. In a program with 200 KLOC this can be non-trivial, and so we used the following methodology to find small examples to report. First, we used Delta Debugging [ZH02] to reduce the code size of Git while preserving the property that ANDERS-FI reports a "large" SCC (e.g., 20). With Delta Debugging, we were able to reduce the code from 171 KLOC to 300 lines. Despite the reduced code size, understanding the nature of the imprecise flow for ANDERS-FI was still difficult. The next step then was to augment ANDERS-FI so that, in addition to the points-to graph, it also generates a witness of the value flows that it computes. Figure 4.10 shows a snippet of an example witness. Using this information, we found the code shown in Figure 4.11.

Essentially, the example code shows how the function pointers stored in the array (of 93 elements) at line 1 can flow to a call-site for which it does not belong. Flow begins at line 1, where a pointer to function `cmd_diff` is stored in the field `fn` of a structure that is in the array `comms`, along with many other functions. Because ANDERS-FI is field-insensitive, this pointer value flows out of the array and structure at line 2 through a *different* field, `cmd`, and ends up in the `cmd` field of structure `con`, declared at line 3.

Strangely enough, ANDERS-FI will conclude that this value ends up flowing from `con` into a field of the structure `opt`, which is defined at line 8. To see how this happens, we first observe that ANDERS-FI's context-insensitivity causes it to conclude that line 7 can call `get_remote_group`, since the call to `git_config` from line 6 passes &`get_remote_group` as the first parameter. Thus, any value passed to the `data` parameter of `git_config` flows to the last parameter of `get_remote_group` (namely `num_hits`). Now, notice that both &`opt` and &`con` are passed to the `data` parameter of `git_config` (on lines 4 and 9), meaning that they both flow to `num_hits`, and thus to `tmp` on line 5. Finally, the assignment on line 5 copies values between all targets of `tmp`, creating a spurious flow from `con` to `opt`.

Summarizing what has happened so far, ANDERS-FI concludes that the

```
struct cmd_struct {
   char const *cmd;
   int (*fn) (...);
};

struct cmd_struct comms[93] = {
1: {"diff", &cmd_diff},
};

handle_internal_command (...) {
2: check_pager_config(comms[i].cmd);
}

check_pager_config (char const *cmd) {
3: struct pager_config con;
   con.cmd = cmd;
4: git_config (&pcg, (void *)(&con));
}

get_remote_group (..., void *num_hits) {
   int *tmp = (int *) num_hits;
5: *tmp = *tmp + 1;
6: git_config (&get_remote_group, ...);
}

git_config (int (*fn) (...), void *data) {
7: (*fn) (..., data);
}

cmd_grep (...) {
8: struct grep_opt opt;
9: git_config (&grc, (void *)(&opt));
10: strbuf_addch (..., opt.cm[0]);
}
```

Figure 4.11: Importance of either field-, or context-sensitivity (from **Git**)

pointer to the function `cmd_diff` flows (1) from `comms` to `con`, due to field-insensitivity and (2) from `con` to `opt`, due to context-insensitivity. The pointer to `cmd_diff` will then continue its journey in a similar manner starting at line 10 from `opt` and will eventually reach an indirect callsite that it should not reach (which we do not show here). All the steps required to get the entire spurious flow from the `comms` array to the indirect call site are similar to either step (1) or (2) above. Thus, either field-sensitivity or context-sensitivity will stop these spurious flows, as evidenced by the Git results for Anders-FS and WL. Because DSA is also context-sensitive and field-sensitive, we suspect that it would also be able to prevent these spurious flows , but DSA did not finish on Git (it ran out of memory).

**Largest Benchmarks and Inclusion vs. Unification.** For the 3 largest benchmarks Nethack, Vim, and Emacs (over 200 KLOC), we find that there is little difference between WL, Anders-FS, and Anders-FI in terms of max SCC size and max fan-out. The only difference is in average fan-out. However, there is a significant gap between the unification-based Steens-FI (to which all bars are normalized) and the inclusion-based approaches. For example, the max fan-out for Emacs drops from 1148 for Steens-FI to 62 for Anders-FI and 28 for Anders-FS and WL. We also notice that these applications have very large SCCs, even when function pointers are ignored (that is to say, the NoFP bar has over 1000 functions).Finally, we noticed that all the pointer analyses perform equally poorly for Vim. Further investigation revealed the code seen in Figure 4.12, which shows how Vim makes heavy use of arrays to store function pointers. Because pointer analyses use a single summary node to represent all elements of the array, the call at line 3 has a fan-out of at least 72 (there are 72 unique functions pointed to by the elements of `nv_cmds`).

### 4.3.3 Recap

**Field-sensitivity and inclusion constraints matter most:** Our experiments demonstrate that, for the purpose of building call-graphs for C programs, field sensitivity and inclusion constraints are the two biggest contributors to preci-

```
struct nv_cmd {
   void (*cmd_func)(cmdarg_T *cap ) ;
   //...
};

struct nv_cmd nv_cmds[182]  = {
   { &nv_error, /* ... */ },
   { &nv_mouse, /* ... */ },
   { &nv_mouse, /* ... */ }, //...
};

void normal_cmd(oparg_T *oap , int toplevel ) {
1:  c = safe_vgetc();
    if (*) {
      ca.cmdchar = c;
    } else { ... }
2:  idx = find_command(ca.cmdchar);
3:  (*(nv_cmds[idx].cmd_func))(& ca); // Fans-out 72 ways
}
```

Figure 4.12: Smallest of 3 large function pointer arrays in **Vim**

sion and both are present in Anders-FS. Moreover, when comparing WL to Anders-FS, recall that there are three main differences: (1) flow-sensitivity, (2) context-sensitivity, and (3) heap-cloning. Our experiments show that there is little difference between WL and Anders-FS, meaning that these three additional components do not contribute a substantial amount of additional precision.

**SCCs are large even ignoring function pointers:** Our experiments also reveal that large C benchmarks may have large SCCs even if function pointers are completely ignored. Given the high lower-bound for max SCC in our largest benchmarks (as seen in the NoFP bar), inter-procedural client analyses that operate on one SCC at a time may need to decompose the problem further to improve scalability.

```
void callee(int *p) {        void caller(int *z) {
   if (p) {                     int x; int *y = NULL;
0:   init(*p);               2: callee(&x);
   }                         3: callee(y);
1: use(*p);                  4: callee(z);
}                            }
```

Figure 4.13: Illustrating example for a bottom-up null-safety analysis

## 4.4 Effect on Client Analyses

Aside from measuring the precision of various C pointer analyses for the purpose of constructing call-graphs, our goal is to also measure the effect of call-graph construction on client inter-procedural analyses. To this end, we have implemented an inter-procedural null-pointer safety analysis, which runs bottom-up on the call-graph, analyzing one SCC at a time. In addition to this null-pointer analysis we have also adapted Relay to make use of the constructed call-graphs. Both of these analyses are representative of the many inter-procedural analyses that use a bottom-up-one-SCC-at-a-time approach in order to scale, for example the ESP type-state verifier [DLS02, HDWY06a], the Saturn analysis system [XA05] and recent work on Shape analysis [CDOY09].

We first give a description of the null-pointer analysis (Section 4.4.1), then present our experimental results for the null-pointer analysis (Section 4.4.2) and the Relay race detector (Section 4.4.3), and finally recap the main conclusions that can be drawn from these experiments (Section 4.4.4).

### 4.4.1 Inter-procedural Null Pointer Analysis

**Overview.** Our null pointer analysis proceeds in two phases. The first computes *may-be-null* and *must-be-non-null* information for each pointer at each program point in a function. The second pass uses the nullness information to check the safety of dereferences in that function.

As the analysis is bottom-up on the call-graph, there are cases in which

the state of a pointer $p$ is unknown when analyzing a callee because of information missing from callers. For example, in the code snippet from Figure 4.13, the dereference at line 1 cannot be verified when analyzing `callee`. Instead of conservatively saying that the pointer may be null, we instead add a pre-condition for the function requiring $NonNull(p)$. A dereference whose check is deferred with a pre-condition is unsafe if any caller violates the pre-condition. For example, the call at line 2 is verified to be safe, but the call at line 3 is certainly unsafe. Therefore the use at line 1 is unsafe. If a caller is also unable to verify a pre-condition, the pre-condition itself may be deferred further up the call-chain. In the example, the pre-condition for the call at line 4 will be deferred to the callers of `caller` (that is, callers of `caller` will have to guarantee that the parameter to `caller` is non-null).

For our experiments we are only concerned with the precision of the call-graph, and so we use a fixed pointer-analysis (namely ANDERS-FI) for the other cases in which pointers are involved during the null-pointer analysis (e.g., to check aliasing between lvalues when handling writes).

**First Pass: Nullness States.** The first pass of our analysis computes sets of pointers that may be null. To capture what information has changed *relative* to the beginning of the current function and to distinguish between non-null, null, and unknown caller-provided information, we track *relative nullsets* at each program point. A relative nullset is a pair of disjoint sets of lvalues $(N_+, N_-)$. $N_+$ contains pointers that *must* be non-null due to program statements since the beginning of the function while $N_-$ contains pointers that *may* be null due to program statements since the beginning of the function. The nullness of any pointers that are in neither of the sets is determined by the caller. After a function is analyzed, its effects on the nullness of pointers can be summarized by the relative nullset at the exit node of the function. These summaries can then be used by callers to take into account the effects of calls.

More formally, we denote by *Lvals* the set of all lvalues, and $N = 2^{Lvals} \times 2^{Lvals}$ the set of all relative nullsets. The lattice of facts then is $(N, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$, where the ordering is defined as:

- $\bot = (Lvals, \emptyset)$, $\top = (\emptyset, Lvals)$
- $(N_+, N_-) \sqsubseteq (N'_+, N'_-)$ iff $N'_+ \subseteq N_+ \wedge N_- \subseteq N'_-$
- $(N_+, N_-) \sqcup (N'_+, N'_-) = (N_+ \cap N'_+, N_- \cup N'_-)$
- $(N_+, N_-) \sqcap (N'_+, N'_-) = (N_+ \cup N'_+, N_- \cap N'_-)$

Dataflow information is initialized with $(\emptyset, Locals)$ at the beginning of a function. $N_+$ being $\emptyset$ corresponds to the fact that no pointers have changed since the beginning of the function, and $N_-$ being *Locals* corresponds to the fact that locals (which are not initialized yet) may be null. All other program points are initialized to $\bot$.

The transfer functions are straightforward (Figure 4.14(b)). When analyzing a statement, if we can determine that a pointer has definitely become non-null, we add it to $N_+$, and remove it from $N_-$. Conversely, when a pointer becomes possibly null, we add it to $N_-$, and remove it from $N_+$. The helper functions Plus and Minus are used to add elements to $N_+$ and $N_-$, respectively, while maintaining the invariant that they are disjoint. For (possibly indirect) function calls, we use the call-graph to compute the set of possible targets for the call. For each target function, we look up its summary (which is a relative nullset) and rename all the lvalues in the summary to the caller's context. Once the summary is renamed, we simply apply the summary to the information before the call to get the information after the call. In particular, if $(N_+, N_-)$ is the information before the call, and $(N^r_+, N^r_-)$ is the renamed summary, then the information after the call is $(\ (N_+ \cup N^r_+) \setminus N^r_-\ ,\ (N_- \cup N^r_-) \setminus N^r_+\ )$. When there are multiple targets for a call, we simply merge the results from all the possible calls.

**Second Pass: Checking Dereferences.** The second phase that checks dereference safety is straight-forward. It does not require iteration and simply uses the dataflow information computed in the first pass. When a dereference for a pointer $lv$, or a call with a pre-condition $NonNull(lv)$ is encountered, and $(N_+, N_-)$ is the relative set for that program point, the following decision is made:

- If $lv \in N_+$ then we count it as safe

$\mathsf{Plus} : N \times 2^{Lvals} \rightarrow N$

$\mathsf{Plus}((N_+, N_-), S) =$
  **return** $(N_+ \cup S, N_- - S)$

$\mathsf{Minus} : N \times 2^{Lvals} \rightarrow N$

$\mathsf{Plus}((N_+, N_-), S) =$
  **return** $(N_+ - S, N_- \cup S)$

$\mathsf{ApplyDiff} : N \times N \rightarrow N$

$\mathsf{ApplyDiff}((N_+, N_-), (N_+^d, N_-^d)) =$
  **let** $N'_+ = (N_+ \cup N_+^d) - N_-^d$ **in**
  **let** $N'_- = (N_- \cup N_-^d) - N_+^d$ **in**
  **return** $(N'_+, N'_-)$

$F : Node \times N \times PPoint \rightarrow N$

$F(l := \texttt{malloc}(\_), N^{in}, p) =$

$F(l := \&l, N^{in}, p) =$
  **return** $\mathsf{Plus}(N^{in}, \{l\})$

$F(l := l', (N_+^{in}, N_-^{in}), p) =$
  **if** $(l' \in N_+^{in})$ **:**
    **return** $\mathsf{Plus}((N_+^{in}, N_-^{in}), \{l\})$
  **else if** $(\exists\ l^r \in N_-^{in}.\mathsf{Alias}(l', l^r)$ **:**
    **let** $S = \{l^l \in Lvals \mid \mathsf{Alias}(l, l^l)\}$ **in**
    **return** $\mathsf{Minus}((N_+^{in}, N_-^{in}), S)$
  **else**
    **return** $(N_+^{in}, N_-^{in})$

$F(l := e, N^{in}, p) =$
  **let** $S = \{l' \in Lvals \mid \mathsf{Alias}(l, l')\}$ **in**
  **return** $\mathsf{Minus}(N^{in}, S)$

$F(\texttt{assume}(l \neq null), N^{in}, p) =$
  **return** $\mathsf{Plus}(N^{in}, \{l\})$

$F(\texttt{assume}(l = null), N^{in}, p) =$
  **return** $\mathsf{Minus}(N^{in}, \{l\})$

$F(call(e, e'), N^{in}, p) =$
  **let** $T = \mathsf{CallTargets}(p)$ **in**
  $\bigsqcup_{(f,c) \in T}$ **let** $N_f = \mathsf{NullEffects}(f, c)$ **in**
  **let** $N'_f = \mathsf{Rebind}(N_f, f, e')$ **in**
  $\mathsf{ApplyDiff}(N^{in}, N'_f)$

Figure 4.14: Non-null dataflow analysis

- If $lv \in N_-$ then we count it as unsafe
- If $lv$ is not in either set, we add the requirement $NonNull(lv)$ to the current function's pre-condition.

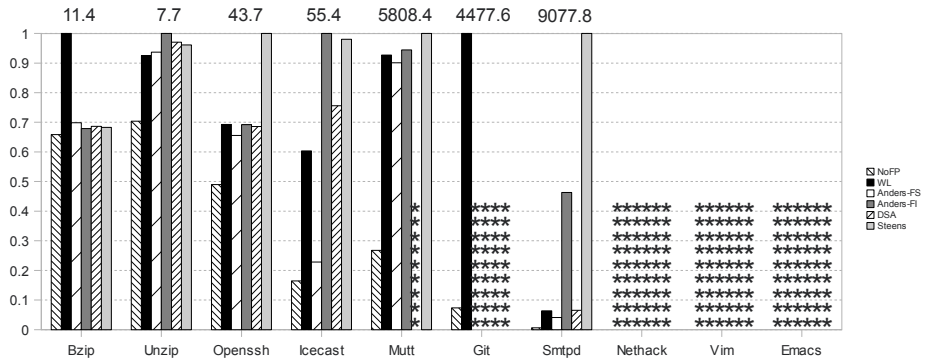



Figure 4.15: Time in seconds for client (normalized to max)

### 4.4.2 Results for Null-pointer Analysis

**Metrics of Interest.** Given the definition of the client analysis, let us present metrics of interest and briefly discuss (qualitatively) reasons in which a more precise call-graph can benefit the client.

- **Time and memory usage:** A more precise call-graph will (1) reduce the number of callee summaries to apply at each callsite, and (2) reduce the sizes of the SCCs. Smaller SCCs will reduce the maximum working set size, as all dataflow information for the SCC is kept in memory during analysis.

- **Number of dereferences shown safe:** A more precise call-graph will reduce the number of spurious callees. Fewer spurious callees translate to (1) fewer opportunities for a pre-condition to fail and hence a dereference to be shown unsafe, and (2) fewer spurious modifications that may nullify pointers.

**Results.** Figure 4.15 and 4.16 show the time and memory usage of the client analysis on each of the benchmarks, with each of the constructed call-graphs. The numbers are normalized for each benchmark to the largest of the values for that benchmark. Even though the time and memory measurements are fairly close for the small benchmarks, more generally, the trend is that smaller SCCs translate to faster running times and lower memory usage. For example, for Smtpd, NoFP

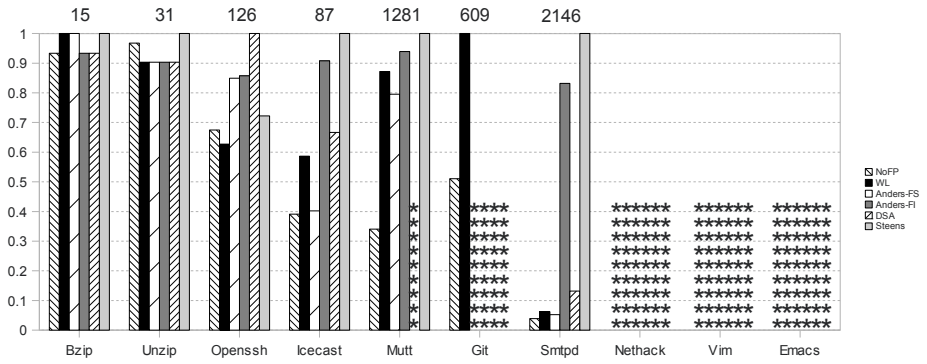



Figure 4.16: Memory used by client in MB (normalized to max)

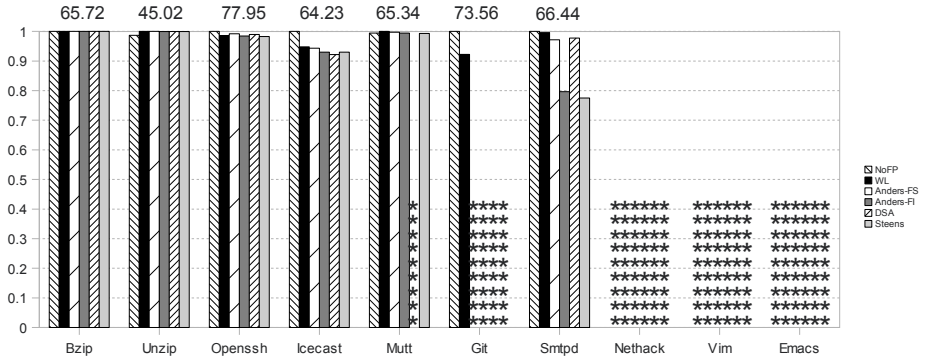



Figure 4.17: Precision of client as % of dereferences shown safe (normalized to max)

runs in 58 seconds using 84MB; WL, Anders-FS, and DSA finish in under 600 seconds using under 300 MB; Anders-FI takes over 4000 seconds and uses over 1.7GB of memory; and finally Steens-FI takes over 9000 seconds and over 2.1 GB. Finally, the three benchmarks with over 1000 functions in an SCC fail to complete within our memory budget using any of the call-graphs, as does Git when using Anders-FI and Steens-FI (449 max SCC). The client paired with Anders-FS does not exceed the memory budget for Git, but fails to complete within 12 hours.

Another result we have found is that although the max SCC can be smaller for WL than for Anders-FS, the client can still complete more quickly with Anders-FS. For example, the max SCC for Mutt is 246 functions for WL vs 304 for Anders-FS, but the client is over 150 seconds faster and uses 98MB less

memory when using the ANDERS-FS call-graph. This is explained by the fact that WL constructs a context-sensitive call-graph. Given a context-sensitive call-graph, the client must re-analyze a function additional times depending on how many additional contexts are created in the call-graph, for that function.

Finally, Figure 4.17 compares the precision of the client analysis with the various call-graphs. The precision numbers measure the percentage of dereferences that are proven safe in each of the cases. For each benchmark, the percentage values are normalized to the largest percentage value for that benchmark. We can see that for Smtpd, the precise call-graphs provide substantially better results than the imprecise ones (22% difference between WL and STEENS-FI). What may be surprising is that only three of the benchmarks show much difference between *any* of the pointer analyses, even when compared to the unsound NOFP. One possible explanation is that indirect calls only make up 3.5% of the total calls on average across the benchmarks (Figure 4.2). Another possible explanation is that many of the targets of the indirect calls are relatively pure (do not modify or nullify parameter pointers). For example, in Mutt, 689 out of the 743 indirect calls are for `printf` like functions, comparison functions supplied to a sort routine, or getter functions. Finally, it is also possible that the imprecision of the client is dominated by the Alias Analysis and not the call-graph analysis.

### 4.4.3 Results for Relay Race Detector

Aside from the non-null client, we have also adapted our RELAY data-race analysis [VJL07] to make use of the constructed call-graphs. The RELAY analysis only makes sense for multithreaded programs in our benchmarks suite, namely Icecast and Git.

The results for time, memory, and data-race warnings follow a similar trend to the null-pointer analysis. For Icecast, NOFP completes in 7 seconds w/ 52MB, WL completes in (320 s, 475 MB), ANDERS-FS in (17 s, 95 MB), DSA in (66 s, 97 MB), ANDERS-FI in (91 s, 100 MB), and STEENS-FI in (109 s, 125 MB). With this client, it is also the case that the context-sensitive call-graph can generate more work than a context-insensitive call-graph. In terms of precision, the different

call-graphs result in roughly the same number of data-race warnings except for NoFP, which leads to half as many warnings. For Git completes in (86 s, 293 MB), WL in (449 s, 1066 MB), Anders-FS in (238 s, 796 MB). The less precise analyses Anders-FI and Steens-FI are unable to finish without exceeding our 3GB memory budget. In terms of precision, the sound analyses are again very close, but here again the unsound NoFP reports substantially fewer warnings than Anders-FS.

### 4.4.4 Recap

**More precise call-graphs improve client performance:** Our results show that in general more precise call-graphs can dramatically improve the running time and memory usage of inter-procedural analyses that use a bottom-up one-SCC-at-a-time approach for scaling. This is due to the fact that the improved precision in the call-graphs reduces the size of the largest SCC.

**More precise call-graphs don't necessarily help client precision:** Our experiments also show that improved call-graph precision does not necessarily lead to improved precision for the client analysis. In particular, for the client analyses we considered, the primary purpose for getting a more precise call-graph is for scalability, rather than precision of the end results.

## 4.5 Other Call-graph Algorithms and Studies

We give a brief survey of previous research on callgraph construction and empirical comparisons of different value-flow analyses.

**Call-graphs for High-Level Languages.** There is a rich literature on the subject of call-graph construction for higher-level functional and class-based Object-Oriented languages. [Shi88] presented the now classical $k-$CFA for Scheme, that uses the call-strings method to determine the set of possible closures that reach a particular call-site. The method cannot be directly applied to C for a variety of reasons — including the fact that it relies on a prepass that translates the code

to continuation-passing style. The problem of call-graph construction has received much attention from researchers looking to optimize the overheads associated with dynamic dispatch in Object-Oriented languages. In this setting, the question reduces to determining, for each call-site, the particular classes a receiver object can belong to. There are a variety of type-based mechanisms, such as the Cartesian Product Algorithm [Age95] and flow-based algorithms such as [TP00] that can reconstruct this information. [GC01] describes a unified framework within which many of these algorithms (and others, like $k$-CFA) can be expressed and provides a detailed experimental comparison of the different techniques on a set of common benchmarks. However, all the above rely heavily on class information which is non-existent in C programs — indeed, C does not even help with determining which fields of a structure hold function pointers — and hence, cannot be directly applied to low-level systems code. The problem of call-graph construction for C has received some attention. [Atk04] describes ways to combine types, syntactic pattern-matching and run-time pointer information to precisely reconstruct call-graphs. While these techniques are applicable in the setting of program comprehension, they are unsound for static program analysis.

**Experimental Comparisons.** Several authors have presented experimental comparisons of different alias, points-to and value-flow analyses. Milanova, Ryder and Rountev [MRR04] show experimentally that a field-sensitive, context- and flow-insensitive algorithm from [ZRL96] suffices to construct precise call-graphs for C programs. However, these experiments were for benchmarks that are an order of magnitude smaller than ours. Our results agree that field-sensitivity is important. [DLFR01] studies how the one-level flow algorithm (OLF) fares against a generalized, context-sensitive version (GOLF) with regards to certain client optimizations. It uses a metric to represent client optimizations (instead of directly carrying out the optimizations), and moreover, the client is itself an intra-procedural analysis. [CDC+04] studies how a variety of alias analyses affect a set of compiler optimizations. However, the client optimizations considered in the paper are intra-procedural, and the most precise (but sound) analysis that was studied was [IH97] which is scalable but less precise than Anders-FI. [PKH04a] compares the sizes

of the points-to sets returned by field-insensitive and sensitive versions of Andersen's algorithm. The paper notes that field-sensitivity starts to become important in larger programs, but does not report on the effect on clients. Finally, Mock et al. [MACE02] measure the impact of Steens-FI in comparison to a lower bound (dynamic points-to data), for a program-slicing client. They find that results follow a bimodal distribution. For programs with few function pointers in use, slicing with Steens-FI is already close to the lower bound even though dereference set sizes are greatly improved. Only programs that use function pointers heavily benefit from a more precise points-to analysis (approximated by the dynamic points-to data). To explain this lack of improvement, they cite Amdahl's Law. For program slicing, they find that most of the data dependencies are induced by direct variable-to-variable assignments or parameter passing. Therefore, improved pointer information only affects a few of the data dependencies and yields few benefits. In a way, this is similar to our findings. We find that precision of some clients is not improved by a more precise callgraph, since indirect function calls make up a small fraction of all function calls for C programs.

# Chapter 5

# Instantiating and Evaluating Radar

This chapter describes how to instantiate the RADAR framework using RELAY. We call this instantiation RADAR(RELAY). This instantiation is then applied to several sequential dataflow analyses to obtain versions that are sound when the analyzed program contains multiple threads sharing memory. This chapter ends with experimental results demonstrating the effectiveness of such conversions compared to appropriate upper and lower bounds.

## 5.1 Putting Relay into Radar

We now show how to instantiate the three functions *Reg*, *RacyRead*, and *SumReg* using RELAY. The result is an instantiation of RADAR using RELAY, namely RADAR(RELAY).

**Region Map.** We define regions as $R = \mathit{Funs} \times \mathit{Locks}$, where *Funs* is the set of function identifiers. Given a program point $p$, $\mathit{Reg}(p)$ returns $(g, (L_+, L_-))$, where $g$ is the function to which $p$ belongs, and $(L_+, L_-)$ is the relative lockset computed at $p$ by the RELAY lockset analysis.

**Summary Map.** To define the *SumReg* function, we first define a helper function *AllUnlocks*. Intuitively, the set $\mathit{AllUnlocks}(cs)$ represents an over-approximation

of the set of locks that could possibly be released by performing the call at $cs$. In particular, given a call-site $cs$, $AllUnlocks(cs)$ computes the union of all the $L_-$ sets in the function being called at $cs$, and then converts this set into the caller context. The conversion consists of replacing the callee's formals that occur in the locksets with the parameters passed in at the call-site.

Given a call-site $cs \in CS$ where function $h$ calls $g$, and given that $(L_+, L_-)$ is the relative lockset computed by RELAY for the program point right before the call to $g$ at $cs$, then $SumReg(cs)$ is defined as follows:

$$SumReg(cs) = (h, (L_+ - AllUnlocks(cs), L_- \cup AllUnlocks(cs)))$$

Essentially, *SumReg* subtracts the *AllUnlocks* set from the locks that were held before the call is made. The result of this subtraction is a conservative approximation of the set of locks that are guaranteed to remain locked *at all points* during the call.

**Race Detection Engine.** Given a region $r = (g, (L_+, L_-))$ and an lvalue $l$, $RacyRead(r, l)$ conceptually runs a full RELAY analysis bottom-up, except that when it analyzes function $g$, it inserts an additional guarded access to the guarded access set. The additional guarded access is the triple $(l, (L_+, L_-), read)$, indicating that we are simulating a read of $l$ with a lockset of $(L_+, L_-)$. The $RacyRead(r, l)$ function returns *true* if and only if, after being propagated up to the thread roots, this pseudo-read leads to a RELAY race warning.

This conceptual description of $RacyRead(r, l)$ is not how we implement it, since each call to *RacyRead* would lead to an entire RELAY bottom-up analysis of the program. Instead, we structure the execution of RADAR(RELAY) into the following four passes, two of which are the RELAY bottom-up analysis.

- **First pass.** RADAR(RELAY) runs the bottom-up RELAY analysis to compute the relative locksets at each program point, and hence, the race equivalence regions and the summaries needed for *AllUnlocks*.

- **Second pass.** RADAR(RELAY) runs the sequential analysis on the entire program with a *RacyRead* function that returns false all the time. This has

the effect of running the sequential analysis without any adjusting, but it allows RADAR(RELAY) to collect the parameters of all the *RacyRead* queries into a set $S$ of $(r, l)$ pairs. Since the sequential analysis computes a superset of the facts computed by the adjusted analysis, the set $S$ is a superset of the queries that the adjusted analysis will make.

- **Third pass.** RADAR(RELAY) then runs the bottom-up RELAY analysis again to insert pseudo-accesses. In particular, when analyzing a function $g$, for each pair $(r, l) \in S$ where $r = (g, (L_+, L_-))$, RADAR(RELAY) adds the guarded access $(l, (L_+, L_-), read)$ to the guarded access summary of $g$. RADAR(RELAY) uses the results of this second RELAY run to build a map $RelayResults : S \rightarrow Bool$. Given $(r, l) \in S$, $RelayResults(r, l)$ returns whether or not the pseudo-read inserted for $(r, l)$ caused a race warning.

- **Fourth pass.** Finally, RADAR(RELAY) runs the sequential analysis again, but this time performs the adjusting process. To do so, RADAR(RELAY) uses the *RelayResults* map computed during the third pass to answer the *RacyRead* queries.

**Reuse of Passes Between Analyses.** If a client has several sequential analyses, for example lazy code motion depends on several uni-directional dataflow analyses, then pass one and pass three (the data-race engine passes) can be shared by all of the dataflow analyses, assuming that the dataflow analyses are not dependent upon each other. Reuse of pass one is obvious because it does not depend on the sequential analysis at all. Reuse of pass three is possible, given sufficient book-keeping, because each sequential analysis only generates pseudo-*reads* and reads cannot race with each other. Therefore the race-engine queries for the different analyses do not interfere with one another during pass three.

**Example: Relative Locksets.** We now illustrate how RADAR(RELAY) would analyze the program in Figure 2.5 with the non-null analysis.

- **First pass.** RELAY computes the *Reg* map where $Reg(p)$ is:

$$\begin{cases} (\texttt{Producer}, (\emptyset, \emptyset)) & \text{if } p \in \{\texttt{P0}\} \\ (\texttt{Producer}, (\{\texttt{buf\_lock}\}, \emptyset)) & \text{if } p \in \{\texttt{P3}, \texttt{P4}, \texttt{P5}, \texttt{P6}, \texttt{P8}, \texttt{P9}\} \\ (\texttt{Producer}, (\emptyset, \{\texttt{buf\_lock}\})) & \text{if } p \in \{\texttt{P1}, \texttt{P2}, \texttt{PA}\} \\ (\texttt{Consumer}, (\emptyset, \emptyset)) & \text{if } p \in \{\texttt{C0}\} \\ (\texttt{Consumer}, (\{\texttt{buf\_lock}\}, \emptyset)) & \text{if } p \in \{\texttt{C3}, \texttt{C4}, \texttt{C5}, \texttt{C6}, \texttt{C7}\} \\ (\texttt{Consumer}, (\emptyset, \{\texttt{buf\_lock}\})) & \text{if } p \in \{\texttt{C1}, \texttt{C2}\} \\ (\texttt{foo}, (\emptyset, \emptyset)) & \text{if } p \in \{\texttt{G0}\} \\ (\texttt{foo}, (\emptyset, \{\texttt{buf\_lock}\})) & \text{if } p \in \{\texttt{G1}\} \end{cases}$$

  Notice that both locksets are empty at the first point in each function meaning the lockset is trivially the same as at the entry point. Using the above *Reg* map, RADAR(RELAY) determines:

$$\mathit{AllUnlocks}(\texttt{P5}) = \{\texttt{buf\_lock}\}$$

  as the `buf_lock` is released inside `foo`. Thus,

$$\mathit{SumReg}(\texttt{P5}) = (\texttt{Producer}, (\emptyset, \{\texttt{buf\_lock}\}))$$

- **Second pass.** In the second pass, RADAR(RELAY) runs a sequential non-null analysis which generates the flow facts shown on the left in Figure 2.5, *including* the facts crossed out by a line. Using these facts, RADAR(RELAY) computes the superset $S$ as the set of tuples $\{(Reg(p), l) \mid NonNull(l) \text{ at } p\}$.

- **Third pass.** RADAR(RELAY) then inserts pseudo-reads corresponding to the queries $S$ generated above, and builds the map *RelayResults* which yields the following *RacyRead* map:

$$\begin{aligned} &\mathit{RacyRead}((g, (L_+, L_-)), l) = \\ &\qquad (l = \texttt{px->data} \wedge \texttt{buf\_lock} \notin L_+) \\ &\qquad \vee\ (l = \texttt{cx->data} \wedge \texttt{buf\_lock} \notin L_+) \end{aligned}$$

- **Fourth pass.** When the adjusted sequential analysis is performed, the fact *NonNull*(`px->data`) gets killed at `P5` since the summary region at that call-site does not include `buf_lock` in $L_+$. The result is shown on the left in Figure 2.5 – the dataflow solution includes only the facts that are *not* crossed out.

## 5.2 Analyses Converted by Radar(Relay)

The RADAR framework has been tested with several dataflow analyses, including both forwards and backwards directional analyses. Some analyses are targeted towards optimizations and others for checking safety properties.

One purpose of evaluating RADAR against multiple clients is to show that the framework is general – each client sequential analysis uses the same simple interface. The client is only expected to expose three functions (1) the flow function $F$, (2) a function enumerating lvalues that a fact depends upon *Lvals*, and (3) a function for removing elements from a set of flow facts as dictated by *ThreadKill*.

The tested analyses are:

- **Constant Values.** For each program point, this analysis computes sets of lvalues holding constants (*ConstValue*(`x`, `c`)). This analysis is used to perform the constant propagation optimization. Constant folding is also done during the analysis using the dataflow facts. It is an intra-procedural forward must analysis and is the canonical non-distributive dataflow analysis. Procedure calls are handled by checking a summary of modified lvalues and killing facts that are modified by the call. A sound version of the analysis would avoid the incorrect optimization shown in Figure 5.1, where a busy-wait loop is optimized into an infinite loop. A sequential-minded analysis mistakenly deduces that the `flag` variable invariantly holds the value `1` during the loop, ignoring the effects of the second thread.

- **Very Busy Expressions (or, anticipatable expressions).** For each program point, this computes a set of expressions that are (1) evaluated

```
Thread 1            Thread 2          Thread 1          Thread 2
 flag = 1;                             flag = 1;
 spawn th2;          doWork();         spawn th2;        /*
 while (flag){}      flag = 0;   ⇒     while (1){}       unchanged
 proceed();                            proceed();        */
```

Figure 5.1: Unsound constant analysis leads to non-termination

along every path after the given program point, and (2) none of the lvalues in the expression are re-defined before the evaluation. It serves as one of the three analyses used to perform the lazy code motion optimization [KRS92]. It is an intra-procedural backwards must analysis. Again, procedure calls are handled by checking a summary of modified lvalues. A sound version of the analysis would avoid the incorrect optimization shown in Figure 5.2, which ignores the re-definition of `y` made by the second thread and causes thread one to loop forever.

- **Non-Null.** This computes sets of lvalues that hold non-null values, and therefore can be safely dereferenced. It is an inter-procedural forwards must analysis. It uses the relative dataflow framework and is described in Section 4.4.1. The *must* set in the relative dataflow formulation is adjusted by Radar, but the *may* set only tracks thread-local may information and therefore is not adjusted by Radar. The may information serves as a more precise modifies summary. A sound version would catch bugs like those described in the overview section and shown in Figure 2.2.

## 5.3 Evaluation

Another purpose of evaluating Radar against multiple clients (and multiple benchmarks) is to extrapolate trends in the *precision* of our approach. This section describes such measurements comparing the Radar(Relay)-generated versions of each analysis to lower and upper bounds.

```
Thread 1                Thread 2        Thread 1                Thread 2
                                        x = 0; y = 0;
x = 0; y = 0;                           t = 10*y;
spawn th2;                              spawn th2;              /*
while (x < 1000)        y = 1;    ⇒     while (x < 1000)        unchanged
  x = x+(10*y);                           x = x+t;              */
x = x+(10*y);                           x = x+t;
```

Figure 5.2: Unsound VBE analysis can cause non-termination

### 5.3.1 Alternative Instantiations and Bounds

The main component of RADAR that determines its precision is the black box that answers *RacyRead* queries. RADAR(RELAY) is one instantiation, but it is also possible to instantiate more scalable but less precise versions, or even an upper bound on precision by changing the black box. This section compares four different instantiations.

**Steensgaard-based Instantiation.** The simplest and least precise instantiation we consider is based on Steensgaard's pointer analysis [Ste96a], and we call this instantiation RADAR(STEENS). For this instantiation, $RacyRead(r, l)$ ignores the region $r$ it is passed and returns true if $l$ is reachable from a global or a thread parameter, according to the Steensgaard's points-to graph. This matches our intuition that lvalues that cannot be reached from globals and thread parameters cannot be shared, and thus cannot be racy.

**Relay-based Instantiations.** We have already described the RELAY-based instantiation of RADAR in Section 5.1. However, for the purpose of better understanding where the precision of RADAR(RELAY) is coming from, we separate RADAR(RELAY) into two instantiations based on the observation that RELAY can prove the absence of a race in two different ways: given a write and another access to two lvalues in two different threads (1) show that the two lvalues do not alias; and (2) if they can alias, show that the intersection of the locksets is non-empty.

To better understand how these two different ways of showing the absence of a race contribute to RADAR(RELAY), we separate the instantiation into two

parts, RADAR(RELAY$_{\neg L}$) and RADAR(RELAY). We have already seen the latter; it is just as described in Section 5.1. The former is a version of RADAR(RELAY) where we change the *Reg* map to always return $\top$, which represents the empty set of locks. This modification simulates a version of RELAY that only answers race queries based on possible aliasing relationships and the existence of accesses in at least two threads (with at least one write), but not on locksets.

**Optimistic Instantiation.** The last instantiation we consider is the most optimistic possible: the one where *RacyRead* always returns *false*. We call this version SEQ because it is equivalent to the sequential analysis without any adjusting. Although this instantiation of RADAR is unsound, it establishes an upper bound on how well any adjusted analysis can do.

Each one of these instantiations – RADAR(STEENS), RADAR(RELAY$_{\neg L}$), RADAR(RELAY), and SEQ – is more precise than the previous, with the last one being unsound.

### 5.3.2 Instantiations with Varying Call-graphs

It is possible to arrive at different instantiations of RADAR by varying the underlying callgraph construction algorithm used by the different components (*e.g.* by RELAY or by the dataflow analyses under adjustment). The goal of the experiments in this section is to compare the precision of each instantiation. However, results from Section 4.4 indicate that more precise call-graphs do not necessarily increase the precision of client analyses. A more precise callgraph does, however, improve the scalability of inter-procedural analyses like RELAY and the non-null dataflow analysis. Therefore, in this section we simply pick the most precise call-graph that can be generated within resource limits, for each benchmark.

### 5.3.3 Radar Benchmarks

To compare each instantiation of our framework, we have collected the eight benchmarks shown in Figure 5.3, totaling over one million LOC. Briefly, the benchmarks consist of two media servers Icecast and mtdaapd, the Retawq

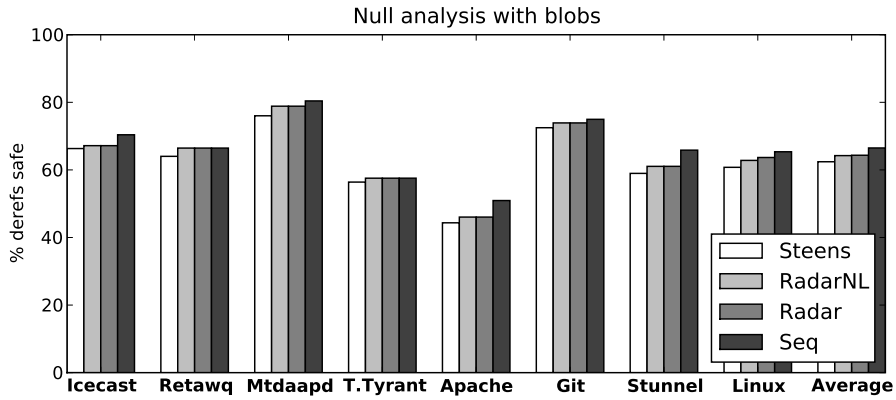
| Benchmark | KLOC | Call-graph | max SCC |
|---|---|---|---|
| **Icecast** | 28 | WL | 2 |
| **Retawq** | 40 | WL | 90 |
| **Mtdaapd** | 57 | WL | 9 |
| **TokyoTyrant** | 101 | WL | 1 |
| **Apache** | 142 | Anders-FS | 575 |
| **Git** | 170 | WL | 6 |
| **Stunnel** | 183 | Anders-FS | 1180 |
| **Linux** | 830 | Per-File | 144* |

Figure 5.3: Radar Benchmark characteristics

text-based web browser, the Apache web server configured to handle requests with threads instead of processes, the Git distributed version control system, the Tokyo Tyrant lightweight database server, the stunnel program which encrypts any client TCP connection with OpenSSL, and a subset of the Linux kernel. The table also indicates the most precise call-graph generated for each benchmark. If it is the case that even the most precise callgraph contains an SCC which is too large for Relay, we fall back on the unsound call-graph generated by a per-file pointer analysis. This is only the case for Linux.

### 5.3.4 Running Times and Memory Usage

Running all four passes of Radar(Relay) on a single 3.2 GHz Pentium 4 machine never requires more than 2GB of RAM. Averaging over all three analyses (non-null, constant propagation, and very-busy expressions), the running time of Radar(Relay) ranges from less than a minute on small benchmarks, all the way to 9.7 hours on Linux, with an average of 2.2 hours. The longest-running benchmark was Linux (9.7 hours), because of its many lines of code and threads. The next longest-running benchmark was Stunnel (4.5 hours), because of its large SCC of over 1,000 functions. All other benchmarks ran in less than 3.2 hours. Although parallelizing Radar(Relay) for Stunnel is a challenge because of its large SCC, in the case of Linux, the implementation of Radar(Relay) can easily be parallelized in the same way as Relay [VJL07] by analyzing independent call-




| | Icecast | Retawq | Mtdaapd | T.Tyrant | Apache | Git | Stunnel | Linux | Average |
|---|---|---|---|---|---|---|---|---|---|
| **Radar** | 21.0 | 100.0 | 64.6 | 99.6 | 26.5 | 56.4 | 45.3 | 63.6 | 59.6 |
| **RadarNL** | 21.0 | 100.0 | 64.6 | 99.6 | 26.5 | 56.4 | 45.2 | 45.3 | 57.3 |

Figure 5.4: Percentage of all dereferences proven safe by each instantiation (top), and percentage of gap bridged (bottom).

graph SCCs in parallel. This would allow Radar(Relay) to run much faster on a cluster of nodes.

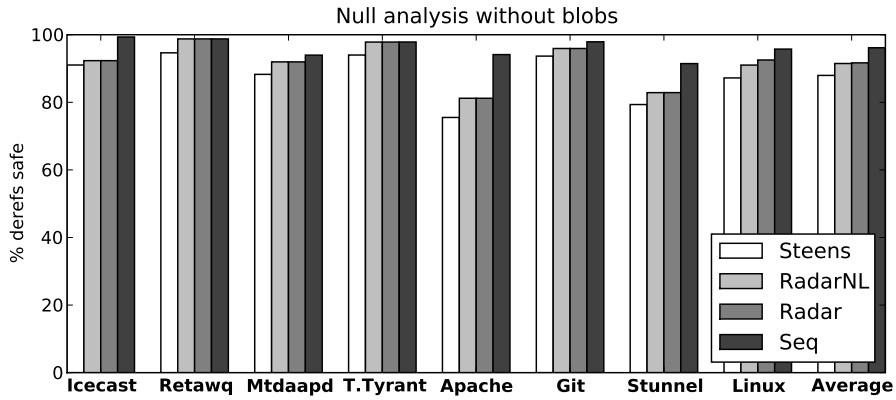
### 5.3.5 Comparison of Precision

#### Null Analysis Results

Figure 5.4 shows the number of dereferences proven safe, as a fraction of all dereferences, by each of the four Radar-adjusted analyses. As expected, the size of each bar grows from left to right, indicating that each analysis is more precise.

The first thing to notice is that for each benchmark the sequential analysis (the fourth bar in each cluster) can prove only a small percentage of dereferences safe, between 44.35% and 80.41%. Thus, no matter how precise the adjusting process, the resulting multithreaded analysis will not be able to prove the safety of a significant number of dereferences.

The imprecision in the sequential non-null analysis is mostly due to imprecision in analyzing the heap. The alias analysis we use merges many of the lvalues on the heap into "blob" nodes, thus losing precision for heap-allocated variables.




| | Icecast | Retawq | Mtdaapd | T.Tyrant | Apache | Git | Stunnel | Linux | Average |
|---|---|---|---|---|---|---|---|---|---|
| **Radar** | 17.3 | 100.0 | 63.7 | 99.2 | 29.2 | 53.3 | 42.0 | 63.1 | 58.5 |
| **RadarNL** | 17.3 | 100.0 | 63.7 | 99.2 | 29.2 | 53.3 | 41.9 | 46.8 | 56.4 |

Figure 5.5: Percentage of non-blobby dereferences proven safe (top), and percentage of gap bridged (bottom).

Previous null-pointer analyses [DDA07] have also found that heap structures are hard to analyze precisely and lead to many false-positives when performing null-dereference checks. To factor this degree of imprecision out of our experiments, we plot in Figure 5.5 the percentages of safe dereferences to pointers *not* including those in blob nodes. These non-blobby dereferences account for a majority of dereferences – on average 76.37%. Considering these remaining dereferences, the sequential analysis is able to prove the safety of a majority of dereferences on each benchmark (again the fourth bar in each cluster). Although this filter may remove true bugs as well as false warnings unrelated to the heap analysis (*e.g.* false warnings from path-insensitivity), this simulates a precise version of the sequential analysis for testing Radar, without making the sequential analysis itself non-scalable.

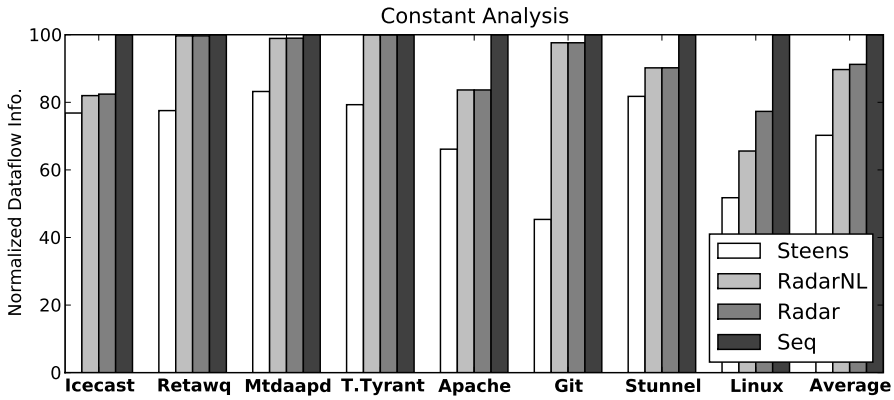
Recall that the sequential analysis is unsound in the concurrent setting. Nevertheless, because we cannot know what an oracle would provide as the "correct" answer for adjusting, we use Seq as an upper bound; the other three analyses, as well as the oracle, cannot do any better. At the other end of the spectrum is Radar(Steens), the least precise of the analyses. We included this Steensgaard-

based approach in our evaluation because it can easily be implemented in a compiler or program analyzer that needs to be sound but is not concerned with being extremely precise. We therefore use RADAR(STEENS) as a lower bound for comparison.

We now evaluate how the RADAR(RELAY$_{\neg L}$) and RADAR(RELAY) instantiations compare to the RADAR(STEENS) lower bound and the SEQ upper bound. We consider what percentage of this *gap* – the difference between the results of SEQ and RADAR(STEENS) – is bridged by the other two analyses. Keep in mind that because SEQ is an unsound over-approximation, the real gap – the difference between a perfect oracle and RADAR(STEENS) – may be smaller than the gap we consider. Thus, the percentages we report are in fact lower bounds on how much of the real gap we bridge. These results are presented along the bottom of figures 5.4 and 5.5. When all dereferences are considered (Figure 5.4), RADAR(RELAY) bridges on average 59.6% of the gap and RADAR(RELAY$_{\neg L}$) bridges 57.3%. When blobby dereferences are taken out (Figure 5.5), RADAR(RELAY) bridges on average 58.5% of the gap, and RADAR(RELAY$_{\neg L}$) bridges 56.4%.

The results on Linux are what we would expect: each analysis is incrementally better than the previous one. From left to right, each analysis incorporates, in the following order, a simple and fast alias analysis, a more precise thread sharing analysis, and a lockset analysis. As a result, each analysis better captures concurrency interactions in the program. This leads to more precise race detection, which is ultimately the factor that determines RADAR's effectiveness.

Other than Linux and Stunnel, however, RADAR(RELAY$_{\neg L}$) is just as effective as RADAR(RELAY). To better understand the small difference for the other benchmarks, recall how the two instantiations differ: RADAR(RELAY) uses the full version of RELAY, whereas RADAR(RELAY$_{\neg L}$) uses a version of RELAY that answers race queries based only on possible aliasing relationships and access types in each parallel thread (at least one must be a write), but not on locksets. The fact that RADAR(RELAY$_{\neg L}$) is nearly as precise as RADAR(RELAY) indicates that in many of the cases arising in our non-null analysis, the lvalue being adjusted is simply not shared. This corroborates with the results of [NAW06], which shows




| | Icecast | Retawq | Mtdaapd | T.Tyrant | Apache | Git | Stunnel | Linux | Average |
|---|---|---|---|---|---|---|---|---|---|
| **Radar** | 24.2 | 98.5 | 93.9 | 99.8 | 51.7 | 95.7 | 46.4 | 53.0 | 70.4 |
| **RadarNL** | 22.3 | 98.5 | 93.5 | 99.8 | 51.7 | 95.7 | 46.3 | 28.7 | 67.1 |

Figure 5.6: Amount of dataflow information for constant analysis (top), and percentage of gap bridged (bottom). Normalized to sequential analysis.

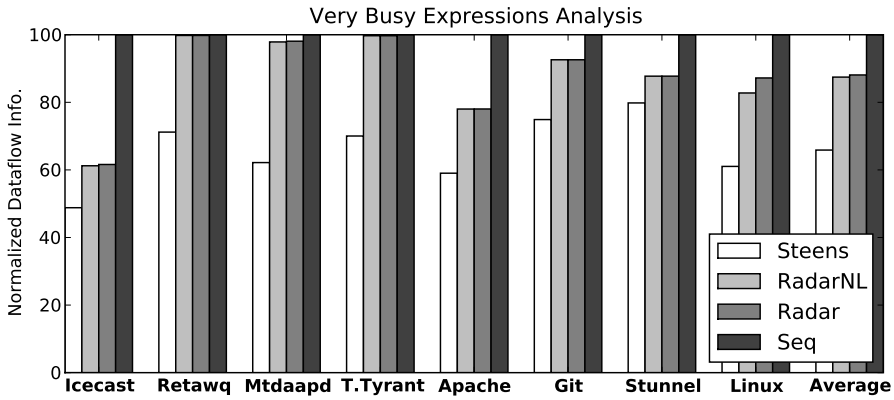
that in Java benchmarks a large majority of potential races are ruled-out by a precise sharing analysis.

Finally, note that for a few benchmarks – Retawq and Tokyo Tyrant – Radar(Relay) and Radar(Relay$_{\neg L}$) are able to bridge close to 100% of the gap. In both of these cases, the program only has a handful of threads other than the main thread, and these threads only read and write a small fraction of the global variables, none of which are pointer variables. For these benchmarks, Radar(Steens) is clearly overly conservative in assuming that anything reachable from a global is potentially racy.

### Constant Values Analysis and Very Busy Expressions

For the constants and very-busy expressions anlayses, we measured the following value for each benchmark: the number of dataflow facts that each analysis finds to hold true at each program point, summed over all program points. We then used the sum from the optimistic upper bound Seq as a baseline and normalized the sums.

Figures 5.6 and 5.7 contain these normalized dataflow counts. Overall,




| | Icecast | Retawq | Mtdaapd | T.Tyrant | Apache | Git | Stunnel | Linux | Average |
|---|---|---|---|---|---|---|---|---|---|
| **Radar** | 25.0 | 99.3 | 94.9 | 99.0 | 46.4 | 70.5 | 39.3 | 67.2 | 67.7 |
| **RadarNL** | 24.3 | 99.3 | 94.4 | 99.0 | 46.3 | 70.5 | 39.3 | 55.8 | 66.1 |

Figure 5.7: Amount of dataflow information for VBE analysis (top), and percentage of gap bridged (bottom). Normalized to sequential analysis.

the results follow similar trends as the results from the non-null analysis. First, Radar(Relay) bridges over 50% of the gap between Radar(Steens) and Seq, on average. Second, there is still very little difference between Radar(Relay) and Radar(Relay$_{\neg L}$), except on the Linux benchmark. However, there is now a more visible difference in some of the other benchmarks, Icecast being one of them.

One data point that stands out is that for the Git benchmark, under the constants analysis, there is an enormous gap between Radar(Steens) and the other instantiations of Radar. The reason is that Git has many functions with the following pattern: many fields of an array or heap structure are initialized at the beginning of a long function, forming dataflow facts that are propagated to the end of the function. Radar is able to determine that these heap structures are not shared and preserves these facts. On the other hand, Radar(Steens) is led to believe that these heap locations are shared and kills the dataflow facts immediately. Although the gap is large, the heap lvalues holding constants are only used a few times, so the amount of the gap bridged may not be a reliable indicator of the amount of performance gain to expect from optimization.

## 5.4 Summary

This chapter described several instantiations of RADAR including one based on the RELAY race detection engine and experimentally evaluated these instantiations.

Overall, our experiments on RADAR(RELAY) demonstrate the precision and scalability of RADAR. In each of the test cases, RADAR(RELAY) was able to bridge a sizable portion of the gap between optimistic and conservative concurrent dataflow analyses, while still producing a sound result. Finally, adapting each dataflow analysis and RELAY to the RADAR framework required few changes. The dataflow analysis designer did not have to explicitly worry about concurrency – all of the concurrency reasoning happened in the tunable race detection engine.

**Acknowledgments:** Chapter 5 in part, has been published as "Dataflow Analysis for Concurrent Programs using Datarace Detection" by Ravi Chugh, Jan Voung, Ranjit Jhala, and Sorin Lerner in *PLDI 08: Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation* [CVJL08]. The dissertation author was the primary investigator and author of this paper.

# Chapter 6

# Related Work

There is a large body of work in analyzing concurrent programs. This chapter compares RELAY (finding data-races) and RADAR (dataflow analysis for concurrent programs) to related approaches.

## 6.1 Datarace Detection

**Dynamic Datarace Detection.** Most race detection techniques used in practice are dynamic. These detectors principally use two techniques. The first is Lamport's happens-before relation [Lam78], represented dynamically by vector clocks, used in [DS91, MC93]. The second is dynamically computed locksets, popularized by [SBN+97]. The first approach handles more synchronization mechanisms to eliminate false positives, while the second has less overhead but may find false positives. Much recent effort has gone into lowering the overhead imposed by dynamic analysis – for example by using statically pre-computed locksets to prune redundant checks [CLL+02], or by using a combination of vector clocks and lighter-weight instrumentation that provide a fast-path for race-free accesses (the majority of accesses) [YRC05, PS07, FF09]. [MMN09] reduces overhead with sampling directed at cold code blocks, as races in hot code blocks should have already been found and fixed, or are benign. Other recent developments include the extension of these techniques to find *atomicity* [FQ03] violations in Java code [WS06, FF04a], and

the use of automated replay to determine whether a given dynamically detected race is benign or harmful [NWT+07]. The drawbacks with dynamic approaches is that they only work on closed programs which can be executed, they require tests that sufficiently exercise the code, and that ultimately, they cannot be used to classify all potential accesses.

**Static Datarace Detection for Java.** Java's native support for multithreading coupled with its restricted use of syntactically scoped locks has given rise to a variety of static techniques for detecting and proving the absence of races in Java code. Early work includes the development of type systems that encode a static lockset analysis [FA99, FF00]. These type based approaches were made more expressive by incorporating a notion of *ownership* [BLR02]. Similar type systems were designed to ensure race-freedom in Cyclone [Gro03]. While these type systems are eminently scalable, they require user annotation, though there has been some work on using SAT solvers [FF04b] and dynamic locksets [ASWS05] to infer the lock annotations. Another line of work is that of [vPG03] which finds races by computing an *Object Use Graph* that statically approximates the dynamic happens-before relation. A recent line of work [NAW06] shows how to effectively use cloning-based context-sensitivity to drastically improve the precision of lockset computations. The approach was further refined in [NA07] by using a notion of must-not aliasing to prune the set of warnings. The above techniques exploit key properties of Java – namely the scoped use of locks, which mitigates the need for flow-sensitivity. Thus, while they are not directly applicable to our setting, we believe that it may be possible to apply ideas like ownership and must-not aliasing to lower the false positives that arise due to initialization and unlikely aliasing respectively.

**Static Datarace Detection for C.** Analyses devised for finding races in C programs must cope with several additional problems. Principal among them is the use of unstructured locks, which force the analysis to be flow- and context- sensitive. The only approach we know of that has scaled to millions of lines is RacerX, which also finds deadlocks, and runs over large code bases in minutes. Unlike Relay, RacerX uses a top-down approach to computing the locksets at each program point. The paper reports that in order to scale, several drastic compromises had

to be made, such as truncating the summaries and representing all lvalues with their types. As a result, the analysis discards valuable information prematurely, discarding possible races well before the warning generation phase. Consequently, the tool was only able to unearth a small handful of warnings and an order of magnitude fewer races. A more precise approach is that of LOCKSMITH [PFH06] which uses a constraint based technique to compute *correlations* that describe the locks that protect an lvalue. While this approach is as precise as ours, it has only been applied to programs two orders of magnitude smaller than the Linux kernel. We conjecture that the principle bottleneck is the difficult task of solving a monolithic set of constraints generated over millions of lines of code. This is in contrast to RELAY whose algorithm is modular and readily parallelizable. LP-RACE [Ter08] is a race detector designed after RELAY that is based on a fractional capability system. Threads are modeled as holding capabilities to read or write shared locations, and synchronization events transfer capabilities between threads. Capability mappings are inferred by a linear programming (LP) solver for each abstract location. In principle the approach is scalable since each LP instance can be parallelized, but the implementation has only been tested on programs an order of magnitude smaller than RELAY. The main benefit of this formalization is that it is a unified system for handling locks as well as non-lock-based synchronization such as condition variables and thread joins. However, initialization, thread local accesses and re-entrant locks (problems in RELAY) are also problems for LP-RACE.

Finally, heavyweight techniques such as model checking [HJM04, QW04] have been applied to find and prove the absence of races. These techniques are essential in situations where the synchronization is not lock-based, but instead is via exotic mechanisms like state variables, interrupt disabling, or the idioms described in Section 3.5. It is unclear whether such heavyweight methods can be scaled to large code bases.

Finally, while others have designed bottom-up analyses using complete summaries [CmWH00, DLS02], our work, and the notion of parallelization is directly inspired by the approach taken by [HA06].

## 6.2 Dataflow Analysis for Concurrent Programs

**Frameworks for Dataflow Analysis.** There are frameworks for dataflow analysis of concurrent programs, limited to the use of nested ParBegin and ParEnd constructs. These frameworks work by building a parallel flow graph (PFG) (a control flow graph with parallel sections). The dataflow analysis is lifted by extending the flow equations to handle the fork and join of the parallel sections. Examples include a reaching definitions analysis [GS93] and bit-vector analyses [KSV96], and a pointer analysis [RR99]. Other representation-based approaches include [Sar97] which uses ParBegin/ParEnd and wait/notify constructs to build a Parallel Program Graph, analogous to the PDG, and [LPM99] which describes a Concurrent SSA (CSSA) representation which enables subsequent optimizations. The causal dataflow analysis framework of [FM07] handle more forms of synchronization including locks. Their approach is based on checking coverability [ERV96] of a Petri net representation of the program and its dataflow facts. The approach is limited to recursion-free programs with bounded numbers of threads and finite dataflow domains. Recently, [CR06] proposed an analysis framework for optimizing embedded programs written in NESC [GLvB+03]. This framework is tailored to NESC 's interrupt-based concurrency and explicit `atomic` sections.

All the above frameworks are more precise than our framework in which facts can only be *killed* by concurrent interactions. In contrast, these frameworks exploit specific concurrency constructs to also allow new facts to be *generated* by concurrent interactions. However, RADAR is more general, as it is independent of the underlying concurrency constructs, requiring only that a race detector exists for the constructs. For example, none of these approaches could be applied to our thread-based benchmarks.

**Leveraging Sequential Analyses with a Context Bound.** For the purpose of bug-finding, recent work has shown that concurrency bugs can be demonstrated with a small number of context switches. KISS [QW04] converts a concurrent program $P$ to a sequential program $P^s$ that simulates $P$ restricted to two context switches. This allows sequential analyses to run on $P^s$ while having the results

apply to $P$. The benefit is that it avoids the exponential complexity of exhaustively analyzing all thread interleavings. [LR08] provide a translation which allows all thread schedules with arbitrary context bound of $k$ to be analyzed using a single round-robin thread schedule.

The main issue is that while such transformations allow the reuse of sequential analyses, context bounding is unsound, and thus not applicable to compiler optimizations. Furthermore, requirements on the sequential analysis used are heavier – it is crucial that the analysis handle branch conditions/`ASSUME` statements and symbolic constants. Finally, though sequential analyses used along with this approach are more precise, they are still limited in scalability.

**Model Checking.** Model checkers explore all interleavings to verify arbitrary safety properties, and so they can be used to encode dataflow analyses [Sch98]. Flavers [CCO02] is a finite-state property checker that employs conservative state and interleaving reductions, *e.g.* a may-happen-in-parallel analysis ([NAC99]) that conservatively prunes interleavings. Even with techniques like these and others like partial-order and symmetry reduction that mitigate the effect of combinatorial explosion in interleavings, model checking has only been shown to scale to relatively small code bases. A technique related to Radar is the *thread-modular* approach, proposed in [OG76, Jon83] which requires that users provide annotations describing when *other* threads can modify shared state. Model checking can be used to infer the annotations [FQ03, HJM04], but these techniques do not scale. If the programs include recursive procedures, model checking (and hence, "exact" dataflow analysis in the sense of computing MOP solutions) is undecidable [Ram00].

In contrast to the above, the principal benefit of our framework Radar is that it is not tied to any particular concurrency constructs or structure, as all reasoning about concurrency is folded into the race detection engine. This allows Radar to switch between race detection engines to explore the tradeoff between precision and scalability of the dataflow analysis. Moreover, Radar enables a finer view of concurrency by preserving facts that are not killed by other threads, without exploring interleavings caused by irrelevant atomicity breaks as in Figure 2.1.

# Chapter 7

# Conclusions and Future Work

We have presented a framework called RADAR for converting a sequential dataflow analysis into a concurrent one using a race detection engine as a black box. The main benefit of this approach is that it cleanly separates the part of the analysis that deals with concurrency, the race detection engine, from the rest of the analysis. With this separation in place, the race detection engine can be fine-tuned to improve its precision without changing any of the client analyses. As a result, RADAR provides a framework that allows the precision with which concurrency is analyzed to be easily tuned.

Paired with the RELAY data-race detection engine, our experiments show that the framework scales, and for several dataflow analyses, achieves good precision with respect to some upper and lower bounds.

Our experience also shows that there is still room for improvement in terms of the precision of the overall concurrent analysis and its running time. All components of the analyses play an important role: the alias and escape analysis, the race detector, and the sequential analyses can all affect precision. The call-graph analysis can drastically affect scalability. We have identified several lines of future work that will, in combination, lead to understanding and addressing these issues.

## 7.1 Experience and Idioms

**Adjust Additional Analyses.** The first direction is to apply the adjusting approach to lift additional previously developed sequential analyses to the concurrent setting. This direction of future research may include discovering what is required to adjust *other forms* of analyses besides dataflow analyses. Examples include array bounds checking analyses [VB04, HDWY06b], and other kinds of null-dereference analyses [DDA07, BH08]. It may also be worthwhile to push the RADAR versions of compiler-assisting analyses (*e.g.* in section 5.2) through an optimizer to measure actual performance impact.

A wide variety of RADAR-converted analyses will allow us to tune the precision of the race detector using actual facts deduced while analyzing real systems for a variety of properties. The generated concurrent analyses can lead to an empirical understanding of the concurrency idioms used in real programs. These patterns can then be used to iteratively tune the precision of parts of the framework.

## 7.2 Precision and Performance

**On-Demand Refinement of Concurrency Analysis.** The second direction is to explore precise and scalable race detection techniques for other concurrency constructs. One starter possibility is to take the successive refinement approach with two race detection engines – one scalable engine like RELAY, which handles lock-based synchronization, and one precise engine like [HJM04], which handles more general forms of synchronization including flag-based synchronization. First, run the scalable engine and take each warning (two data accesses) as a seed for slicing the program. The hope is that at least some slices will be small enough for the precise engine to confirm or refute the warning, within a given time budget. A side benefit is that a precise race detection engine based on model checking can produce concrete error traces [KSG09].

Other obvious areas to improve include the alias and escape analysis, and

perhaps a lightweight may-happen-in-parallel analysis to handle "static" synchronization mechanisms `spawn` and `join`.

**Annotations.** If it proves too difficult to automatically and statically improve the precision of the relevant modules, one possibly is to have the programmer supply annotations. These annotations should be checked statically, but may still provide valuable hints. [AGEB08], for example, suggests that annotations about how data is *shared* between threads are valuable. Many of their annotations are checked statically, but some are checked at run-time (similar to dynamic casts). Although annotations add human effort, the burden can be lowered if done at a higher-level – *e.g.* annotations for all instances of particular types or over entire modules. Such higher-level annotations can serve as documentation as well.

**Runtime Support.** Besides annotations, it may be useful for the runtime or hardware to provide assistance to Radar. For example, if there existed hardware that was capable of avoiding certain interleavings, would static analyses be more precise or more tractable? An example of increased tractability include analyses which are only sound up to a context-bound [QR05]. Of course, this exact assumption is unlikely to be guaranteed by the runtime system but perhaps finer-grain reductions in interleavings will be useful. For example, one can instrument the program with additional synchronization or use work such as [YN09], which develop hardware with the goal of avoiding interleavings not seen during testing.

**Modularity and Separate Compilation.** Finally, while compiler optimization-oriented dataflow analyses converted by Radar are more precise than those relying on more conservative assumptions, they are not modular. Radar involves a whole-program analysis (the race detector). Code generated for a function $f$ using a Radar-adjusted constant propagation analysis may now be sensitive to racy writes in other functions and compilation units. Again, runtime schemes ([RRRV09]) may be of assistance, ensuring isolation between modules. However, they suffer from certain drawbacks including the introduction of deadlock. Other methods of re-enabling separate compilation, while retaining precision, may be interesting.

# Appendix A

# Properties of Relative Dataflow Analyses

This chapter contains proofs that the flow functions presented for our relative dataflow analyses (*e.g.* the lockset analysis in RELAY, and the non-null client for RADAR) have the two classical properties of dataflow analyses: monotonicity and distributivity. Distributivity actually implies monotonicity, but we show proofs for each individual property.

**Abstract Relative Analysis.** In order for the proofs to work for a general relative dataflow analysis (and not just the lockset analysis, or the non-null analysis in particular), we begin with an abstraction of relative dataflow analyses and do proofs on this abstraction. First, dataflow facts are restricted to be pairs of sets with one set being the set of facts that must have become true since the beginning of the current procedure and the other a may set for the negation of the must facts. Any fact not mentioned in either set is preserved by the current procedure. Next, how the flow functions transform the input can be described by two sets. One set, which is denoted here as $S_+$, dictates what should be added to the must-sets and removed from the may-sets. The other set, $S_-$, dictates what should be added to the may-sets and removed from the must-sets. They are like GEN and KILL sets, but they swap roles between the relative must-sets and the may-sets. These sets $(S_+, S_-)$ may come from function summaries, or be based on the other kinds of

statements (assignments and branches). Thus, every flow function is parameterized by some pair $(S_+, S_-)$ and has the following shape:

$$F_{S_+,S_-}(A_+, A_-) = ((A_+ \cup S_+) - S_-, (A_- \cup S_-) - S_+)$$

Aside from fixing the form of flow functions, we also assume the lattice relations $(\sqcup, \sqsubseteq)$ to be:

$$\begin{aligned} (A_+, A_-) \sqcup (B_+, B_-) &= (A_+ \cap B_+, A_- \cup B_-) \\ (A_+, A_-) \sqsubseteq (B_+, B_-) &\text{ iff } A_+ \subseteq B_+ \text{ and } B_- \subseteq A_- \end{aligned}$$

The proofs below simply rely on algebraic properties (distributivity of set intersection w.r.t. set union), so similar proofs would apply if $\cap$ and $\cup$ were swapped in the above relations.

**Monotonicity.** Now we can show that if $A \sqsubseteq B$, then $F_{S_+,S_-}(A) \sqsubseteq F_{S_+,S_-}(B)$. Assuming the hypothesis, compare each set in the pairs $F_{S_+,S_-}(A)$ and $F_{S_+,S_-}(B)$ pointwise.

For the positive sets, we know $A_+ \subseteq B_+$. Thus, $(A_+ \cup S_+) \subseteq (B_+ \cup S_+)$ for any set $S_+$. This implies that (1) $(A_+ \cup S_+) - S_- \subseteq (B_+ \cup S_+) - S_-$ for any set $S_+$ and set $S_-$.

For the negative sets, we know $B_- \subseteq A_-$. Thus, $(B_- \cup S_-) \subseteq (A_- \cup S_-)$, and we can conclude that (2) $(B_- \cup S_-) - S_+) \subseteq (A_- \cup S_-) - S_+)$.

Given (1) and (2) and by the definition of $\sqsubseteq$, we can conclude that if $A \sqsubseteq B$, then $F_{S_+,S_-}(A) \sqsubseteq F_{S_+,S_-}(B)$.

**Distributivity.** To show that $F_{S_+,S_-}$ is distributive, we must show that $F_{S_+,S_-}(A \sqcup B) = F_{S_+,S_-}(A) \sqcup F_{S_+,S_-}(B)$. This can be shown with some basic set algebra:

$$
\begin{aligned}
&F_{S_+,S_-}((A_+,A_-)\sqcup(B_+,B_-))\\
&=F_{S_+,S_-}(A_+\cap B_+,A_-\cup B_-)\\
&=(((A_+\cap B_+)\cup S_+)-S_-,((A_-\cup B_-)\cup S_-)-S_+)\\
&=(((A_+\cup S_+)\cap(B_+\cup S_+))-S_-,((A_-\cup S_-)\cup(B_-\cup S_-))-S_+)\\
&=(((A_+\cup S_+)\cap(B_+\cup S_+))\cap S_-^c,((A_-\cup S_-)\cup(B_-\cup S_-))\cap S_+^c)\\
&=(((A_+\cup S_+)\cap(B_+\cup S_+))\cap S_-^c\cap S_-^c,((A_-\cup S_-)\cup(B_-\cup S_-))\cap S_+^c)\\
&=(((A_+\cup S_+)\cap S_-^c)\cap((B_+\cup S_+)\cap S_-^c),((A_-\cup S_-)\cup(B_-\cup S_-))\cap S_+^c)\\
&=(((A_+\cup S_+)\cap S_-^c)\cap((B_+\cup S_+)\cap S_-^c),((A_-\cup S_-)\cap S_+^c)\cup((B_-\cup S_-)\cap S_+^c))\\
&=F_{S_+,S_-}((A_+,A_-))\sqcup F_{S_+,S_-}((B_+,B_-)).\quad\square
\end{aligned}
$$

# Appendix B

# Function Pointer Slicing for WL

Chapter 4 compares several callgraph construction algorithms, including one based on a context-sensitive, flow-sensitive, and field-sensitive pointer analysis – a version of the Wilson-Lam algorithm (WL) restricted to function values. Here we describe the approach to slicing out non-function values. The approach is based on possibly unsound assumptions, but it is able to detect or correct when the assumptions may be unsound for a given input program.

## B.1 Evaluating Benefits of Slicing

It is believed that the WL algorithm on its own is not scalable. By only track pointers that transitively reach function pointers it is possible to analyze large, hundred thousand-line programs. Figure B.1 compares time and memory usage with and without this form of slicing. Without slicing, two of the benchmarks are unable to complete within a 6 hour time budget (Vim and Nethack) and two others take at least 4 times longer to complete (Mutt and Git). For the benchmarks over 60 KLOC, memory usage is also noticeably higher.

## B.2 Difficulties in Slicing

Because of the weak C type system, casts, the use of pointer arithmetic, the ability to embed structs within structs, and the ability for a pointer to reference

| Benchmark | KLOC | #fp-calls | Time (s) | | Mem (MB) | |
|---|---|---|---|---|---|---|
| | | | sliced | all-ptr | sliced | all-ptr |
| Bzip21.0.5 | 12 | 20 | 4.8 | 5.4 | 15 | 11 |
| Icecast2.3.2 | 28 | 199 | 232.3 | 391.0 | 62 | 71 |
| Unzip6.0 | 31 | 381 | 59.3 | 66.1 | 45 | 35 |
| OpenSSH5.2 | 56 | 141 | 43.3 | 48.3 | 57 | 50 |
| smtpd2.6.2 | 67 | 337 | 6042.1 | 7582.4 | 159 | 227 |
| Mutt1.5.9 | 83 | 743 | 213.2 | 11586.8 | 82 | 204 |
| Git1.3.3 | 171 | 91 | 446.2 | 1816.9 | 170 | 278 |
| Vim7.2 | 230 | 27 | 5371.3 | T-O | 281 | 490 |
| Emacs22.3 | 272 | 288 | 3000 | 3549 | 281 | 367 |
| Nethack3.4.3 | 284 | 863 | 1655.9 | T-O | 287 | 2147 |

Figure B.1: Time and memory usage of WL with and without slicing

arbitrary fields within structures, slicing out non-function-pointer-related pointers takes some care.

Typically the concern with C casts, structures, and pointer arithmetic is with the soundness of a field-sensitive pointer analysis [YHR99, Ste96b]. We are instead concerned with the problem of slicing without relying on a dependence analysis, which would essentially involve another pointer analysis.

One attempt at a lightweight approach to slicing-out pointers unrelated to function pointers is to simply look at type declarations. Figure B.2 illustrates some of the problems with such an approach, given the C type system. In the figure, there are only three user-defined structs `string_t`, `dlist_t`, and `obj_t`. Given a pointer to each type of structure, can such a pointer be used to access a function pointer? The programmer-intended answers are "no", "yes", and "yes". That `obj_t*` can be used to access a function pointer is obvious.

**Casts.** For `string_t*` to not reach a function pointer (get a "no" answer), an analysis must trust that the pointer field `char*` is not cast to/from pointer types that do reach function addresses.

**Embedding Structs and Pointer Arithmetic.** To notice that `dlist_t*` does reach a function address (get a "yes" answer), an analysis must notice that two

things. First, the pointer may reference a `dlist_t` that is *embedded* within an `obj_t` struct (the `otherObjs` field). Second, that with pointer arithmetic the container `obj_t` and its function pointer field can be accessed. Lines 2 and 3 in the figure show how this is possible. However, if the analysis is not careful, it may think that a pointer to `string_t` may also be a pointer to an embedded string (the `id` field) and that the pointer can be used to access the container `obj_t` structure. Therefore, simply looking at the types and how they are embedded within each other can mean that almost no pointers are ever sliced out of the analysis. The analysis must look at what pointer arithmetic expressions are actually present in the code as well.

## B.3 Approach to Slicing

Callgraph construction with WL involves two phases. First, a pointer analysis computes a mapping from memory locations which represent pointers to targets (which are other memory locations including function addresses). Second, the mapping is consulted to generate the actual callgraph. The goal is to reduce the time spent in the first phase, as it is the dominant phase.

The core of pointer analysis involves analyzing sets of assignments ($l := e$) which are either explicit in the program (or implicit from parameter passing and function returns). This analysis builds a map from lvalues (pointers) to sets of lvalues and functions (targets). Rather than perform program slicing based on program dependencies for indirect calls to reduce the size of the input program, this approach "filters-out" assignments by flowing a non-function pointer address, *Nfp*, in place of the pointer value of the expression on the right side which would have been flowed without slicing. Replacing concrete pointer sets with *Nfp* speeds up convergence of the dataflow analysis and reduces memory usage.

**Slicing with Assumptions and Coping With Deviance.** Our technique for slicing is based on certain assumptions (*e.g.* only certain forms of pointer arithmetic are present) and it is possible that this approach is unsound. This form of slicing allows us to cope with violated assumptions and actually ensure soundness. If

```
typedef struct string {
   int len; char *buff;
} string_t;

typedef struct dlist {
  struct dlist *prev; struct dlist *next;
} dlist_t;

typedef struct obj {
  void (*fp)(char *);
  dlist_t otherObjs;
  string_t id;
} obj_t;

#define offsetof(T, f) \
1:  ((size_t) ( (char *)&((T *)(0))->f - (char *)0 ))

void test() {
  dlist_t *head, *l;
  obj_t *o;
  head = allocate_list();
  //...
  for (l = head->next; l != head; l = l->next) {
2:  o = (obj *)((char *)l - offsetof(obj_t, otherObjs));
3:  o->fp(o->id.buff);
  }
}
```

Figure B.2: Example: Embedded structs complicate slicing

in the second phase of callgraph construction, the pointer analysis results map a function pointer to *Nfp* it is possible to *correct* this unsoundness by using a second pointer analysis (such as Steens-FI which does not rely on slicing but is scalable).

Unsoundness can also be *detected* during the pointer analysis when *Nfp* values are encountered. Inevitably, the pointers with *Nfp* will be dereferenced as part of other assignment statements. In cases where the dereference is on the rhs of an assignment, we simply flow another *Nfp*, as *Nfp* pointers should only reach *Nfp* values. If this is not true it is caught during the callgraph construction phase

as mentioned above. In cases where the dereference is on the lhs of an assignment, the rhs must be an *Nfp* value. If it is, then the write can be ignored, as the actual target should not relevant. Otherwise it means that a function pointer related value is to be written to a location, but the possible locations were not tracked. In this case, the analysis can issue a warning to reduce the strictness of slicing or rewrite the source to match the assumptions used for slicing. For example, one case in which the source may need to be written is when `char*` is used as a polymorphic pointer instead of `void*`. Thus, this approach of using *Nfp* instead of slicing the original code allows us to detect and/or compensate for unsoundness.

**When to Generate *Nfp* Values.** Assignments $l := e$ either propagate existing pointer information from $e$, or are base assignments where $e$ is of the form `&x`, "`strLiteral`", or calls to malloc. String literals can be filtered out with *Nfp*. For the case of `&x` we know the concrete C type of the target. For the case of dynamically allocated malloc cells, the concrete C type of the target can also be known, assuming that the input C program has been transformed appropriately:

- malloc is called with a single `sizeof(T)` argument. A common counterexample involves the case where multiple blocks of memory may be allocated in one shot (*e.g.* `sizeof(T1)` + `sizeof(T2)`). This is done to simplify checks for out-of-memory situations and ensure memory locality, among other things. Such calls to malloc are rewritten into individual calls.

- The `sizeof(T)` argument to malloc must accurately reflect the intended use of the allocated memory. A counter-example is custom allocators, which allocate large blocks of memory and hand out sub-blocks at a later point. Calls to custom allocators are transformed into calls to malloc.

Given a base assignment with the C type of a location, we estimate whether or not the location will ever reach a function pointer through a sequence of dereferences, field accesses, and pointer arithmetic operations. If it does not, then the analysis flows an *Nfp* value in the place of its address. If it does, then the address is used in the pointer analysis. Note that although the C type of the initial location is known, because base assignments have been transformed to make them

known, further locations that are reachable from accesses to pointer fields or dereferences may not be precise. Slicing relies on a reachability analysis that uses type declarations of structure fields as well as the following assumptions:

- Pointer arithmetic can be used to stride along an array (*e.g.* `p + 1`). However, strides along arrays are ignored since arrays are modeled as a single summary node. The expression `p + 1` can also be used to reference memory which follows a block of memory in the case that multiple blocks of memory, each of different type, are allocated in a single call to malloc. As noted earlier, such forms of allocation are assumed to be rewritten.

- Pointer arithmetic can be of the form (`byte*`)`p + c` where `c` is a constant expression based on *recognizable* patterns that give the offset of a structure field (for example, line 1 of figure B.2). These patterns relate two types by a constant, $D(T1, T2, c)$. In other words, all other pointer arithmetic is on the byte-level and is used transform pointers into a struct to pointers to the beginning of a struct.

- A related assumption is that one may only access the fields of a structure `T1` given a pointer to `T2`, if (a) `T2` is embedded at the beginning of `T1`, or (b) there exists a constant such that $D(T1, T2, c)$.

- Pointer fields declared as `T2*` are assumed to point to a location of type `T2`, with the possibility that it is embedded in another struct. This allows structural subtyping, where the pointer `T2*` can be used to access fields of a structure of type `T1` if `T2` is embedded as a prefix of `T1` (due to case (a) of the above assumption).

- Pointer fields which are *polymorphic* and do not follow structural subtyping must be declared `T*` where `T` is an empty structure such as `void` (rather than `char*`). Pointers to empty structs are assumed to be opaque and are conservatively assumed to reach any type including function pointers.

- Pointers which are cast to numeric types are either (1) only done so temporarily (stored as a local or function parameter) and so are not stored in a

structure field, or (2) may be stored in a structure field but never cast back into a pointer (*e.g.* if an address is used as a hash value).

Given those assumptions, the types declared in the program, and a scan of pointer arithmetic expressions in the program to construct the relation $D$, testing reachability from a C type to a function pointer is straightforward.

# Bibliography


[Age95] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP*, pages 2–26, 1995.

[AGEB08] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. Sharc: checking data sharing strategies for multithreaded c. *SIGPLAN Not.*, 43(6):149–158, 2008.

[And94] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[ASWS05] R. Agarwal, A. Sasturkar, L. Wang, and S.D. Stoller. Optimized runtime race detection and atomicity checking using partial discovered types. In *ASE 05: Automated Software Engineering*, pages 233–242, 2005.

[Atk04] Darren C. Atkinson. Accurate call graph extraction of programs with function pointers using type signatures. In *APSEC*, pages 326–335, 2004.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI 03: Programming Languages Design and Implementation*, pages 196–207. ACM, 2003.

[BH08] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In *ICSE '08: International conference on Software engineering*, pages 211–220, New York, NY, USA, 2008. ACM.

[BLQ+03] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie J. Hendren, and Navindra Umanee. Points-to analysis using bdds. In *PLDI*, pages 103–114, 2003.

[BLR02] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA 02: Object-Oriented Programming, Systems, Languages and Applications*, pages 211–230, 2002.

[BR02] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.

[CCO02] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. FLAVERS: A finite state verification technique for software systems. *IBM Systems Journal*, 41(1):140–165, 2002.

[CDC+04] Rezaul Alam Chowdhury, Peter Djeu, Brendon Cahoon, James H. Burrill, and Kathryn S. McKinley. The limits of alias analysis for scalar optimizations. In *CC*, pages 24–38, 2004.

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[CLL+02] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multi-threaded object-oriented programs. In *PLDI 2002: Programming Languages Design and Implementation*, pages 258–269. ACM, 2002.

[CmWH00] Ben-Chung Cheng and Wen mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI*, pages 57–69, 2000.

[CR06] N. Cooprider and J. Regehr. Pluggable abstract domains for analyzing embedded software. In *LCTES*, pages 44–53, 2006.

[CRL99] Ramkrishna Chatterjee, Barbara G. Ryder, and William Landi. Relevant context inference. In *POPL*, pages 133–146, 1999.

[CVJL08] Ravi Chugh, Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using datarace detection. In *PLDI*, pages 316–326, 2008.

[DDA07] Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. *SIGPLAN Not.*, 42(6):435–445, 2007.

[Det] Dave Detlefs. Array bounds check elimination in the clr. `http://blogs.msdn.com/clrcodegeneration/archive/2009/08/13/array-bounds-check-elimination-in-the-clr.aspx`.

[DLFR01] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. *Lecture Notes in Computer Science*, 2126:260–??, 2001.

[DLS02] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68. ACM, 2002.

[DRS03] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. *SIGPLAN Not.*, 38(5):155–167, 2003.

[DS91] Annette Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991.

[EA03] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *SOSP 03: ACM Symposium on Operating System Principles*, pages 237–252. ACM Press, 2003.

[ERV96] Javier Esparza, Stefan Rmer, and Walter Vogler. An improvement of mcmillan's unfolding algorithm. In *Formal Methods in System Design*, pages 87–106. Springer-Verlag, 1996.

[FA99] C. Flanagan and M. Abadi. Types for safe locking. In *ESOP 99: European Symposium on Programming*, LNCS 1576, pages 91–108. Springer, 1999.

[FF00] C. Flanagan and S.N. Freund. Type-based race detection for Java. In *PLDI 00: Programming Languages Design and Implementation*, pages 219–232. ACM, 2000.

[FF04a] Cormac Flanagan and Stephen N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, 2004.

[FF04b] Cormac Flanagan and Stephen N. Freund. Type inference against races. In *SAS*, pages 116–132, 2004.

[FF09] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, pages 121–133, New York, NY, USA, 2009. ACM.

[FFSA98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*, pages 85–96, New York, NY, USA, 1998. ACM.

[FM07] Azadeh Farzan and P. Madhusudan. Causal dataflow analysis for concurrent programs. In *TACAS*, pages 102–116, 2007.

[FQ03] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN 03: Model Checking Software*, LNCS 2648, pages 213–224. Springer, 2003.

[GC01] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

[GLvB+03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI 2003: Programming Languages Design and Implementation*, pages 1–11. ACM, 2003.

[Gro03] Dan Grossman. Type-safe multithreading in cyclone. In *TLDI*, pages 13–25, 2003.

[GS93] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *PPoPP*, San Diego, CA, 1993.

[Gup93] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2:135–150, 1993.

[HA06] Brian Hackett and Alex Aiken. How is aliasing used in systems software ? In *FSE 2006: Foundations of Software Engineering*, pages 69–80. ACM, 2006.

[HDWY06a] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE*, pages 232–241, 2006.

[HDWY06b] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE*, pages 129–144. ACM, 2006.

[HJM04] T.A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, pages 1–12. ACM, 2004.

[HL07a] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299, 2007.

[HL07b] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *SAS*, pages 265–280, 2007.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *PLDI*, pages 254–263, 2001.

[IH97] Marc Shapiro II and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *POPL*, pages 1–14, 1997.

[Jon83] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[KRS92] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *SIGPLAN Not.*, 27(7):224–234, 1992.

[KSG09] Vineet Kahlon, Sriram Sankaranarayanan, and Aarti Gupta. Semantic reduction of thread interleavings in concurrent programs. In *TACAS '09: Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–138, Berlin, Heidelberg, 2009. Springer-Verlag.

[KSV96] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *TOPLAS*, 18(3):268–299, May 1996.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.

[KYSG07] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV*, pages 226–239, 2007.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lip75] R.J. Lipton. Reduction: A new method of proving properties of systems of processes. In *POPL*, pages 78–86, 1975.

[LLA07] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *PLDI*, San Diego, California, June 2007.

[LPM99] J. Lee, D.A. Padua, and S.P. Midkiff. Basic compiler algorithms for parallel programs. In *PPOPP*, pages 1–12, 1999.

[LR08] Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV '08: Computer Aided Verification*, pages 37–51, Berlin, Heidelberg, 2008. Springer-Verlag.

[MACE02] Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. *SIGSOFT Softw. Eng. Notes*, 27(6):71–80, 2002.

[MC93] John M. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Workshop on Parallel and Distributed Debugging*, pages 129–139, 1993.

[MMN09] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *PLDI '09*, pages 134–143, New York, NY, USA, 2009. ACM.

[MRR04] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engineering*, 11(1):7–26, 2004.

[NA07] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *POPL*, pages 327–338, 2007.

[NAC99] G. Naumovich, G.S. Avrunin, and L.A. Clarke. An efficient algorithm for computing *mhp* information for concurrent java programs. In *ESEC / SIGSOFT FSE*, pages 338–354, 1999.

[NAW06] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, pages 308–319, 2006.

[NMRW02] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC 02: Compiler Construction*, Lecture Notes in Computer Science 2304, pages 213–228. Springer, 2002.

[NWT+07] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, pages 22–31, 2007.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[PFH06] P. Pratikakis, J.S. Foster, and M.W. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI*, pages 320–331, 2006.

[PKH04a] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *PASTE*. ACM Press, 2004.

[PKH04b] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Control*, 12(4):311–337, 2004.

[Pra08] Polyvios Pratikakis. *Sound, Precise and Efficient Static Race Detection for Multithreaded Programs*. PhD thesis, University of Maryland, College Park, 2008.

[PS07] Eli Pozniansky and Assaf Schuster. Multirace: efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, 2007.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *In TACAS*, pages 93–107. Springer, 2005.

[QW04] Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *TOPLAS*, 22(2):416–430, 2000.

[RC00] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *PLDI*, pages 47–56, 2000.

[RR99] R. Rugina and M.C. Rinard. Pointer analysis for multithreaded programs. In *PLDI*, pages 77–90, 1999.

[RRRV09] Sriram Rajamani, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Isolator: dynamically ensuring isolation in comcurrent programs. In *ASPLOS: Architectural Support for Programming Languages and Operating Systems*, pages 181–192, New York, NY, USA, 2009. ACM.

[Sar97] V. Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *LCPC*, pages 94–113, 1997.

[SBN+97] S. Savage, M. Burrows, C.G. Nelson, P. Sobalvarro, and T.A. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[Sch98] D.A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *POPL 98: Principles of Programming Languages*, pages 38–48. ACM, 1998.

[Shi88] Olin Shivers. Control-flow analysis in scheme. In *PLDI*, pages 164–174, 1988.

[Sim09] Luke Simon. Optimizing pointer analysis using bisimilarity. In *SAS*, 2009.

[SR01] A. Salcianu and M.C. Rinard. Pointer and escape analysis for multithreaded programs. In *PPOPP*, pages 12–23, 2001.

[Ste93] N. Sterling. Warlock: a static data race analysis tool. In *USENIX Winter 1993 Technical Conference*, pages 97–106, 1993.

[Ste96a] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[Ste96b] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. pages 136–150. Springer-Verlag, 1996.

[Ter08] Tachio Terauchi. Checking race freedom via linear programming. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–10, New York, NY, USA, 2008. ACM.

[TP00] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA*, pages 281–293, 2000.

[VB04] A. Venet and G.P. Brat. Precise and efficient static array bound checking for large embedded c programs. In *PLDI*, pages 231–242, 2004.

[VJL07] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: Static race detection on millions of lines of code. In *ESEC/FSE*. ACM, 2007.

[vPG03] C. von Praun and T.R. Gross. Static conflict analysis for multithreaded object-oriented programs. In *PLDI*, pages 115–128, 2003.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *In Network and Distributed System Security Symposium*, pages 3–17, 2000.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI*, pages 1–12, 1995.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

[WS06] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Software Eng.*, 32(2):93–110, 2006.

[WWM07] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the java hotspot™client compiler. In *PPPJ '07: Principles and practice of programming in Java*, pages 125–133, New York, NY, USA, 2007. ACM.

[XA05] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, pages 351–363. ACM, 2005.

[XA06] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[YHR99] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. In *PLDI*, pages 91–103, 1999.

[YN09] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA '09: International Symposium on Computer architecture*, pages 325–336, New York, NY, USA, 2009. ACM.

[YRC05] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2005.

[ZC04] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *PLDI*, pages 145–157, 2004.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *FSE*, pages 81–92, 1996.