Cautiously Optimistic Program Analyses for Secure and Reliable Software

by

Subarno Banerjee

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2021

Doctoral Committee:

Professor Satish Narayanasamy, Chair
Assistant Professor Jean-Baptiste Jeannin
Professor Karem A. Sakallah
Assistant Professor Xinyu Wang

Subarno Banerjee
subarno@umich.edu
ORCID iD: 0000-0001-5449-2264

© Subarno Banerjee 2021

And I say that life is indeed darkness save when there is urge,
And all urge is blind save when there is knowledge,
And all knowledge is vain save when there is work,
And all work is empty save when there is love.

– *The Prophet*, by *Kahlil Gibran*, 1923

To my parents and teachers

ACKNOWLEDGMENTS

This dissertation work is the product of many different contributions from multiple people. I'm grateful, fortunate and often humbled by their generosity.

My advisor Satish Narayanasamy took a chance on me, has shepherded me throughout my graduate career, and inspired me to grow as a researcher. He has taught me how to identify worthy problems, think critically, and organize and present ideas better. He has been a steady source of feedback and reasoned advice. I owe this PhD to him, and am eternally grateful to him.

I'm grateful to my collaborators Peter Chen and David Devecsery who have made significant contributions in developing this body of work. Peter discusses ideas with an unmatched energy and enthusiasm, shares advice with a genuine desire to see students succeed, and is very generous with his time; he is a great advisor. David has an impressive expertise on nearly all topics on computer systems, and has helped me learn and develop the tools and techniques used in this work; I hope to be half as good as him someday. I'm grateful to Jean-Baptiste Jeannin, Karem Sakallah, and Xinyu Wang for serving on my dissertation committee. Evaluating this dissertation and attending exams was quite the time investment, and their insightful comments has helped improve this dissertation. The UM-CSE department provided a stimulating learning environment and valuable resources. I'm thankful to the brilliant CSE faculty and all staff members.

I was fortunate to work with some amazing people during a few internships, and I'm thankful to all my hosts and colleagues: Manu Sridharan and Lazaro Clapp at Uber, Weidong Cui and Xinyang Ge at Microsoft, and Dragos Sbirlea at Google. I'm especially grateful to Manu for his collaboration on the NullAway paper, and for his continued mentorship and help during the job search process. I'm also extremely grateful to Gogul Balakrishnan at Google for being an excellent mentor and sharing his valuable insights and career advice.

I would like to particularly thank Shaizeen Aga, my labmate and mentor, who served as my lifeline during grad school. She helped me manage expectations, navigate advisor relationships, and deal with paper rejections. My success in grad school is thanks to her. I'm fortunate to have the generous support of our lab alumni Ram Srivatsa Kannan, Gaurav Chadha, Abhayendra Singh and Animesh Jain. I'm also thankful to Swagath Venkataramani for his advice at different times.

I had the pleasure of working with Yirui Liu on the OPT-SC project; I thank her for the collaboration. And, I'd like to thank fellow CELAB students and my labmates for the many engaging discussions, reading groups, and social events. Special thanks to Sanjay Singapuram for many interesting conversations and for his help in proofreading this document.

I was fortunate to have excellent teachers who inspired me to pursue a research career. I'm thankful to Mainak Chaudhuri for advising my master's thesis work, and I doubt that I would have pursued a PhD but for his support and encouragement. I also owe this journey to my college professors and school teachers; I could only reach this point in life because of them.

I feel obliged to acknowledge the influence of some notable authors and speakers in informing my general outlook: Sam Harris, Peter Singer, Vaclav Smil, and Shankar Vedantam. If you're reading this document at leisure, I'll direct you to read or listen to any of their works, and your time is guaranteed to be spent better.

I would like to thank the friends who made my time in Ann Arbor so enjoyable: Arun, Aman, Ankush, Aditya, Mani, and Vivek. Some of my cherished memories are going places and eating good food with them. I'm also thankful to Gourab and Dibyajyoti for their friendship and shared grad school experiences that carried me through, and to Paritosh for his eternal friendship. I also enjoyed a sample of the diverse student activities at UM, thanks to the climbing club, the SCUBA club, and the Food Recovery Network. I'm also thankful to the American Red Cross, South-East Michigan chapter for the service opportunities that brought unique life lessons and perspectives.

Finally, I'm blessed to have the constant support of my family. I'm grateful to my parents for their tireless work to get me to where I am today, and to have my brother alongside. They shall remain a source of purpose behind all my endeavors. I dedicate this dissertation to them.

TABLE OF CONTENTS

Dedication ii

Acknowledgments iii

List of Figures vii

List of Tables viii

Abstract ix

Chapter

1 Introduction **1**

1.1 Need for Secure & Reliable Software 2
1.2 Traditional Program Analyses Landscape 5
1.3 Motivation and Contribution 7

2 Cautiously Optimistic Program Analysis **10**

2.1 Conservative Hybrid Analysis 10
2.2 Optimistic Hybrid Analysis 11
2.3 Cautious Recovery with Safe Elisions 15
2.4 Soundness Proof 16

3 Iodine: Live Information-flow Security Monitoring **20**

3.1 Introduction: Live Information-flow Tracking is Challenging 20
3.2 Background: Information Flow Analyses 23
3.3 Design: Cautiously Optimistic Program Analysis for Fast DIFT 24
3.4 Iodine Implementation 29
3.5 Evaluation: Precise Static & Fast Dynamic IFT 31
3.6 Related Work 41

4 PROV-GC: Provenance-based Sound Garbage Collection for C **45**

4.1 Introduction: Enforcing Memory Safety is Challenging 45
4.2 Background: Garbage Collection for C/C++ 50
4.3 Design: Provenance-Based Garbage Collection 54
4.4 PROV-GC Implementation 65
4.5 Evaluation: Sound & Efficient GC for C 68
4.6 Related Work 74

5 OPT-SC: Efficient Sequential Consistency for `Java` **78**
5.1 Introduction: Enforcing Strong Concurrency Semantics is Challenging 78
5.2 Background: Memory Consistency Models 81
5.3 Design: Precise Predicated Static Datarace Detection 84
5.4 Design: COPA for Efficient Sequential Consistency 88
5.5 OPT-SC Implementation 95
5.6 Evaluation: Precise Data-race Detection & Efficient SC 98
5.7 Related Work 106

6 Conclusion **108**

Bibliography **112**

LIST OF FIGURES

Figure

1.1 Program analyses landscape . . . 6

2.1 COPA workflow . . . 13
2.2 Soundness of COPA: Transformed program P' . . . 18

3.1 Example of taint analysis . . . 23
3.2 DIFT optimizations . . . 25
3.3 Workflow of optimistic hybrid taint analysis . . . 27
3.4 COPA forward recovery switching mechanism . . . 29
3.5 Result: Dynamic information-flow tracking overheads . . . 32
3.6 Result: Iodine compared to pure (full) dynamic and conservative hybrid DIFT. . . . 35
3.7 Result: Taint tracking performance on SPECint C benchmarks . . . 36
3.8 Result: Improved static taint analysis precision by assuming different invariants . . . 37
3.9 Result: Profiling invariants while software testing . . . 40
3.10 Result: Iodine's sensitivity to fraction of tainted data . . . 41

4.1 XOR linked list . . . 52
4.2 Explicit pointer provenance propagation . . . 58
4.3 Copying a pointer via implicit flow . . . 60
4.4 Implicit pointer provenance propagation . . . 61
4.5 Result: PROV-GC dynamic pointer provenance tracking overheads . . . 70
4.6 Result: PROV-GC reduces overhead of a single GC invocation . . . 72
4.7 Result: PROV-GC reclaims more memory per GC invocation . . . 72
4.8 Result: PROV-GC performance with varying heap limits . . . 73

5.1 Benefits of predicated over conservative data-race analysis . . . 86
5.2 Example Java program benefiting from predicated data-race analysis . . . 87
5.3 Workflow of OPT-SC . . . 89
5.4 Result: OPT-SC reduces execution time overhead compared to VBD and S-VBD . . . 102
5.5 Result: OPT-SC benefits for Spark with profiling using its test suite . . . 104

LIST OF TABLES

Table

3.1 Static analysis time breakups for Iodine's some-to-some taint analysis 38

4.1 PROV-GC Benchmark configurations 69

5.1 COPA improved precision of intermediate static analyses and data-race detection . . . 103

ABSTRACT

Modern computer systems still have various security and reliability vulnerabilities. Well-known dynamic analyses solutions can mitigate them using runtime monitors that serve as lifeguards. But the additional work in enforcing these security and safety properties incurs exorbitant performance costs, and such tools are rarely used in practice. Our work addresses this problem by constructing a novel technique- **Cautiously Optimistic Program Analysis** (COPA).

COPA is optimistic- it infers likely program invariants from dynamic observations, and assumes them in its static reasoning to precisely identify and elide wasteful runtime monitors. The resulting system is fast, but also ensures soundness by recovering to a conservatively optimized analysis when a likely invariant rarely fails at runtime. COPA is also cautious- by carefully restricting optimizations to only safe elisions, the recovery is greatly simplified. It avoids unbounded rollbacks upon recovery, thereby enabling analysis for live production software.

We demonstrate the effectiveness of Cautiously Optimistic Program Analyses in three areas–

- **Information-Flow Tracking (IFT)** can help prevent security breaches and information leaks. But they are rarely used in practice due to their high performance overhead ($> 500\%$ for web/e-mail servers). COPA dramatically reduces this cost by eliding wasteful IFT monitors to make it practical ($\sim 9\%$ overhead – $4\times$ speedup).

- **Automatic Garbage Collection (GC)** in managed languages (e.g. Java) simplifies programming tasks while ensuring memory safety. However, there is no correct GC for weakly-typed languages (e.g. C/C++), and manual memory management is prone to errors that have been exploited in high profile attacks. We develop the first sound GC for C/C++, and use COPA to optimize its performance ($\sim 16\%$ overhead).

- **Sequential Consistency (SC)** provides intuitive semantics to concurrent programs that simplifies reasoning for their correctness. However, ensuring SC behavior on commodity hardware remains expensive. We use COPA to ensure SC for Java at the language-level efficiently, and significantly reduce its cost (from $\sim 24\%$ down to $\sim 5\%$ on x86).

COPA provides a way to realize strong software security, reliability and semantic guarantees at practical costs.

CHAPTER 1

Introduction

Software is everywhere– from controlling critical infrastructure to democratizing technology for the masses. The very essential backbones of our industrial, financial, healthcare, education and government systems are built in its realm. Today, it not only manages our modern society, but sometimes drives aspects of its progress. Given this, ensuring that these systems are secure and reliable is critical.

And yet, we continue hearing such news as losing identities and information of millions of users to security breaches, and losing billions of dollars to system downtimes, and so on. As software have evolved, it has simultaneously grown in complexity as well as its demand for performance. Even advanced industrial computing machinery has been limited to meet this demand. Since systems cannot tolerate under-performance, this unfortunately means that strong security and reliability measures are deliberately disabled due to their prohibitive costs.

This dissertation seeks to improve the security and reliability of software systems. Some additional work in maintaining or monitoring for these properties is fundamentally unavoidable. But redundant costs can be eliminated by carefully reasoning and constructing analyses, that use optimistic assumptions, dynamic information, and careful reasoning to induce more precise and stronger optimizations. We develop a novel program analysis technique and demonstrate its practical value in designing more secure and reliable compiler and programming language runtime support tools. This enables many powerful program analyses that provide stronger guarantees at practical costs.

1.1 Need for Secure & Reliable Software

The following discussion presents a brief summary of our work along three different areas.

Always-on security monitoring

Data breaches and inadvertent leaks recur increasingly often, causing tangible damages and that to our trust in technology. Why then are systems not running always with the necessary checks? Performance considerations often preclude continuous runtime monitoring of production software, as even simple dynamic analyses incur prohibitively large overheads. Static reasoning can help avoid some of the unnecessary work by proving that some program operations *can not ever* cause potentially insecure behaviors. But such reasoning is fundamentally conservative, becoming too slow and often not completing for real programs. Even when it finishes, the results are too imprecise and thus ineffective in practice. Optimistic hybrid analysis (OHA) [1] uses likely program invariants to predicate static analysis, making it more precise and thereby effective in inducing more aggressive optimizations that reduce dynamic analysis overheads. However, one key challenge remained- how to guarantee analysis soundness in the rare event that an assumed invariant is dynamically violated. Rollbacks, although address this problem for offline analysis, are intolerable on live running software as it may require re-running years of execution history, and moreover, effects of certain operations are irreversible. We solve this rollback problem in our Cautiously Optimistic Program Analysis (COPA) to apply it for the first time on live systems. An optimistically optimized analysis can reason two types of optimizations- (1) it can remove an analysis monitor operation that provably does not modify the analysis metadata state, we call these `noop` monitors; and (2) it can remove a monitor that although updates the analysis metadata but does not affect the analysis outcome. We prove that removing the `noop` monitors are *safe elisions* as they maintain the exact metadata state as in a conservative analysis. We construct rollback-free COPA to only perform safe elisions, so that upon an invariant failure, the analysis simply performs forward recovery by switching to the conservatively optimized analysis. This allows us to apply COPA for live security monitoring of web / mail / database servers that require such soundness guarantees, significantly improving the overhead (to $\sim 9\%$) of enforcing information flow security.

Safe runtime systems for legacy languages

A large number of contemporary high-performance software are being developed in unsafe languages like C/C++. Since these languages do not provide a managed memory, such systems remain vulnerable to security threats and reliability issues. However, the cost of providing a safe runtime with a managed memory remains high for these legacy systems since reasoning for safety is much harder under their weak language-level semantics. Garbage Collection (GC) is a useful language feature that automatically enforces temporal memory safety of programs, prevents memory leaks, and at the same time alleviates programmers from the burden of explicitly reasoning about memory management. Prior works that attempted to import this language feature to C essentially rely on *value-based* heuristics to identify pointers (memory allocation addresses) at runtime. But, legal C programs can violate type-safety and manipulate pointer values, so that pointers' values no longer identify their referent memory objects. Therefore, such value-based approaches are unsound and can incorrectly reclaim memory objects that are still reachable. How could the runtime system correctly identify such hidden pointers? Although it is hard to identify, pointer information has well-defined sources (memory allocation functions). So, we construct the first *sound* GC for C that is *provenance-based*, essentially tracking all values that are derived from pointers. A naive approach would incur significant overheads in tracking pointers through all explicit and implicit channels. So, we leverage optimistic analysis, and apply targeted reasoning to identify most common pointer operations that would not propagate sufficient pointer information and would not entail dynamic tracking. We additionally identify optimizations induced by spatial memory safety of programs with well-defined behavior and properties of the C language standards. This enables us to realize a sound GC solution for C with practical overheads ($\sim 16\%$). Our provenance-based GC tool is sound, and it improves the scanning overheads and memory reclamation rate compared to state-of-the-art value-based GC for C. We show that legacy systems written in weakly-typed languages can be provided temporal memory safety at practical costs.

Language guarantees for concurrent programs

Modern high-performance computing applications rely on languages providing useful abstractions for concurrency to leverage the inherent parallelisms in the application. The *memory consistency model* defines the correct behavior of such concurrent programs by dictating the allowable orderings among accesses to shared memory from different threads of the program. While *strong* memory models provide simple and intuitive semantics, thereby simplifying reasoning for program correctness and debugging tasks, they prevent aggressive memory reordering optimizations by the compiler and underlying hardware thus incurring a high runtime performance cost. Consequently, the current language standards for widely-used C++ and Java provide strong consistency semantics only for *data-race-free* programs, where all conflicting shared memory accesses are already strictly ordered by the program's synchronization needs. The vast majority of programs with *data races* run with weak or no guarantees. This can lead to concurrency bugs with obscure behaviors, making it very difficult for programmers to reason for correctness. We seek to close this semantic gap and provide a strong memory consistency model- *Sequential Consistency* (SC), at the language-level for all programs. While prior work has explored static program analysis and speculative compilation techniques to bring down the runtime cost of enforcing SC for all programs, these solutions remain inadequate due to the ineffectiveness of applying inherently conservative analyses in their optimizations. The compiler can statically identify potential data-races and then protect only such memory accesses with expensive *fence* operations to enforce the SC orderings at runtime. However, traditional static data race analyses are imprecise and are not able to prove many memory accesses to be data-race-free. We construct an *optimistic* data-race analysis that is significantly more precise, reporting 84% fewer potential races. The resulting SC-compiler for Java, using this precise analysis, thus emits much fewer fences and enforces SC at only $\sim 5\%$ runtime overhead on commodity x86 hardware. We also improve upon the recovery mechanism of our COPA to leverage the just-in-time compilation features of the Java virtual machine.

1.2 Traditional Program Analyses Landscape

We present the relevant background on existing program analyses techniques, discuss their limitations, and motivate the challenges. Traditionally, program analyses situate along two contrasting ends– Static or Dynamic analyses.

Static Analyses use known properties of the programming language and reason across models or representations of the program in order to prove certain properties that hold for the entire program for all its possible executions. They attempt to be *sound*[1] by reasoning correctly for all possible program executions. But on the downside, characterizing possible program behaviors is generally undecidable, and so these techniques employ conservative approximations that generally make them *imprecise* thereby proving weaker properties, and becoming too slow to be practical.

Dynamic Analyses augment actual executions to track useful metadata and check or enforce properties only for the monitored executions. They attempt to be *precise* by only reasoning for program states encountered during the monitored execution and avoiding conservative approximations with dynamic observations. Dynamic analyses are widely useful in detecting well-known bug patterns, concurrency bugs, mitigate security attacks, enforce privacy policies, and even dynamically re-optimize programs. But on the downside, they incur overheads in performing the additional analysis work and can slow down programs significantly. So, their use is limited to offline retroactive debugging and not for online continuous monitoring.

Next, we discuss the directions in which static and dynamic analyses can be combined to overcome the shortcomings of each other. This design space is illustrated in Fig. 1.1.

Hybrid Analyses are a common approach to improve the efficiency of dynamic analyses by using static analyses. They first perform a *sound* static analysis, and then use the results of that analysis to induce optimizations that elide unnecessary or redundant dynamic analysis checks. For example, CCured [3] enforces memory safety for C dynamically at runtime, but uses static type inference to remove the vast majority of memory checks and only dynamically checking

[1] Static analyses for modern languages with dynamic features attempt to be *soundy* [2] and do not account for behaviors that are hard to characterize statically, e.g. dynamic dispatch and class loading in Java.

those accesses for which the static type inference fails. Hybrid analysis has been used to optimize dynamic race detectors [4], taint tracking systems [5], and enforce memory safety at runtime [6, 7].

However, traditional sound static analyses have a fundamental limitation– in order to be sound, they become over-conservative in assuming all possible program states. This often leads to unacceptable imprecision in reasoning that is inadequate in effectively reducing the dynamic overheads.

Blended Analyses work in the other direction to improve the precision of static analyses by using light-weight dynamic analyses. They combine the results of a fast dynamic analysis over several executions to construct a dynamic program structure representation, which can then be used by the static analysis to significantly reduce its scope and induce more precise reasoning. This approach has enabled useful static analyses like reasoning for web-based frameworks [8, 9], program slicing of large complex programs [10], and taint analysis [11]. A few systems also use the more precise static analysis to accelerate a final dynamic analysis [12, 13].

Unfortunately, the initial dynamic analysis being unsound, the resulting static analysis is also unsound which these systems fail to compensate for. As a result, these systems provide much weaker analysis guarantees and cannot be applied for critical analyses like security monitoring.

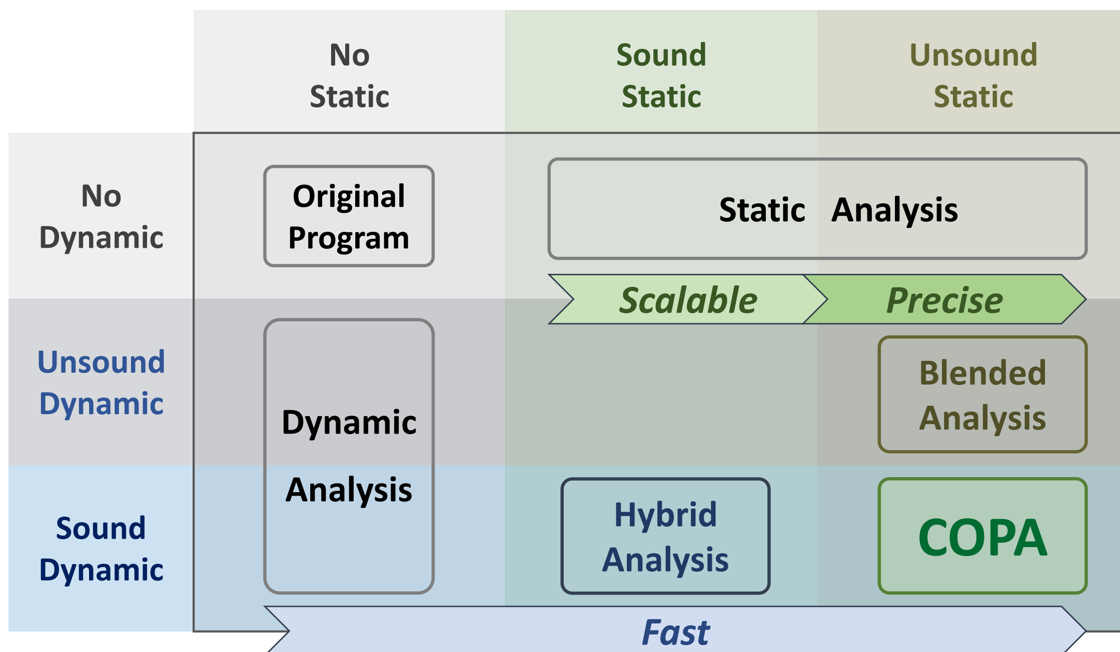


Figure 1.1: Program analyses landscape

Optimistic Hybrid Analysis (OHA) improves traditional analyses in both directions. It first learns useful properties about programs' dynamic behaviors from several profiling runs and uses these *likely invariants* to carefully predicate the static analysis. The more precise static analysis induces stronger optimizations on the final dynamic analysis. Importantly, OHA solves the unsoundness introduced during static analysis with speculative execution, identifying when an invariant actually fails during any execution and then recovering to a sound conservatively optimized analysis. This approach has enabled highly optimized solutions to program slicing and data-race detection [1].

However, the key question in applying OHA is how to recover analysis soundness when an assumed likely invariant fails. Because invariant violations are quite rare, prior work relies on a rollback recovery approach. Upon an invariant violation, the optimistic analysis is rolled-back and a conservative hybrid analysis is re-executed. However, this approach can suffer from **unbounded rollbacks**, because a whole-program analysis can induce optimizations as far back as at the beginning of the program. This severely limits its application only to offline post-mortem debugging analyses where such rollback-replays can be tolerated.

1.3 Motivation and Contribution

The rollback recovery problem limits OHA from being applied with powerful whole-program analyses for online monitoring on live production software. Rollback and re-execution on server applications would severely impact their availability, as re-executing the conservative analysis from beginning would potentially incur replaying years worth of execution since the last reboot. Moreover, rollbacks are infeasible when certain operations like sending a packet over the network cannot be reverted. This dissertation focuses on the challenge of applying OHA to online dynamic analyses on live production software by eliminating the need for rollback recovery.

Thesis statement: ***Program analyses leveraging optimistic assumptions with cautious reasoning can make online dynamic analyses practical on live production software***.

Our Cautiously Optimistic Program Analysis (COPA) technique employs two key principles:

Optimistic Analysis: It infers ***likely program invariants*** from dynamic observations, and assumes them in a ***predicated static analysis*** to reason more precisely thereby identifying and eliding many wasteful runtime monitors from the ***optimized dynamic analysis***. The resulting system is fast, and is naturally sound for executions that satisfy the assumed likely invariants.

Cautious Reasoning: In case a likely invariant rarely fails at runtime, it also ensures soundness by recovering to a conservatively optimized analysis. It identifies the exact conditions when the optimistic analysis can become unsound, and detects such invariant violations early with ***eager invariant checks***. It also carefully restricts optimizations to only ***safe elisions*** that do not diverge the analysis metadata state. These two properties together greatly simplify the recovery process when a likely invariant rarely fails– it avoids rollback upon a likely invariant violation, and enables ***forward recovery***– the analysis can simply switch to a conservatively optimized one and continue forward.

The combined benefit of these two principles can enable several useful dynamic analyses applications for online analyses on live production software at practical overheads.

Organization

The rest of this dissertation is organized as follows:

[Chapter 2] discusses the limitations of conservative and prior optimistic hybrid program analyses, and then presents our *Cautiously Optimistic* Program Analysis (COPA) technique in detail with its design objectives, the workflow, and a formal discussion of its soundness guarantee.

Then, we demonstrate COPA's benefits in three areas–

[Chapter 3]: Dynamic Information Flow Tracking (DIFT) can actively enforce information-flow policies and detect malicious behaviors, but remains prohibitively expensive. We solve the rollback recovery problem in optimistic program analysis and apply it to optimize DIFT for live security monitoring on production software.

[Chapter 4]: Enforcing temporal memory safety by constructing a correct garbage collector remains difficult for languages like `C` with weak semantics. We design a sound way of garbage collection for `C` by efficiently tracking provenance of pointers at runtime.

[Chapter 5]: Providing strong and intuitive semantics for concurrent programs simplifies program analyses and debugging tasks, but remains expensive on commodity hardware platforms. We enable efficient language-level *Sequential Consistency* for `Java`.

[Chapter 6] concludes with a summary of our key contributions and directions for future work.

CHAPTER 2

Cautiously Optimistic Program Analysis

Traditional hybrid analysis remains prohibitively slow due to the fundamental conservative approach and imprecision of the underlying sound static analysis. We discuss how the recent approach of Optimistic Hybrid Analysis (OHA) [1] mitigates some of these limitations by leveraging likely invariant assumptions, but present a new problem- how to recover soundness of the analysis when an invariant assumption is violated. Then our Cautiously Optimistic Program Analysis (COPA) approach solves this problem to enable fast dynamic analysis on live software while eliminating the need for rollback-replay. This chapter builds the foundational ideas of our work, and we later elaborate on specific details while discussing its applications.

2.1 Conservative Hybrid Analysis

A naïve dynamic analysis would instrument virtually all instructions with additional *monitor* operations to maintain dynamic information that may be needed to check certain safety or security properties. This can result in an order of magnitude or more overhead. However, this overhead is not fundamental in enforcing most useful properties. Because, in rigorously tested and well-behaved programs, such properties hold in most correct executions. As a result, much of the additional work in maintaining analysis state and checking for properties is wasteful. A sound static analysis can ideally prove for a large number of program instructions that they do not violate the properties in any possible execution and then safely elide the dynamic analysis monitors associated with them [14].

Traditional sound static analyses reason about *all* possible future executions, including many infeasible ones due to over-approximation. As a result, the analysis state-space can explode, and indeed many useful static analyses become too slow and do not scale to large complex software. Moreover, if and when the analyses terminate, the results are imprecise as they cannot effectively reason the target property for many instructions over the large space of all possible (and many infeasible) execution states. As we will see in the following chapters, this fundamental imprecision has limited conservative hybrid analysis from effectively optimizing dynamic analysis, and such techniques still remain impractical.

2.2 Optimistic Hybrid Analysis

When used for optimizing online dynamic analysis during execution, the static analysis need not reason over all possible execution states. Instead, it needs to only care about those dynamic executions that will actually be encountered.

Likely Invariants to Predicate Static Analysis The static analysis can be *predicated* by making a set of assumptions called *likely invariants*. These are program properties that are almost always true but hard to prove statically. For example– one simple type of likely invariant is unreachable code. A vast majority of code deals with custom logic to deal with uncommon, exceptional and erroneous executions. Furthermore, programs only exhibit a small subset of their possible behaviors when running under a specific configuration. So, it is reasonable to expect a significant fraction of code being not reached in most correct executions. So, assuming this invariant would significantly reduce the analysis state-space, allowing the static analysis to reason much more precisely on common-case behavior of the analyzed program. Consequently, the predicated static analysis can additionally prove the target property for many instructions in all the dynamic executions that satisfy the assumed likely invariants.

In our work, the likely invariants are collected in a rigorous profiling phase prior to the static analysis. Observed dynamic behaviors that hold true across all profiled executions are assumed

in the predicated static analysis. The type of invariants are specific to the target analysis being optimized and their benefits are also varied. We discuss the specific invariants that we use in the subsequent chapters.

Traditional static analyses have not leveraged such observed likely invariants for the sake of soundness. Contrary to intuition, a key distinction of our design is that it does not achieve precision by sacrificing soundness. Although the predicated static analysis is only sound for executions in which the assumed likely invariants actually hold, the soundness of the final dynamic execution can still be guaranteed as long as invariant violations are detected immediately and the execution is then recovered appropriately.

Optimistic Dynamic Analysis The predicated static analysis effectively elides many dynamic analysis monitors using its more precise reasoning. The resulting optimized dynamic analysis is fast, and is guaranteed to be sound for all executions in which the assumed invariants hold. However if an invariant rarely fails, the dynamic analysis may lose all soundness guarantees. In order to recover the analysis soundness, the dynamic analysis must then additionally validate the assumed likely invariants during execution and somehow recover the analysis when an assumed likely invariant is violated. Fig. 2.1 illustrates the overall workflow of the analysis framework. Profiled likely invariants are assumed in the predicated static analysis thereby improving its precision. The precise static analysis results are used to effectively optimize the dynamic analysis, and invariant checks are instrumented to validate the assumed likely invariants. In the rare event of an invariant violation, the analysis must be recovered. This is the key challenge that we address in this dissertation.

Design Objectives

Since our COPA analysis framework is to be applied on live running programs for online dynamic analysis, it must meet the following objectives in order to be effective:

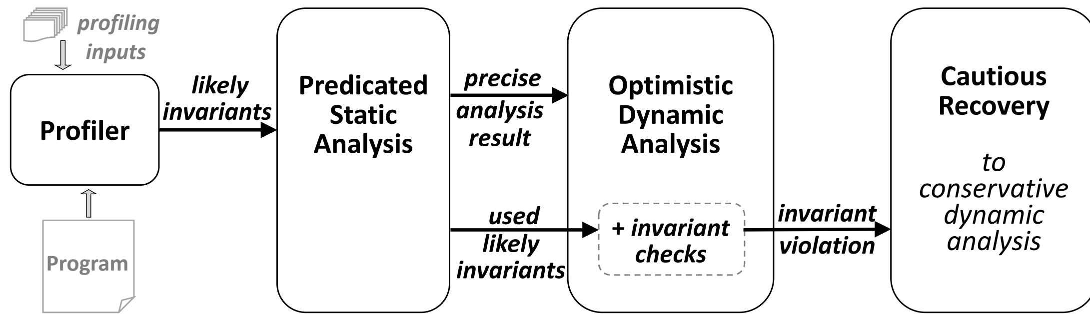


Figure 2.1: COPA workflow

O1 *Invariants induce strong optimizations*: By assuming the invariants, the predicated static analysis should be able to significantly reduce its state-space, thereby making it more precise. If invariants do not induce such strong optimizations, they will not effectively reduce the dynamic overheads in the common-case.

O2 *Invariant checks are inexpensive*: To guarantee soundness, the dynamic execution should additionally check that the assumed invariants hold during an execution. Theses invariant checks should be simple and inexpensive, so that the benefits of COPA are not outweighed by the overhead of dynamically verifying the invariants.

O3 *Invariants fail only rarely*: Invariants should capture dynamic behaviors that are hard to prove but almost always hold true. This would ensure that executions do not suffer from frequent invariant violations. Otherwise, recovering from an invariant violation can incur additional overheads thereby limiting our system's benefits.

O4 *Invariant checks are eager*: Invariant failures must be detected *before* the invariant actually fails. This would allow the execution to pause at a dynamic state that has not yet been affected by the likely invariant violation and recover the analysis without losing its guarantees.

O5 *Recovery is safe and quick*: The recovery process itself must be efficient and carefully constructed so as to not violate the analysis correctness. The support needed for recovery in a rare execution should not entail extensive overhead during common executions. Rollback-replay based recovery is not practical.

2.2.1 Problem: Rollback Recovery in OHA

In most dynamic executions, the likely invariants will hold and the optimistic dynamic analysis will be sound. But when a likely invariant fails dynamically, it may render the predicated static analysis' optimizations unsound. The dynamic analysis then requires a mechanism to recover from an invariant failure. As we use whole-program static analysis, at runtime it is non-trivial to determine the effect of a current invariant failure on the soundness of an elided monitor in the past.

Prior OHA work [1] addressed this problem by completely replaying the program execution from the beginning using the conservatively optimized dynamic analysis. Since invariants rarely fail, this rollback recovery is an acceptable solution for offline retrospective analyses such as debugging and forensic analyses. However, a rollback to the beginning of the program is intolerable for online analyses on live executions, as it would severely compromise the system's availability.

Unbounded Rollbacks Bounding rollbacks is generally hard for predicated *whole-program* static analyses. Determining the latest point in program execution up to which a rollback is needed is an unsolved problem. For many analyses, especially backward data-flow analysis, it may not be possible to bound the rollback window. This unpredictable and unbounded downtime caused by rollback is problematic for live executions.

Logging Overheads Support for rollback introduces significant additional overheads even for executions where the likely invariants hold true. This overhead includes the cost of logging for replay and periodic check-pointing. Therefore even when the invariants are not violated, eliminating rollbacks altogether would improve OHA by getting rid of these overheads. The actual cost for rollback-replay is minor as invariant violations can be made to be rare with sufficient profiling.

We address this problem by enabling forward recovery upon any invariant failure, thus completely eliminating the need for rollbacks.

2.3 Cautious Recovery with Safe Elisions

Rollbacks are fundamentally caused by the dependence between the current monitor being elided and potential future invariant failures. Our idea behind COPA is to distinguish safe elisions, which do not have such dependencies, from unsafe elisions.

A predicated static analysis in OHA elides a monitor as long as it can prove that it is unnecessary to guarantee soundness of dynamic analysis in an execution where the invariants hold. But an elided monitor is a *safe elision* only if it can additionally prove that an invariant violation in an execution would not affect the soundness of any preceding elisions of that monitor.

Rollback-free COPA is realized by restricting its predicated static analysis to only using safe elisions, and switching to a conservatively optimized analysis on invariant violation.

Statically proving safe elisions is non-trivial for many analyses. To make such an analysis practical and simple to construct, we further observe that `noop` monitors are safe elisions. A `noop` monitor is one that does not change the analysis metadata state. Eliding `noop` monitors is safe for the following reasons: by construction, COPA instruments invariant checks such that they detect any invariant violation *before* an execution violates the invariant. Given this, when a `noop` monitor is elided before an invariant violation, it is guaranteed that it would be a `noop` monitor even in the conservatively optimized analysis, and therefore its elision is sound even when there is a later invariant violation. Thus, `noop` monitor elisions are safe elisions.

By restricting to only safe elisions of `noop` monitors that do not modify the analysis metadata state, and eagerly checking for invariant violations dynamically, the analysis state is guaranteed to be exactly the same as in a conservative analysis at the point of invariant violation detection. Thereafter, the analysis can then be recovered by simply switching to a conservative analysis that re-instruments the optimistically elided dynamic monitors. We discuss this recovery mechanism in detail in Chapter 3.

The cautious approach of restricting to only safe elision optimizations and eagerly checking likely invariants solves the rollback-recovery problem of OHA and enables COPA to be applied for online analysis on live running software.

2.4 Soundness Proof

In this section, we formalize the notion of two analyses being state-identical, and then prove the soundness of rollback-free COPA analysis by showing that it's state-identical to a conservative hybrid analysis.

2.4.1 Notations and Notions

An analysis A is a transformation of a program P that only generates additional metadata state σ_A and has no side-effect on P's program state μ_P. We define out_A to be the outcome of all dynamically failed check monitors.

We will use the following notations to refer to analyses instances:

UNOP is the unoptimized dynamic analysis that does not elide any monitors.

CONS is the dynamic analysis optimized by conservative static analysis.

OPTI_I is the dynamic analysis optimized by predicated static analysis assuming the set of invariants I.

COPA_I is the rollback-free dynamic analysis optimized by safe elisions using predicated static analysis assuming the set of invariants I.

$\sigma_A(l)$ denotes the metadata state of dynamic analysis A at the program location l. I-FAIL(i) denotes the point(s) in program execution where the invariant assumption i dynamically fails. I-CHECK(i) denotes the program location(s) where the invariant validation checks are instrumented. A `noop` monitor is either a track monitor that does not modify σ_A, or a check monitor that succeeds.

Definition 1. *Analysis equivalence* : We say that dynamic analysis A' is equivalent to dynamic analysis A, denoted by $A' \equiv A$, if for all executions, their analysis outcomes are the same, i.e., $out_{A'} = out_A$.

Definition 2. *State-identical* : We say that dynamic analysis A' is state-identical to dynamic analysis A, denoted by $A' = A$, if for all executions, their terminating metadata states σ_A and $\sigma_{A'}$ are identical, i.e., $\sigma_{A'} = \sigma_A$.

2.4.2 Axioms

Axiom 1. CONS *is sound* [14], *i.e.,* CONS $\equiv$ UNOP.

CONS only elides those monitors which can be proven to not change the analysis outcome in all executions. $\therefore$ CONS $\equiv$ UNOP.

Axiom 2. OPTI$_I$ *is sound when the invariants hold* [1], *i.e.,* $I \models$ OPTI$_I$ $\equiv$ CONS.

In addition to those elided by CONS, OPTI$_I$ elides only those monitors that can be proven to not change the analysis outcome in dynamic executions that satisfy I. $\therefore I \models out_{\text{OPTI}_I} = out_{\text{CONS}} \rightarrow I \models$ OPTI$_I$ $\equiv$ CONS.

Axiom 3. *Invariant violation is detected before a program execution reaches a state that fails an invariant, i.e.,* I-CHECK$(i) <$ I-FAIL(i).

By construction, our invariant checks are instrumented such that this property holds.

Axiom 4. COPA$_I$ *only elides monitors that are* `noops`*.*

By construction in §2.3, COPA$_I$ uses forward predicated static data-flow analysis to elide only those monitors that it can prove are `noops`.

2.4.3 Soundness of Rollback-free COPA

We first show that COPA$_I$ is state-identical to a sound conservative hybrid analysis for executions where the invariants hold. Next, we provide a simple program transformation that makes the COPA$_I$ state-identical to CONS even at the point of a dynamic invariant failure. Finally, we show that the above property allows a forward recovery of COPA$_I$ upon an invariant failure, and makes the whole dynamic analysis sound for all executions.

Lemma 5. COPA$_I$ *is state-identical to* CONS *when the invariants hold, i.e.,* $I \models$ COPA$_I$ $=$ CONS.

Proof Sketch. By Axiom 4, COPA$_I$ elides only those monitors that can be proven to be `noops` in dynamic executions that satisfy I. $\therefore I \models \sigma_{\text{COPA}_I} = \sigma_{\text{CONS}} \rightarrow I \models$ COPA$_I$ $=$ CONS. $\square$

Lemma 6. COPA$_I$ *is sound until an invariant fails, i.e.,* $\sigma_{\text{COPA}_I}(\text{I-FAIL}(i)) = \sigma_{\text{CONS}}(\text{I-FAIL}(i))$.

Proof Sketch. Consider the analysis COPA$_{\{i\}}$ with a single invariant i. $\neg\{i\} \not\models$ COPA$_{\{i\}}$ $=$ CONS, i.e., we cannot guarantee soundness for the entire program P if the invariant fails in a dynamic execution.

Let I-FAIL(i) be the first instance of an invariant failure in the dynamic execution of P. Now, consider the program P' obtained by the following transformation (shown in Fig. 2.2): immediately after the location of each invariant check, we instrument a HALT instruction conditional on the invariant i having failed. The elided monitors are shown as equivalent `noops`.

By Axiom 3, the invariant check preceding I-FAIL(i) will detect the invariant failure before the program execution reaches a state that fails the invariant. Therefore, the modified program P' will HALT after the failed I-CHECK(i), and before I-FAIL(i). This is equivalent to a program executing without an invariant failure.

By Lemma 5, COPA$_I$ $=$ CONS for P'. Since, P and P' only differ in their termination behavior and P' HALTs at I-FAIL(i), we have that:
$\sigma_{\text{COPA}_I}(\text{I-FAIL}(i)) = \sigma_{\text{CONS}}(\text{I-FAIL}(i))$ for P. $\square$

```
                ...
         l1:    noop
                ...
I-CHECK(i):     if (¬i)
                  HALT
 I-FAIL(i):     ...
         l2:    noop
                ...
```

Figure 2.2: Transformed program P'

Theorem 7. COPA_I *with forward-recovery is sound.*

Proof Sketch. By the soundness of COPA_I on the HALT-transformed program P' in Lemma 6, we have that the metadata state $\sigma_{\text{COPA}_I}(\text{I-FAIL}(i))$ at the location of invariant failure is state-identical to that in CONS. Therefore, the forward-recovery mechanism can simply switch to CONS on an invariant failure, and that analysis as a whole is analysis-equivalent to CONS. $\therefore$ by Axiom 1, COPA_I with forward-recovery is sound. $\square$

2.4.4 Insight Summary

Contrasting Axiom 2 and Lemma 5, the key difference is that when I holds, $\text{OPTI}_I \equiv \text{CONS}$ but $\text{COPA}_I = \text{CONS}$. While the generic OPTI_I aggressively elides monitors to only preserve analysis-equivalence, COPA_I only elides `noop` monitors, thus being state-identical to CONS. This allows the analysis to simply switch to conservative analysis CONS upon invariant violation.

The primary contribution of this dissertation is to solve the rollback recovery problem in an Optimistic Hybrid Analysis framework, thereby enabling online analyses on live running software. We first use COPA to optimize taint analysis for live security monitoring in Chapter 3, then apply this for a novel application in Chapter 4– constructing a sound Garbage Collector for C. In Chapter 5, we improve COPA's recovery mechanism and apply it for efficient language-level Sequential Consistency for concurrent Java programs.

CHAPTER 3

Iodine: Live Information-flow Security Monitoring

Dynamic information-flow tracking (DIFT), also referred to as taint-tracking, is useful for enforcing security policies, but rarely used on live running software, as it can slow down a program by an order of magnitude. Static program analyses used to prove safe execution states and then elide unnecessary DIFT monitors, yield only marginal benefits due to their need to maintain soundness.

Cautiously Optimistic Program Analysis (COPA) can significantly reduce DIFT overhead and still be sound– it predicates the static taint analysis to assume likely invariants gathered from profiles to dramatically improve precision. The optimized DIFT is sound for executions in which those invariants hold true, and otherwise recovers to a conservative DIFT. We overcome the main problem with using COPA to optimize live executions – *unbounded rollbacks*. We eliminate the need for *any* rollback during COPA recovery by limiting to only *safe elision* optimizations.

Our tool, Iodine, reduces DIFT overhead for enforcing security policies to 9%, which is $4.4\times$ lower than that with traditional hybrid analysis, while still being able to be run on live systems.

3.1 Live Information-flow Tracking is Challenging

Dynamic information-flow tracking (DIFT) [15] is a powerful method for enforcing a security or privacy policy. It tags source data (e.g., sensitive user input) as tainted, propagates taints through data and/or control flow, and checks if tainted data reaches sinks (e.g., network output). DIFT can help detect a wide range of security attacks [16, 17, 18, 19, 20, 21, 22] such as SQL injection, cross-site scripting, overwrite attacks, etc. It is also used to enforce information-flow policies that prevent sensitive information from leaking through untrusted channels [23, 24, 25].

In spite of its established benefits, DIFT is rarely used in practice today, due to its prohibitive performance overhead [26]. In a naive dynamic taint-tracking, every instruction has to be monitored to propagate taints. There have been several attempts to reduce this cost, e.g. by reducing tainted sources [27], by coarsening the taint granularity [18], and by decoupling program execution to perform a symbolic taint analysis [28, 29]. These approaches can compromise accuracy, introduce parallelization and synchronization costs, and still remain prohibitive for production use [30].

Optimistic Hybrid Analysis [1] (OHA) For rigorously tested production software, execution paths that violate an information-flow policy are almost certainly either rare or impossible. For such programs, pure dynamic taint analyses fundamentally do more work than necessary. A static taint analysis can identify instructions which cannot propagate taints to a sink [14], and DIFT monitors for such instructions can be elided. By assuming program properties that are almost always true but hard to prove statically, OHA can dramatically improve the precision and scalability of static taint analysis, thereby reducing DIFT overhead.

A fundamental problem with OHA is that, if the assumed *likely invariants* fail during an execution, then the soundness of dynamic analysis for that execution is compromised. To ensure soundness, prior OHA work [1] checked the likely invariants at runtime, and when they fail, the program execution is replayed from the *beginning* with a conservatively optimized dynamic analysis. This unbounded rollback-recovery is acceptable only for retrospective offline analyses, and not feasible for online security analysis of live executions.

We solve this problem by completely eliminating the need for rollbacks and enable *forward recovery* on a likely invariant failure. Rollbacks in OHA are caused by the runtime dependence between the current monitor being elided, and any potential future invariant violations that may affect the soundness of that elision. In order to construct rollback-free COPA, we must break this dependence. In other words, any monitor elided during a program execution, before an invariant failure, has to be proven to be unnecessary to ensure soundness of the dynamic analysis for the entire execution. We refer to eliding monitors satisfying this property as ***safe elisions***.

Safe Elisions for *rollback-free* COPA: Our key idea is to constrain predicated static analysis, such that it removes a runtime monitor only if it can prove that it is a safe elision. Given this, when a likely invariant fails at runtime, it is sufficient to simply switch to a conservatively optimized analysis, and continue forward with the execution.

To restrict COPA to safe elisions, we further observe that many analyses, particularly bug finding and security analyses such as DIFT, often have monitors that do not modify any analysis' metadata state when executed. We call such monitors ***noop* monitors**. By constructing a predicated static analysis that identifies and elides only `noop` monitors, we guarantee that any elision done by our predicated static analysis will not have any effect on dynamic analysis state until an invariant failure. Consequently, the soundness of these elisions cannot depend on any potential future invariant violations, because eliding a `noop` has the same effect as executing a `noop`, making the `noop` elisions *safe*, and enabling forward recovery.

This enables efficient and sound DIFT running on live executions without requiring rollbacks. Our work makes the following contributions:

- We construct a novel optimistic hybrid analysis technique to realize low-overhead DIFT for live executions.
- We solve an important unresolved problem with OHA, which prevents its use for live analysis: need for unbounded roll-back when a likely invariant fails. We prove that restricting predicated static analysis to eliding only `noop` monitors guarantees meta-data equivalence between optimistic and conservative hybrid analyses. This property in turn enables forward recovery when an invariant fails.
- We improve the profiling methodology for OHA based on regression and beta-testing. We show that likely invariants profiled using regression test suites are effective in obtaining majority of the performance benefits.
- Our approach reduces the overhead of DIFT to 9%, which is $4.4\times$ lower than that with conservative hybrid analysis, and $68\times$ lower than that with pure dynamic analysis.

3.2 Information Flow Analyses

Information-flow analysis, also called *Taint Analysis* [15] computes how the information in a given value of a program state is influenced by other relevant values. The analysis requires specifying a *taint policy* consisting of three components–

Sources identify program locations where a specific *taint* marking is attached with a value. Typical sources include program arguments and external input interfaces like console / file / network input.

Sinks assert specified checks on the taint state of certain values at a given program location. These checks typically assert the presence or absence of certain taints on values, typically before the program emits them via an output interface, e.g. sending a network packet.

Propagation policy determines how taints for new values are computed. Taints typically propagate **explicitly** via data-flow– when a tainted value is used in a computation, the resultant should also carry the source's taint(s). Taints can also propagate **implicitly**, e.g. via control-dependence– when values are computed conditional on a tainted value, the resultant values should carry the taint(s) of the condition variable. The propagation policy also specifies how to accumulate taint sets, typically using union or re-assignment.

Additionally, some taint analyses can specify certain *untaint* operations that clear or sanitize a tainted value, e.g. an encryption function may untaint the taint of its plaintext argument.

```
1  void main (...) {
2    int a, b, c;
3    scanf("%d", &a);
4     t(a) ← {secret}                          source
5    if (a < 0) b = -1;
6     t(b) ← t(a) = {secret}          implicit track
7    c = a * b;
8     t(c) ← t(a) ∪ t(b) = {secret}            track
9     assert(secret ∉ t(c));                   check
10   send(..., c);
11 }
```

Figure 3.1: Example of taint analysis

A dynamic taint analysis instruments *monitor* operations for instructions in the target program. *Track monitors* propagate taints from the source operands to the resultant of an instruction, as per the specified propagation function. *Check monitors* assert predicates on the taint state at sinks.

Fig. 3.1 shows an example program instrumented with taint analysis monitors (highlighted). In line line 3, a is read from user in-

put, so this is treated as a taint source and attached with a $secret$ taint. In line line 5, b's value is determined by whether a is negative or not, so taint propagates implicitly. In line line 7, c is computed from a and b, so taint propagates explicitly and is computed as the union of taints of source operands. Finally in line line 10, c's taint is checked before emitting it on the network.

Uses: Taint analysis can be tailored to a specific application by adjusting its taint policy. For example, information leakage is an important concern in database and web-service applications, where taint analysis is used to track the flow of sensitive information through program execution and prevent its leakage through unsecured channels. Taint analysis [31] is widely used in security analyses of programs to detect and prevent against overwrite attacks [16, 17, 18], command injection attacks [19, 20], cross site scripting attacks in web applications [21, 22], and to enforce information flow policies [24]. It has also been applied in semantic analysis of programs for program understanding [32], testing and debugging [33, 34].

3.3 Cautiously Optimistic Program Analysis for Fast DIFT

First, we illustrate the limitations of conservative hybrid analysis and motivate OHA [1] with an example of DIFT monitoring. Then we introduce the design of Iodine, an instance of our COPA analysis that significantly reduces DIFT overhead and supports live executions by eliminating the need for rollback-replay.

Fig. 3.2(a) illustrates naive DIFT. Assume that s is a taint source, and printf is a sink. Taint propagates from s to y (line 2), and then it may or may not propagate to z (line 4) depending on the branch outcome in line 3. If the taint does propagate to z, it can reach out (line 5), and then reach the sink (line 6), causing an assertion failure.

3.3.1 Conservative Hybrid Taint Analysis

As shown in Fig. 3.2(a), a pure DIFT instruments virtually all instructions to propagate taints. This can result in an order of magnitude or more overhead. However, this overhead is not fundamental to enforcing a taint analysis. Because, in a rigorously tested program, information-flow leaks are rare. As a result, many of the DIFT monitors are either not propagating taints, or even if they do, they do no reach any sink. A sound static data-flow analysis can prove these properties and remove these dynamic monitors [14].

A static analysis constructs a data-flow model of the program, using the same taint policy as the dynamic taint analysis. From this static model, the hybrid analysis will typically optimize its dynamic taint monitors in two ways:

Forward Taint Analysis reasons from taint sources forward in the program, determining if the source operands of an instruction *may* be tainted or not. If none of the source operands may be tainted for an instruction, then the static analysis can remove its monitor. For example, in Fig. 3.2(b), the analysis can reason that neither source operands in the instruction `x = c + 3` are tainted, and therefore `x` will not be tainted, allowing its monitor to be elided.

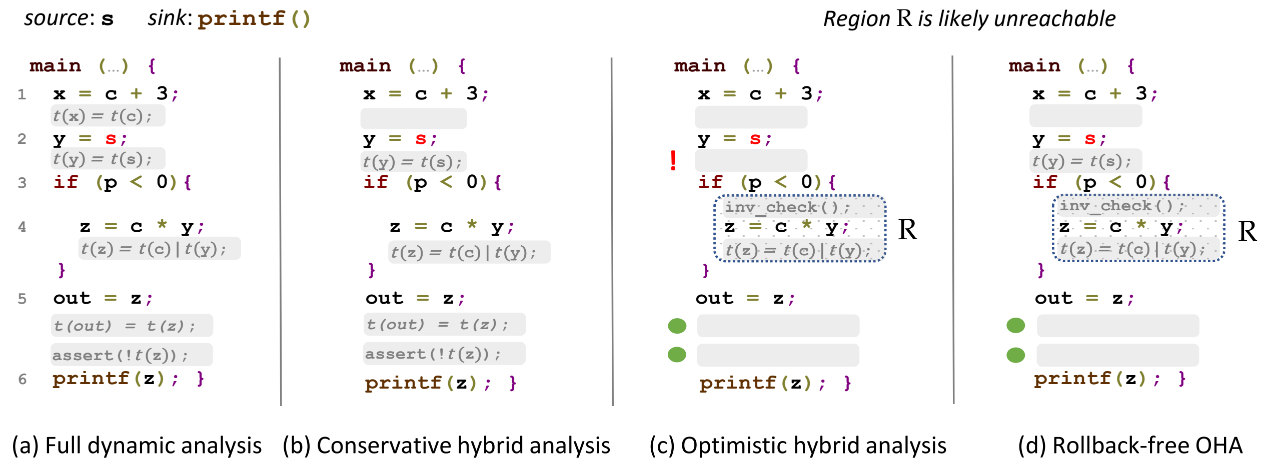


Figure 3.2: DIFT optimizations. Green dot indicates safe `noop` elisions, and ! indicates unsafe elision.

Backward Taint Analysis reasons whether a destination operand of an instruction *may* reach a sink. If not, the monitor for that instruction is elided, even if it can be tainted. In Fig. 3.2(b), the conservative static analysis cannot leverage this optimization to elide any monitors, because it cannot prove this property for any of the instructions considering all possible executions.

3.3.2 Optimistic Hybrid Taint Analysis

As we discussed in Chapter 2, traditional sound static analysis is limited by its sound consideration of all possible execution states. But, static analyses used for optimizing a dynamic analysis should ideally consider only those states that will be realized in the analyzed dynamic executions. Targeting the expected executions can significantly improve the precision and scalability of static analysis, thereby optimizing a dynamic analysis much effectively than its conservative counterpart.

Fig. 3.2(c) illustrates this untapped opportunity. If all expected executions of this program only have non-negative values for the variable `p`, the code region `R` is never executed. A sound static analysis cannot assume this behavior, because there are legal executions where `p < 0`. However, by constraining the static analysis to expected dynamic executions, Iodine can reason that the variable `z` does not get tainted due to `y` in line 4, and in turn proves that `out` in line 5 cannot be tainted. Therefore, it elides track monitor for line 5, and check monitor for the sink in line 6. Furthermore, backward data-flow analysis determines that taint of `y` in line 2 can never reach any sink, and elides its monitor. None of these three monitors could be elided using conservative static analysis (Fig. 3.2(b)).

Iodine's work-flow is illustrated in Fig. 3.3. First, a profiler observes representative executions to gather a set of *likely invariants* – dynamic execution properties that almost always hold, but are hard to prove statically, e.g. likely unreachable code, likely callee sets, and likely unrealized call contexts [1]. Second, these likely invariants are used to constrain the state-space resulting in a *predicated static taint analysis*. This is much more precise and scalable than a conservative sound static taint analysis, and enables Iodine to aggressively elide DIFT monitors. The program is then instrumented with the remaining DIFT monitors along with necessary invariant checks.

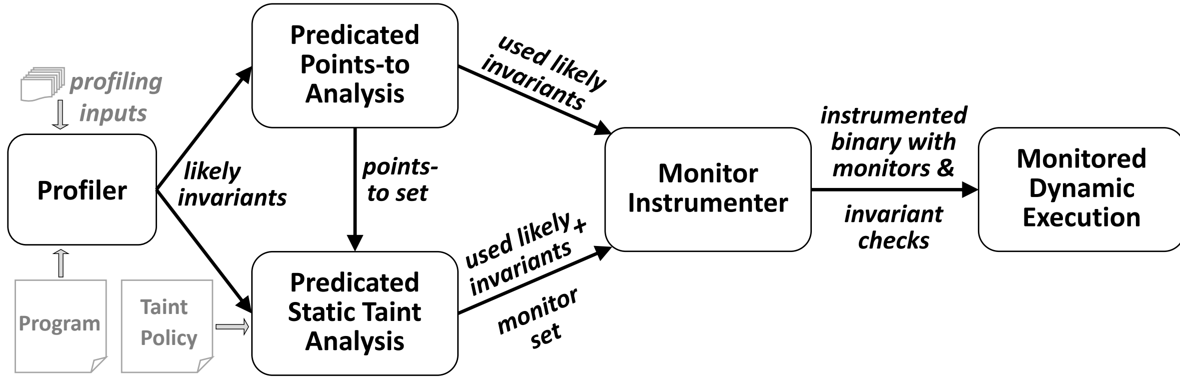


Figure 3.3: Workflow of optimistic hybrid taint analysis

3.3.3 Safe Elisions of `noop` Monitors

In §2.2.1, we saw how invariant violations in a general OHA analysis may require a rollback to recover the analysis. For example, in Fig. 3.2 (c), if the likely unreachable code invariant (R) is violated in line 3, it would render the past elision of monitor for line 2 to be unsound since the taint metadata state would diverge.

We then discussed in §2.3, how to overcome this problem by identifying `noop` monitors and performing *safe elisions.* A `noop` track monitor is one that does not change the taint analysis metadata state. A `noop` check monitor is one that always succeeds the taint check. For example, in Fig. 3.2(c), monitors for lines 5 and 6 are `noop` monitors, if we assume `R` is unreachable. Monitor for line 2, however, is not a `noop` monitor, as its execution can modify the taint set even if invariants hold true.

Elisions in Predicated Forward Analysis are Safe

In §3.3.1, we discussed forward and backward static taint analysis. Forward static data-flow taint analysis elides a monitor for an instruction by proving that its source operands must not be tainted. The taint for the destination operand of such an instruction remains unchanged. Thus, all the monitors elided by predicated forward taint analysis are `noop` monitors, and therefore safe elisions.

Fig. 3.2(d) shows that in rollback-free COPA, the elision of the monitors for lines 5 and 6 induced by forward taint analysis, are both `noop` safe elisions.

Elisions in Predicated Backward Analysis may not be Safe

Monitors elided by a predicated backward taint analysis are not guaranteed to be safe elisions. A backward taint analysis seeks to prove that an instruction's destination taint does not reach a sink, and if so it elides its monitor. Monitors elided by this analysis are not guaranteed to be `noops`. For example, the monitor for line 2 in Fig. 3.2(d) is not a `noop`, because it changes the taint of `y`. But a predicated backward analysis can elide it by assuming `R` is unreachable. However, during an execution, if that invariant fails, recovery must somehow produce the correct taint state of `y`, before proceeding forward. Given that we use a whole-program analysis, it is unclear how far the execution needs to be rolled-back and re-executed.

A more fundamental reason why elisions in backward-analysis may not be safe is their dependence on invariants holding true in the future. It may still be possible to construct safe elisions through sophisticated optimizations. For example, if we can somehow determine the set of all monitors elided due to a particular invariant (`R` is unreachable), then hoisting the invariant check before those elisions can make them safe elisions. Such a transformation is non-trivial for a predicated whole program analysis. Fortunately, we found the predicated forward taint analysis to be quite effective by itself. Also, backward analysis is not useful for certain information-flow policies such as one that monitors taints from sources to all possible locations in a program.

3.3.4 Rollback-Free Cautiously Optimistic Taint Analysis

Iodine uses a predicated forward taint analysis along with a conservative backward taint analysis. Optimized dynamic analysis (*fast-path*) is executed until an invariant fails. As the analysis only elides `noop` monitors, it tracks exactly the same metadata as a conservative analysis at all program points. Fig. 3.4 shows the forward recovery mechanism– a conditional branch is instrumented for every invariant check, which switches the control to a conservative analysis (*slow-path*) when any check fails. The execution then continues forward in the slow-path. This switch is safe due to two reasons: (1) the two paths only differ in analysis logic and maintain the same program state, and (2) safe elision guarantees equal analysis metadata state at invariant violation. Care is taken to

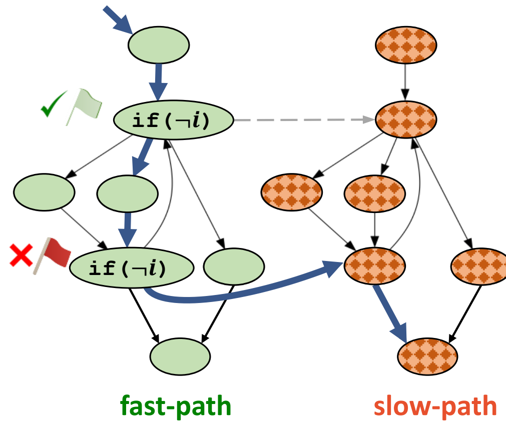


Figure 3.4: COPA forward recovery switching mechanism: Each function implements fast-path and slow-path in separate control flow domains, and execution switches from fast-path to slow-path upon detecting an invariant violation.

ensure a safe switch. At the time of the switch, the return addresses on the stack would be pointing to fast-path return sites. We address this problem by checking every return site, and transferring control to either the fast or slow path based on the current mode of execution.

Iodine conservatively disables all optimistic optimizations upon an invariant violation. Given adequate profiles for a rigorously tested production software, invariant failures are very rare. If there is indeed an invariant failure in production, the program can be re-optimized offline after removing the offending invariant from the likely-invariant set. Thus, in the steady-state, invariant violations would be extremely rare. Alternatively, only the optimizations induced by the violated invariant could be selectively disabled. Also, since it is common for live systems to be periodically restarted [35], the execution can switch back to the fast-path on a restart.

3.4 Iodine Implementation

We present an overview of the notable features of our tool here, and the details of its implementation are presented in [36, §5].

The Iodine tool consists of a profiler, a profile-driven predicated static analysis phase, and the optimized dynamic analysis instrumenter. These are implemented in the LLVM 3.9 compiler infrastructure [37], and we run our analysis tool after all other compiler optimization passes. Io-

dine supports programs written in the C language, tracks taint flows through external libraries via static linkage. The final optimized DIFT is added using LLVM's Data Flow Sanitizer[38] as our instrumentation backend.

Specifying Information-Flow Policies By default, Iodine uses a configurable taint policy that treats all types of external inputs to the program as potential taint sources (e.g. terminal, file, socket input functions, and command-line arguments) and asserts that the appropriate arguments to standard output interfaces (e.g. terminal, file and socket outputs) should not be tainted.

Useful taint policies can be specified to identify custom taint sources, sink locations, and untaint functions via a flexible interface of source-level annotations. This adapts the tool to evaluate security-critical applications with realistic information flow policies– e.g. the Postfix mail server with policies to check for email integrity and privacy, and the Nginx web server with detection against malicious overwrite attacks.

Static Taint and Pointer Analysis Static taint analysis computes how the taints of data propagate through the program under a given selection of taint sources, sinks, and propagation policies. The static taint analysis uses a whole-program context-sensitive flow-sensitive data-flow may-analysis [16] to constructs a inter-procedural definition-use graph (DUG) [39]. To track taint flows via indirect memory operations to aliased locations, we compute the points-to set of each pointer location and then use this information to add taint-flow edges to the DUG from pointer definition to its aliasing uses.
Once the DUG is constructed, the analysis can induce two optimizations.

Forward optimizations: Taints are propagated through the whole-program DUG using forward data-flow until a transitive closure is reached. Since our dataflow analysis is a *may* analysis, the absence of taint flow is a sound *must not* assertion. Therefore, any instruction without tainted source operands can elide dynamic monitors for taint tracking.

Backward optimizations: Taint flows that do not eventually reach a sink can be pruned out using a backward co-reachability analysis on the DUG. These optimizations are only enabled in the conservative static analysis.

Predicated Static Taint Analysis To improve the precision of the static taint analysis, Iodine profiles three types of likely invariants – *likely unreachable code*, *likely callee set*, and *likely unrealized call contexts* [1]. By assuming these likely invariants, the DUG constructed for static analysis is much smaller, thereby improving scalability and accuracy of our optimistic pointer and taint analyses. Iodine implements predicated versions of pointer analysis and forward data-flow analysis, and the backward data-flow analysis is not predicated as required in §2.3.

Optimistic Hybrid Taint Analysis The predicated static taint analysis identifies the set of instructions that need to be monitored. DIFT monitors for only these instructions are then instrument using LLVM DFSan [38], effectively eliding the remaining `noop` monitors.
Metadata tracking monitors track taints for each program variable and memory locations at the byte-granularity in separate taint data structures in a shadow memory, and we only consider explicit taint flows [26]. Iodine can track a single logical taint as well as multiple taint tags per location.
Invariant checks for all used invariants are added to detect if an invariant is violated dynamically.

3.5 Evaluation

We compare the performance of our approach with conservative hybrid and state-of-the-art full dynamic taint tracking [38]. Dynamic taint tracking incurs $7\times$ overhead over native execution, and hybrid analysis-optimized taint tracking incurs 37% overhead. Our optimized taint tracking tool brings down the overhead of dynamic taint tracking to 9%. Our evaluation shows that Iodine:

- Enables production use of taint tracking by dramatically reducing the overhead of taint tracking compared to conservative hybrid analysis and pure dynamic analysis.
- Efficiently implements real-world information-flow policies for security-critical applications.
- Requires reasonable profiling efforts. We show regression tests are adequate to get majority of the performance benefits.
- Improves the precision and scalability of static taint analysis.

Experimental Setup

We evaluate Iodine over several security-sensitive real-world applications, including web servers, mail server, database server, and utility programs. Our benchmarks are listed in Table 3.1.

We test Iodine in a manner that parallels how we envision it will be used in practice. We first profile a set of profiling executions to gather likely invariants. Then, we use these profiled invariants in a predicated static analysis to construct our final optimized dynamic taint analysis for a given information-flow policy. We generate a set of 500 diverse profile inputs by sweeping the programs' parameter space (e.g., data size, #clients, #requests, compression factor, etc.; excluding standardized parameters, e.g., TCP/SMTP port). We randomly partition these inputs into two disjoint sets- a *profile set* consisting 400 executions, and a *performance test set* of 100 executions. We note that in an actual production environment the profiling overhead of Iodine would be amortized over all future executions of the program, not just the 100 we test.

All experiments are run on a single core of an Intel Xeon E5-2620 processor with 16GB RAM running Linux 4.4.

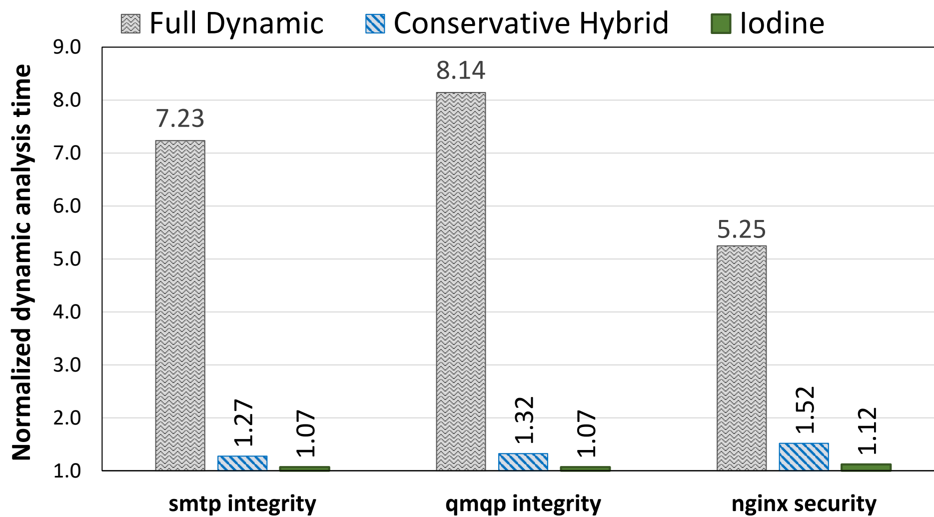


Figure 3.5: Result: Dynamic information-flow tracking overheads

3.5.1 IFT Security Policies

We demonstrate the effectiveness of Iodine using real taint policies by applying it to a set of commonly used applications with realistic taint policies adapted from Dytan [26] and Google desktop's privacy policy [40]. The policies we study are:

Email integrity and privacy: We add security checks to the Postfix mail server, following the policies outlined in [27, 40]. These policies ensure: receiver addresses are entirely determined by user input and message dates are only determined by the `time` system call (email integrity), and message bodies are passed through sanitizing functions that perform encryption, and check for unmatched HTML tags or scripting tags (privacy + security).

Overwrite attacks on web server: We enforce a taint policy on the Nginx web server that taints all network inputs, and asserts that tainted values are not used as function pointers, return addresses, or format strings. This policy detects a malicious overwrite attack [26].

Results: Iodine shows a $4.4\times$ reduction in runtime overhead for these realistic case studies, incurring only 7% to 12% overhead, compared to 27% to 52% obtained with conservative hybrid analysis. These results are shown in Fig. 3.5, as well as those of a naive dynamic IFT analysis. With these significant runtime improvements Iodine enables taint tracking in many production systems where performance concerns often preclude security.

3.5.2 Generic Information-Flow Policies

We further test Iodine's effectiveness in reducing taint overhead over additional benchmarks by using two synthetic taint policies to evaluate the effectiveness of our framework in a forward-only analysis versus a forward-backward analysis.

Some-to-some: Propagates taints from a randomly sampled fraction of the taint sources to the set of all sink instructions. This uses both forward and backward static taint analyses.

Some-to-all: Treats all instructions as potential sinks and propagates taints from the sampled taint sources. Only forward static taint optimizations are used to optimize this analysis.

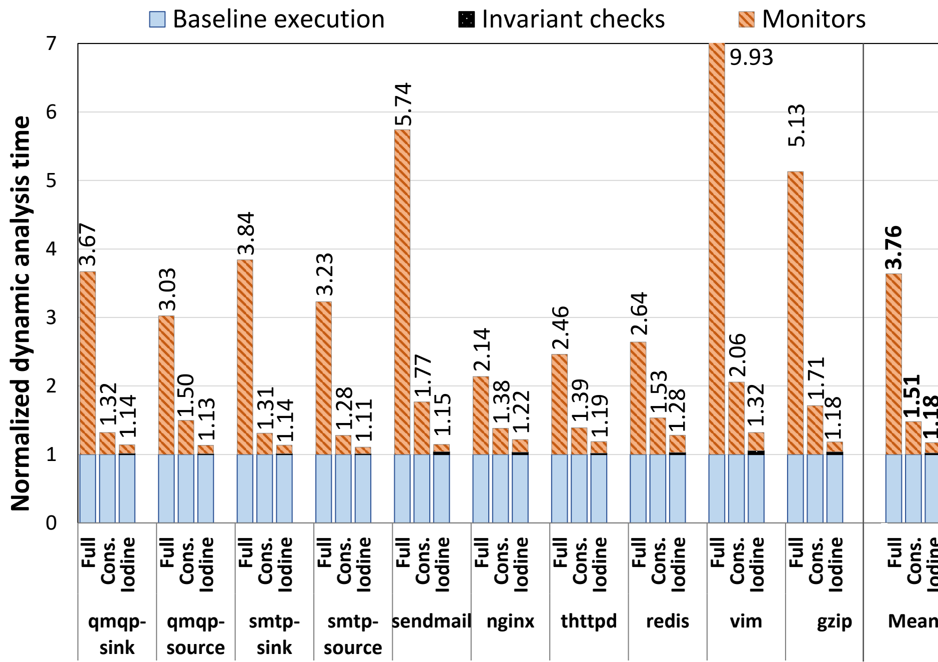
Some-to-all taint policies are useful in many non-security contexts such as database provenance

and lineage queries, information flow in debugging and software testing. This optimization also isolates the forward optimizations of our hybrid IFT framework, showing directly how effective predicated static analysis is at optimizing taint checks versus a sound static analysis. We treat all input interfaces from console/file/network as potential taint sources and elect to randomly sample $\frac{1}{3}$ of them for these taint policies. All output interfaces to console/file/network are taint sinks.

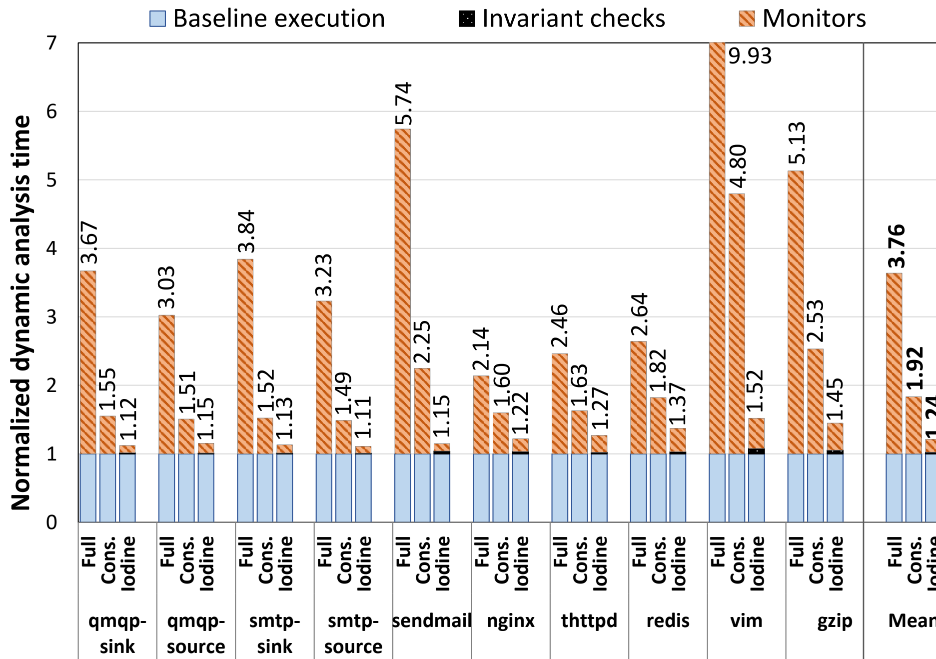
Results: When applied to some-to-some taint tracking (Fig. 3.6a), Iodine reduces the dynamic overhead of conservative hybrid taint analysis by $2.8\times$, bringing the overhead of taint tracking from 51% with conservative hybrid analysis down to 18% over native unmonitored execution. Iodine sees similar reductions in some-to-all tracking overhead (Fig. 3.6b), reducing overhead of taint tracking to 24%, versus 92% for conservative hybrid analysis, and 276% for a pure dynamic analysis. Once again, Iodine brings overheads down significantly, demonstrating its capability to enable taint-tracking on production systems.

SPEC benchmarks: To further evaluate Iodine's performance on compute-intensive programs, we run it with the same randomized some-to-some analysis setup on the SPECint benchmarks that are written in C with reference inputs. The results of these experiments are shown in Fig. 3.7. The SPECint benchmarks are tuned to be CPU bound, and therefore exhibit higher DIFT overheads compared to our other case studies. Iodine improves the dynamic overhead of taint analysis by $4.5\times$, bringing the overhead of taint tracking over unmonitored execution from 183% with conservative hybrid analysis down to 41%.

Comparison to ideal analysis: We construct an optimal analysis that only monitors instructions that are dynamically found to propagate taint, the very minimum set of instructions a some-to-all analysis could gather. We measure the average dynamic overhead of this ideal some-to-all taint analysis to be 13%. This shows that at 24% overhead, Iodine is 86% closer to optimal than traditional hybrid's 92%, and beginning to approach the realm of optimal dynamic taint analysis.



(a) some-to-some taint analysis



(b) some-to-all taint analysis

Figure 3.6: Result: Iodine compared to pure (full) dynamic and conservative hybrid DIFT.

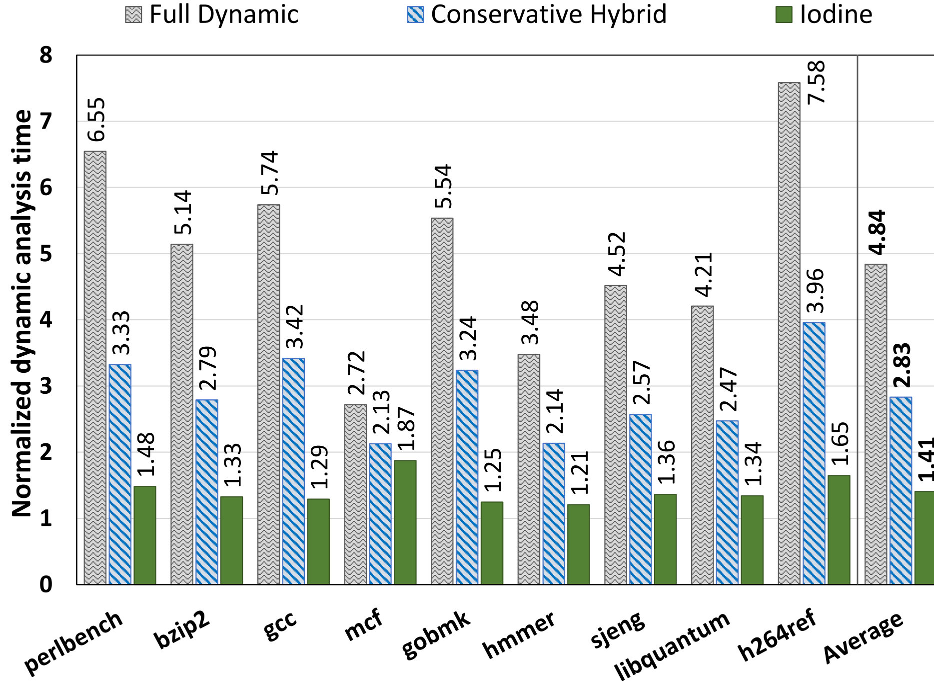


Figure 3.7: Result: Taint tracking performance on SPECint C benchmarks

3.5.3 Memory Overheads

Iodine maintains the exact metadata state as a conservative analysis. Therefore, the memory space overhead of metadata tracking remains unchanged. Iodine does increase code-size by generating two versions of the code: the fast-path and slow-path. However, as only one version of the code is executed at a time, this has little impact on the caching behavior or performance of the program. On average, the code footprint of a program instrumented by Iodine increases by $2.1\times$, compared to $1.4\times$ with conservative hybrid taint analysis, and $1.8\times$ with pure dynamic taint analysis.

3.5.4 Iodine's Framework Overheads

Invariant Check Cost: Fig. 3.6 also isolates the invariant checking costs. Invariant checks are only required in Iodine's optimistic analysis framework and are absent from the full dynamic and conservative hybrid analysis. Overall we observe that invariant checks have nearly no effect on end runtime, incurring only 2% of overall execution time.

Invariant Violations and Switching Overhead: Overall Iodine observes largely inconsequential rates of invariant violations, with only `sendmail`, `redis` and `vim` violating an invariant during some-to-all analysis in 3, 2, and 5 (out of 100) executions respectively. This indicates that our profiling methodology captures the common-case dynamic execution behavior effectively, significantly optimizing the dynamic analysis. The amortized overhead of the slow path analysis resulting from these violations is less than 0.5%. Note that the slow-path overhead can be no worse than that of conservative hybrid analysis.

We also find that the runtime overhead of the switching mechanism at function call return sites is negligible.

3.5.5 Precise and Scalable Static Analysis

Fig. 3.8 shows how assuming different types of invariants successively reduces the number of required static monitors for a some-to-some taint analysis. While the conservative static analy-

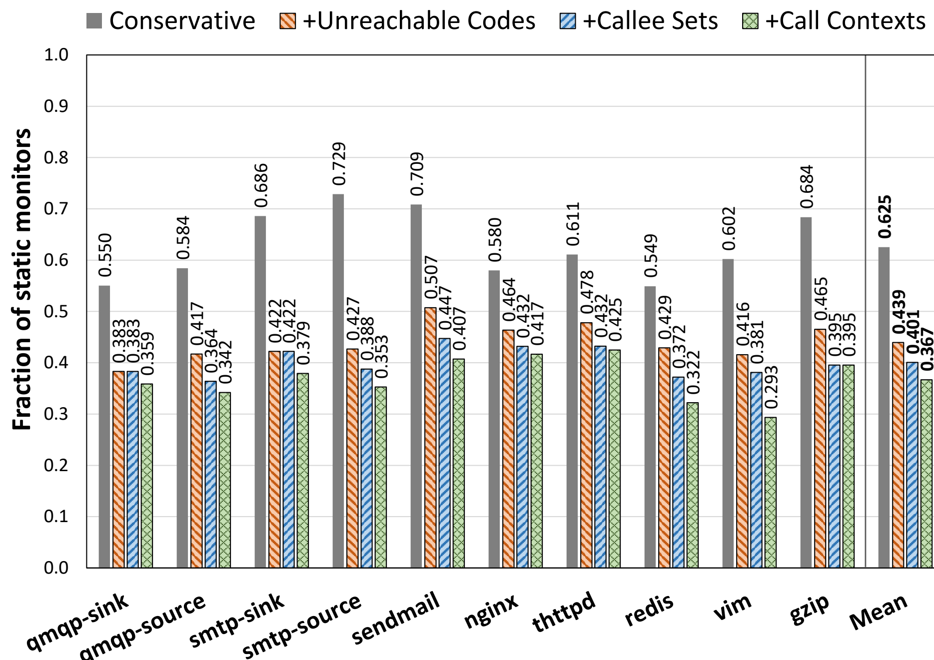


Figure 3.8: Result: Improved static taint analysis precision by assuming different invariants– Conservative some-to-some analysis uses a context-insensitive pointer-analysis, while the predicated analysis can scalably apply a context-sensitive pointer analysis.

Table 3.1: Static analysis time breakups for Iodine's some-to-some taint analysis

Benchmark	Conservative Static Analysis			Predicated Static Analysis			
	Points-to	Taint	**Total**	Profiling	Points-to†	Taint	**Total**
qmqp-sink	8s	4m 28s	4m 36s	1m 19s	12s	36s	2m 07s
qmqp-source	7s	14m 18s	14m 25s	1m 45s	5s	1m 12s	3m 02s
smtp-sink	9s	6m 12s	6m 21s	2m 00s	16s	44s	3m 39s
smtp-source	11s	11m 44s	11m 55s	2m 19s	9s	1m 08s	3m 35s
sendmail	15s	16m 53s	17m 08s	2m 02s	13s	1m 37s	4m 32s
nginx	19s	20m 04s	20m 24s	1m 12s	12s	1m 30s	2m 54s
thttpd	18s	17m 54s	18m 12s	59s	16s	1m 14s	2m 29s
redis	1m 18s	19m 43s	21m 01s	2m 01s	10s	1m 25s	3m 35s
vim	32s	61m 22s	61m 54s	5m 12s	88s	2m 54s	9m 35s
gzip	8s	8m 49s	8m 58s	7m 03s	17s	1m 22s	8m 42s

†Our optimistic framework enables us to scalably apply more accurate context-sensitive points-to analysis during the predicated static analysis

sis requires instrumenting 63% instructions on average, our predicated static taint analysis nearly halves this value at 37%, providing the foundation for Iodine's performance results. This translates to eliding 54%(nginx)-86%(vim) of the dynamic taint checks from a conservatively optimized analysis.

Table 3.1 summarizes the breakdown of static analysis times for both the conservative static and our predicated static versions. Applying the invariant assumptions to constrain the static analysis search space enables us to scalably apply a context-sensitive pointer analysis. This further improves the precision of our predicated static analysis. We see that a reasonable effort spent in profiling significantly reduces the overall static taint analysis time. In fact, the total static analysis time including the profiling time is lower than that of conservative static analysis for all our test programs. This makes Iodine suitable for deployment in production where the applications are constantly evolving thereby requiring re-analyzing them statically for hybrid analysis.

3.5.6 Profiling During Regression Testing Is Effective

An important concern with profile-based optimizations is the time and effort spent in profiling as well as the system's sensitivity to the profile set. We observe that software regression tests seek to maximize code and path coverage, and are therefore good candidates for conservative profiling.

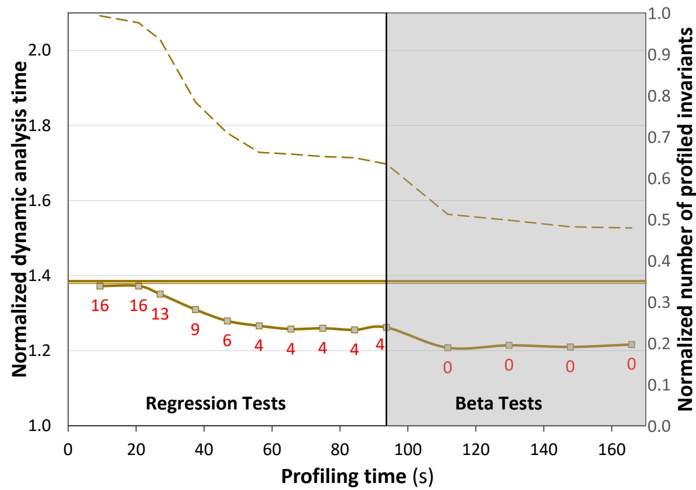
We evaluate this approach by profiling three programs- `nginx`, `redis` on their packaged regression test suites, and `vim` on open-source test suites [41, 42]. The results in Fig. 3.9 show that profiling on regression test suites alone is very effective. It reduces the runtime overhead to 31% compared to 55% with conservative hybrid analysis. We however observe invariants being violated dynamically after this profiling, and so recommend further profiling on beta tests. Profiling on the beta tests (shaded right halves) reduces the invariant violation rate significantly and brings down the analysis overhead to 23%.

Thus, we leverage the existing software testing suites to perform Iodine's initial profiling, and recommend reasonable beta testing for learning invariants to optimize Iodine. Moreover, Iodine is resilient to weak profiling. Our analysis needs no guarantees that all states are profiled; and even if the invariants fail dynamically, the constructed optimized analysis is still sound. Failing invariants can be learned over time and the optimized analysis can be adaptively re-constructed to exclude those without requiring analysis rollbacks. Iodine requires test suites with reasonable coverage for profiling, and is moreover resilient to profiling inaccuracies.

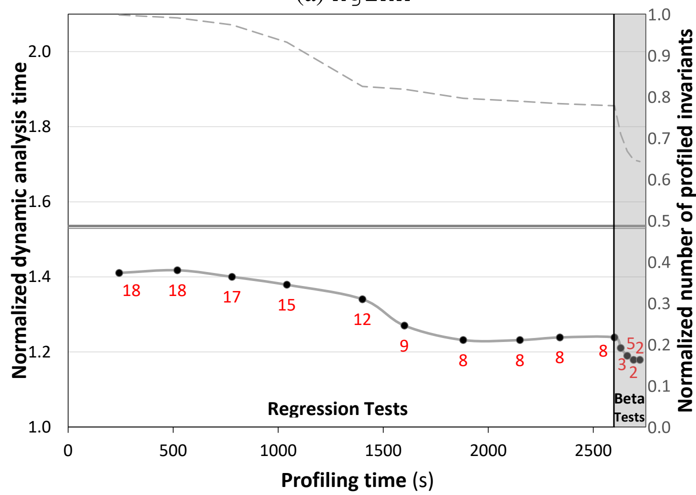
3.5.7 Sensitivity to Fraction of Tainted Data

Hybrid analyses (both traditional and Iodine) elide instrumentation that cannot propagate taint. As a growing set of inputs carry taints, the taints spread faster to nearly the program's entire data space. If nearly all data is tainted, there is no optimization opportunity and Iodine fails to effectively elide taint checks. To investigate this behavior, in Fig. 3.10, we look at how Iodine's normalized runtime varies with increasing the taint sampling fraction in our some-to-all taint analysis in §3.5.2. We statically identify all viable taint sources (input interfaces from console/file/network) and randomly sample the stipulated fraction of them to be active. Since selected sources might vary in their dynamic execution frequencies, we run on 100 different samples for a given sampling fraction (except for 100%).

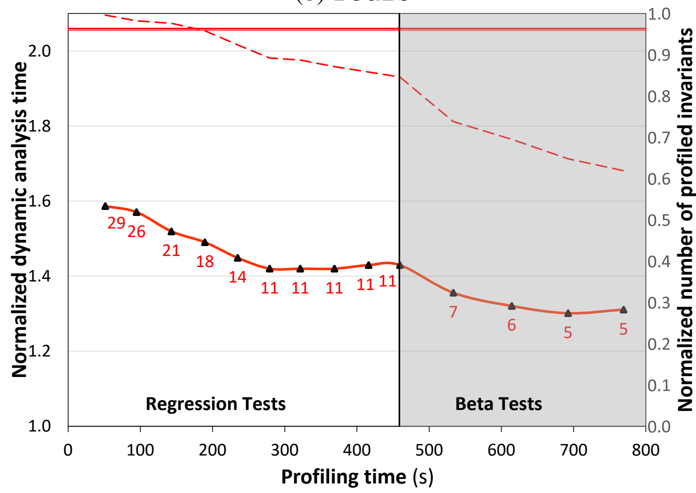
As expected, we observe that Iodine's performance degrades in general when dealing with larger fraction of tainted inputs, although Iodine shows significant benefits for many realistic levels



(a) nginx



(b) redis



(c) vim

Figure 3.9: Result: Profiling invariants while software testing– Profiling is done in two phases- first on regression test suites (left unshaded), and then on beta tests (right shaded). The solid marked lines plot analysis overheads with Iodine using invariants gathered at different stages of profiling. The numbers labeled on the plot indicate the number of dynamic invariant violations. The horizontal solid lines representing conservative hybrid analysis are an upper bound to Iodine's overheads. The dashed lines against the secondary (right) y-axis plot the number of invariants used normalized to that after profiling a single execution.

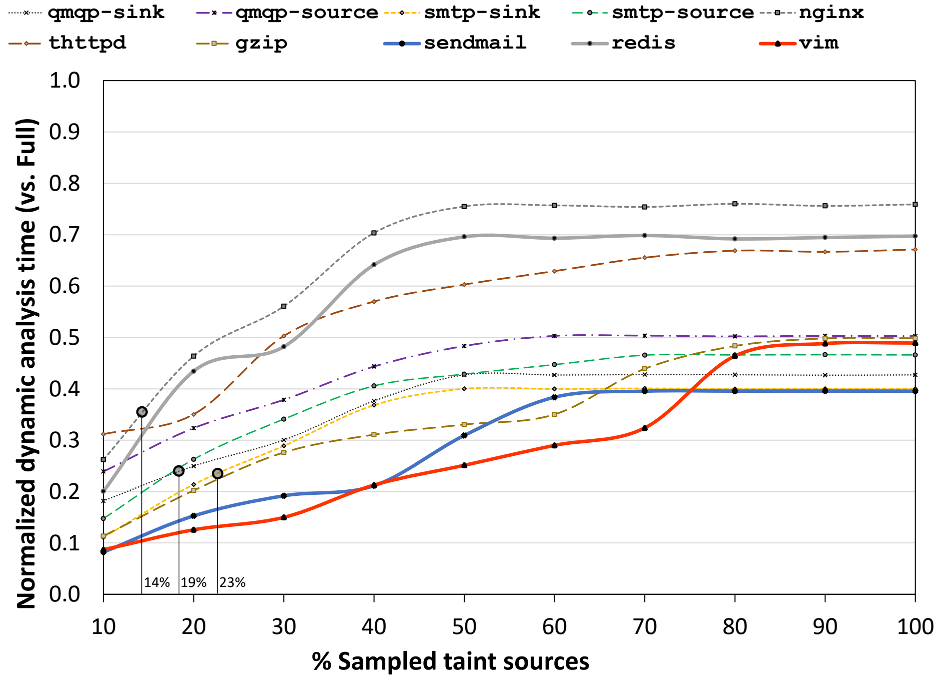


Figure 3.10: Result: Iodine's sensitivity to fraction of tainted data – performance benefits reduce with larger fractions of the program's data space being tainted. Fraction of taints observed for realistic taint policies (§3.5.1) are annotated.

of tainted input. This behavior is fundamental to hybrid analysis, and is no worse in Iodine than in a conservative hybrid analysis. Iodine is effective when the target program and the taint policy induce a low fraction of tainted data. We observe that this property indeed holds for the IFT security policies studied in §3.5.1; the static fraction of active taint sources therein are between 14-23% (circled in Fig. 3.10).

3.6 Related Work

Iodine builds on the prior optimistic hybrid analysis work [1] in two major ways- (1) it constructs a rollback-free OHA by limiting to only *safe elision* optimizations thereby solving the recovery problem in OHA, and (2) applies this novel technique to realize a low overhead DIFT solution for live executions. Below, we discuss relevant prior work on DIFT, hybrid program analyses, and profile-based optimizations.

Dynamic Analysis

There has been significant work on dynamic taint tracking systems [26, 43, 23]. Past work has developed many optimized dynamic techniques, such as creating highly specific information-flow policies [25, 20, 19], reducing its scope to only apply to related processes [44], optimizing low-level taint operations [45], writing minimal emulators targeted for taint tracking [30], or even providing custom hardware support [24, 46, 47, 48]. All of these optimizations operate purely on the dynamic state of the program, attempting to make existing set of taint operations faster. Iodine elides taint operations through static analysis, reducing the set of instructions monitored, making its optimization complementary to these prior approaches.

Taint tracking has also been parallelized either by partitioning the execution into epochs to perform local analysis and then aggregating results [49, 50], or by decoupling taint analysis from the program execution [51, 28, 29, 52], wherein the dynamic instrumentation only performs lightweight logging followed by an offline analysis. These efforts reduce latency of taint tracking through parallelization, but not overall work, like Iodine does. They too are complimentary to Iodine's optimizations.

Static Analysis

Several systems have attempted to solve taint tracking using language features to enforce a taint policy at compile-time, sometimes with limited dynamic checks [53, 54]. These systems achieve low runtime overhead, but place the burden on the programmer to specify and guarantee taint policy using an unfamiliar restrictive language. Iodine optimizes dynamic analysis, and does not require source code changes, other than trivial annotations specifying taint sources, sinks, and untaint functions.

Hybrid Analysis

Hybrid analysis has been explored in the past [55] for accelerating DIFT. Moore et al. provide the soundness conditions for static analysis to determine when it is safe to stop tracking certain vari-

ables dynamically [14]. In addition to removing unnecessary monitors using static analysis, Chang et al. statically transform untrusted programs into policy-enforcing programs to further reduce the amount of data to be tracked dynamically [5]. Jee et al. statically separate the taint tracking logic from the program logic and then optimize it using abstract taint flow algebra [28]. Hybrid systems have also coalesced taint checks through static analysis [18, 56]. While these traditional hybrid analyses use sound static analysis to conservatively reduce dynamic overheads, Iodine further improves runtime overheads with use of unsound, predicated static analysis. Iodine's use of optimistic hybrid analysis with forward recovery could likely be combined with these systems for further taint optimizations.

Blended analysis [8] uses dynamic information to improve the accuracy of a best-effort static taint checking tool for JavaScript applications [11]. While they utilize dynamic information to make static analysis tractable for corner-case dynamic language features (e.g., `eval`), our likely invariants captures common program behaviors to improve whole-program static analysis. Moreover, their end goal is just to improve static analysis, and stop short of optimizing dynamic analysis. They also do not provide soundness or completeness guarantees for any results produced. Iodine produces sound and complete dynamic analysis for live executions.

Profile-guided Compiler Optimizations

Profile-guided optimizations [57, 58] learn invariants through profiling and use them for local optimizations. In particular, work on JIT optimizing compilers such as those that speculatively inline functions [59], or speculatively convert indirect function calls to direct function calls [60], speculatively optimize execution, as done in Iodine. Our work differs in two key ways. First, while compiler optimizations focus on optimizing program logic, Iodine aims at eliding unnecessary runtime DIFT monitors. A more fundamental difference is that Iodine uses invariants to improve precision and scalability of whole-program static analysis. In contrast, profile-guided optimizations do not typically consider whole-program static analysis, and therefore the methods for checking invariants and recovery are simpler and cheaper than optimistic hybrid analysis.

Summary

Iodine presents a novel Cautiously Optimistic Program Analysis (COPA) technique to optimize DIFT. We solve a key challenge in applying COPA to online analyses on live executions – rollback recovery. We eliminated the need for rollbacks by restricting our predicated static analysis optimizations to `noop` *safe elisions*. Iodine significantly improves the precision of static data-flow and pointer analysis, thereby drastically reducing DIFT overhead for important security policies to 9%.

CHAPTER 4

PROV-GC: Provenance-based Sound Garbage Collection for C

Garbage collection (GC) support for unmanaged languages can reduce programming burden in reasoning about liveness of dynamic objects. It also avoids temporal memory safety violations and memory leaks. *Sound* GC for weakly-typed languages such as C/C++, however, remains an unsolved problem. Current value-based GC solutions examine values of memory locations to discover the pointers, and the objects they point to. The approach is inherently unsound in the presence of arbitrary type casts and pointer manipulations, which are legal in C/C++. Such language features are regularly used, especially in low-level systems code.

In this paper, we propose Dynamic Pointer Provenance Tracking to realize sound GC. We observe that pointers cannot be created out-of-thin-air, and they must have provenance to at least one valid allocation. Therefore, by tracking pointer provenance from the source (e.g., `malloc`) through both explicit data-flow and implicit control-flow, our GC has sound and precise information to compute the set of all reachable objects at any program state. We discuss several static analysis optimizations, that can be employed during compilation aided with profiling, to significantly reduce the overhead of dynamic provenance tracking from nearly $8\times$ to 16% for well-behaved programs that adhere to the C standards. Pointer provenance based sound GC invocation is also 13% faster and reclaims 6% more memory on average, compared to an unsound value-based GC.

4.1 Enforcing Memory Safety is Challenging

Unmanaged languages such as C/C++ are the languages of choice for a vast set of large, complex, ubiquitous, and critical software bases, such as- Linux, OpenSSL, Nginx, Redis, and these languages continue to be popular among many developers. Unmanaged languages require programmers to explicitly allocate and free memory space. This requirement not only increases pro-

gramming burden, but is also a source of common classes of bugs: use-after-free and memory leaks. Use-after-free bugs are not just a reliability issue, but a significant source of security vulnerabilities in modern systems [61], as they compromise temporal memory safety [6]. In spite of significant advancements, prior solutions for temporal memory safety incur prohibitive performance overheads ($\sim 60\%$ [6, 7, 62]). Memory leaks also compromise system reliability and can cause unpredictable performance [63]. Prior solutions have tried to address memory leaks through a combination of offline bug detectors [61, 64], and runtime systems that probabilistically repair these bugs [65, 66, 67].

Replacing manual deallocation of memory in unmanaged languages with a sound and efficient garbage collector (GC) would address all of the above problems by guaranteeing temporal memory safety, avoiding memory leaks, and reducing programmer burden. Unfortunately, a sound GC for weakly-typed languages like C/C++ has remained elusive.

A sound GC is one that guarantees to not free an object that is accessed later [1]. Typically, a sound mark-and-sweep GC [68] automatically reclaims memory at runtime by freeing a set of objects that can be guaranteed to be unreachable from a set of "root" pointers (pointers in global variables, stack variables, and registers). To compute this set at runtime, given a pointer, a GC should be able to (1) identify a pointer's *dynamic points-to* object, that is, the object reached by dereferencing the pointer, and (2) locate all the pointers contained in that reachable object. We refer to the latter set as the *pointers-within* set for an object. In strongly-typed, memory-safe languages like Java, both of these operations are straightforward [69]. A pointer's value can be used to identify its dynamic points-to object due to spatial memory safety [70], and the pointers-within set of an object can be easily determined due to the strong type system.

C/C++, however, is weakly-typed. Pointer values can reside in, or be computed from, non-pointer variables, making it difficult to locate them within a reachable object at runtime. Even if we can locate all the locations with pointer values, they are not guaranteed to point within the referenced objects. This is true even in spatially memory safe programs, as a pointer value may

[1]Ideally GCs would free objects that *will not* be accessed later, but real GCs settle for the *cannot* be accessed approximation.

be arbitrarily transformed to point away from the object, then manipulated back just before a dereference. Such pointer manipulations are regularly used in low-level systems code [71].

Prior works on GC for C/C++ [72, 73] have tried to overcome some of these problems using *value-based* heuristics, but these works do not guarantee soundness. They assume that only memory locations with values within an allocated heap object's address range are valid pointers, and that the value points within the referent object. This assumption is *unsound* as they cannot identify the referent object when a pointer value goes out-of-bounds due to arbitrary pointer manipulations allowed in C/C++. They are also *imprecise*, and therefore, prone to memory leaks when non-pointer locations hold values that happen to be within the heap address range. Finally, they have to examine the value of every reachable memory location to determine if it is a pointer or not, leading to higher performance overhead.

We design the first *sound* GC for C/C++ called *Provenance-based Garbage Collection* (PROV-GC). We observe that a C/C++ program cannot create a pointer out-of-thin-air; instead, pointer values must be derived from a *valid pointer source*. Valid pointer sources are from *allocation functions* (e.g. `malloc`) and the *address-of* (`&`) operator. These pointer values subsequently propagate to other variables either through explicit data-flow or implicit control-flow. Thus, our key idea is to use dynamic information-flow tracking to soundly and precisely determine the set of all memory locations that hold values derived from pointers, and the object locations they point-to. Our mark-and-sweep GC uses this information to soundly reclaim unreachable objects.

Conventional dynamic information-flow tracking (DIFT), however, is known to incur significant performance overhead, slowing execution down by several times [26]. This is due to the need to execute a "monitor" typically for every instruction that could propagate a "taint". Furthermore, taint propagation through implicit control-flow (a necessity for us to ensure soundness) is known to be not only expensive in terms of performance, but also can imprecisely taint a significant fraction of memory locations [26].

We observe several optimization opportunities that we exploit using hybrid taint analysis [36] to realize a low-overhead DIFT for pointer provenance. A common property that our static analysis

exploits is that if an instruction destination operand's taint meta-data is guaranteed to not change, then the taint monitor for that instruction can be elided. This property holds true for instructions that are non-pointer operations (e.g., `inta = intb * 10;`), which is the vast majority of the instructions executed. We also show that if an instruction's destination operand is a statically declared pointer and its source operand is guaranteed to be a "safe" pointer (value points within the object), then its taint value is known at compile time, and therefore a runtime monitor is unnecessary (e.g., `int* ptr = safe_ptr;`). This category includes common pointer assignments, where the pointer has not been manipulated arithmetically. Finally, if the source and destination operands of an instruction are the same (e.g., `ptr = ptr+4;`), then there is no need to update the destination's taints.

Somewhat surprisingly, tracking implicit information flows, known to be intractable in general, turns out to be practical for pointer provenance. Conditional branches dependent on pointer variables can propagate taint implicitly to its control-dependent instructions. While we do not expect reasonable programmers to use such programming constructs, to realize sound GC, we must consider its possibility, as they are legal in C/C++. Fortunately, we are able to show that when branch conditions are based on comparisons between in-bounds pointers or with `NULL` value, there is insufficient implicit information flow to require dynamic tracking. While we need modest dynamic checks to establish that pointer values are in-bounds, we never have to propagate implicit flow for the programs we studied.

Finally, to prove the above properties statically to aggressively elide dynamic taint monitors, we apply Cautiously Optimistic Program Analysis (COPA) [1, 36]. OHA uses profiled likely invariants to predicate the whole-program context-sensitive flow-sensitive static taint analysis.

The C standard [74] specifies several restrictions on using pointers and operations that can be performed on pointer types. We show that assuming these properties benefits our pointer provenance tracking significantly, and reduces the overall cost of GC for standard-compliant programs. We provide solutions with and without this optimization, because, in practice, there are many programs that regularly violate the standard [75], and as such they require provenance tracking without

optimizations that depend on these C standard specifications.

We evaluate our PROV-GC tool on several long-running large applications as well as memory-intensive benchmark programs. Unlike the Boehm-Demers-Weiser GC (BDW-GC) [73], PROV-GC is sound. For well-behaved programs that are C standards-compliant, we pay only an additional 16% average performance cost to dynamically track pointer provenance. Of which, 14% is due to explicit data-flow tracking, and the remaining is to track implicit-flow. We find that our optimistic optimizations that elide dynamic monitors are very effective with adequate profiling. Without them, we see nearly $8\times$ slowdown. In addition to soundness, compared to BDW-GC, the performance of our GC invocation is about 13% faster, as we avoid scanning, and reclaim about 6% more memory per GC invocation due to our GC's improved precision.

16% performance overhead of our GC is especially appealing as it obviates the need for a slower dynamic temporal safety solution (60% overhead [62]), besides reducing programming burden and avoiding memory leaks.

We make the following contributions in this paper:

- We present PROV-GC, a GC that is sound for all legal C/C++ programs. Previous GC solutions for C/C++ are unsound as they might free reachable objects.
- We present the idea of dynamic pointer provenance, and use it to realize a sound GC for C/C++.
- We show how we can elide taint monitors for a vast majority of instructions such as operations on non-pointers, "safe" pointers, etc.
- We show tracking implicit information-flow in the context of pointer provenance is necessary and practically feasible.
- We show how the C standard specifications induce a significantly improved provenance tracking solution for standard-compliant programs.
- We apply optimistic hybrid analysis [1] to optimize dynamic pointer provenance and realize an efficient GC that incurs 16% overhead. This overhead is much lower than dynamic temporal safety checking, and it avoids memory leaks and reduces programmer burden.

4.2 Garbage Collection for C/C++

We briefly discuss the motivation for using GC in weakly-typed languages like C/C++, and the unsolved problems in realizing a sound GC for them.

4.2.1 Why GC for C/C++?

GC obviates the need for manual memory management and thereby eliminates two common classes of bugs in unmanaged languages: memory leaks and use-after-free [63, 61]. Memory leaks and use-after-free bugs are considered important classes of bugs and thus have received significant attention from academia and industry, who have tried to address these bugs through a variety of methods. For memory leaks, there exists a number of offline debugging tools and runtime probabilistic methods to mitigate the ill effects of these bugs [63, 76, 77, 78, 79, 80, 81, 82]. We argue that GC for C/C++ would address the memory leak problem more comprehensively than these methods.

Use-after-free bugs are particularly important because they compromise system security. These bugs violate temporal memory safety [6], which along with spatial safety is necessary to ensure full memory safety. Recognizing the importance of memory safety, even commodity processors (e.g., Intel MPX) [83] have started providing specialized hardware support for efficiently implementing spatial memory safety checks. Spatial memory safety, however, solves only part of the problem. Efficiently guaranteeing temporal safety remains expensive, as state-of-the-art solutions incur $\sim 60\%$ performance overheads [62].

We argue that if we can realize a sound and efficient GC for C/C++, it would not only reduce the burden on future software development, but also help improve reliability and security of both future *and* legacy systems. If the execution time overhead of GC can be made lower than the overhead of other temporal safety solutions, then it certainly would be a superior solution, as it not only removes temporal errors, but also improves programmability.

4.2.2 GC and its Pointer Data Requirements

Identifying dead objects precisely at a given instant of a program's execution is hard as it depends on future execution. Therefore, current GCs conservatively identify live objects by assuming that the set of all "reachable" objects from a "root" set of pointers are live. The root-set consists of all pointers in the registers, global and stack address space. A reachable object can still be dead as it may never get referenced in future. Current GCs use either incremental mark-and-sweep [73] or reference counting [84, 85] to compute reachability. In this paper, we use mark-and-sweep, though our provenance-based approach could also be used by a reference-counting GC.

When a mark-sweep GC is invoked, it performs two separate steps: *marking* computes the live set of objects that are transitively reachable from a *root set* of pointers, and *sweeping* reclaims memory from unreachable objects. To perform the reachability analysis, the mark step requires two crucial pieces of information about pointers: a pointer's **points-to object** (PT), and an object's **pointers-within set** (PW). The points-to object of a pointer is the dynamic object that can be dereferenced using that pointer. The pointers-within set of an object is the set of all pointers that are contained within that object.

Both points-to and pointers-within data are straightforward to determine *soundly* and *precisely* in type and memory safe languages such as Java, Python, and C#. The points-to object of a pointer can be determined from its value due to spatial memory safety. That is, a pointer's value is guaranteed to be within the address range of its points-to object. Furthermore, due to type safety, given an object, it is possible to precisely and quickly determine its pointers-within set, because only variables that are typed as pointers can hold pointer data; non-pointer variables cannot hold pointer data. *Determining points-to and pointers-within data in weakly-typed languages like C/C++, however, is a significant challenge as we discuss next.*

4.2.3 Value-based GCs for C/C++ is Unsound

All prior attempts to provide GC for C/C++ use *value-based* heuristics to compute points-to and pointers-within information. For example, the best-known such work [73] computes the pointers-

```c
typedef struct { uintptr_t val, xptr; } xorlist;
xorlist *head = NULL, *tail = NULL;
void traverse(xorlist *start) {
  xorlist *prev = NULL, *curr = start;
  while (curr) {
    printf("%ld\n", curr->val);
    uintptr_t next = (uintptr_t)
                        prev ^ curr->xptr;
    prev = curr;
    curr = next;
  }
}
```

```c
void main() {
  insert(...);
  ...
  traverse(head);
  traverse(tail);
}
```

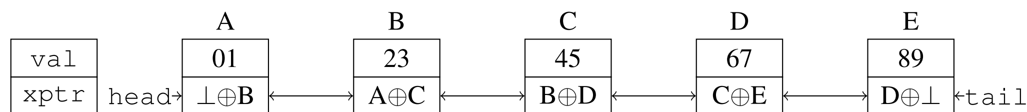


Figure 4.1: XOR linked list – Doubly Linked Lists can save space by storing the XOR of previous and next node pointers in a single integer location; **uintptr_t** is sufficiently long to hold pointer values. The inner nodes never store literal pointer values, but have sufficient information to reconstruct valid pointers to its two adjacent nodes.

within set of an object by scanning every pointer-sized field in the object and checking if its value falls within the address range of any allocated heap object. If the check succeeds, then that field is assumed to be a pointer, and the points-to object for that field is assumed to be the allocated heap object whose address range includes that field's value.

Value-based GCs work by assuming that any allocated heap object that may be referenced in the future has at least one live register or memory location pointing to it at all times. This assumption may be violated, for instance, if the C program breaks any of the following three assumptions: (A1) Pointers are only stored in variables that are declared to be pointers or in sufficiently large integral type that can hold a pointer. (A2) If a memory location's value falls within the bounds of an allocated heap object then it *is* a valid pointer, or else it is a non-pointer. (A3) A pointer discovered through its value is assumed to point within its points-to object. Value-based GCs are **unsound** whenever any of the above invariants is violated.

In C/C++, even legal programs can violate these three assumptions because C/C++ has a weak type system and allows programs to store pointer values in integers and manipulate them in arbitrary ways. Fig. 4.1 shows an example of a XOR linked list, a clever representation of a doubly

linked list used in memory constrained embedded systems [86]. Each node stores the XOR of pointers in the two directions and recovers them using XOR operations during traversal. Note that none of the inner nodes store literal values of pointers, but have information encoded to reconstruct two valid pointers. A purely value-based GC approach can incorrectly reclaim objects pointed to by such *hidden* pointers, e.g. node C's address is not stored literally anywhere in the program state. This is a legal C program which can break the correctness of a value-based GC. More extreme examples are also possible via casting and other manipulations; for example, a program may split a pointer into several smaller integers then reconstruct the pointer later; this would violate all three assumptions.

Value-based GCs can also be **imprecise** because they may think a non-pointer is a pointer when its value happens to lie within the heap address range. If this non-pointer's value points-to an unreachable object, then GC would avoid reclaiming it. This can lead to memory leaks and lower performance. There has been follow-up work [87, 88] that addressed this problem by adding another assumption that pointers only reside in declared pointer typed variables. This approach achieves greater precision, but sacrifices even more soundness, as it would ignore an integer value derived from a pointer through a cast.

Value-based GCs can also incur significant overhead while scanning the state space for pointers. Given a reachable object, GC has to scan each of its fields, and check if it could be a valid pointer or not. The check involves looking into a data-structure that maintains the address ranges of all heap objects.

In this work, we improve upon pure value-based GCs by using dynamic pointer provenance to soundly and precisely determine the set of all pointers and the objects they point-to. Also, this can quickly identify pointers-within set of an object without scanning each field, improving GC performance.

4.2.4 Need for Sound GCs

Guaranteeing correct GC behavior– i.e. objects reachable from the root set of pointers will not be freed, is important for *all* programs in the same way that it is important to have a sound compiler or runtime. Value-based GCs impose additional restrictions, as seen in §4.2.3 earlier, making them work correctly only on a subset of the language. These language properties might not be followed by legacy programs, and can be generally difficult to verify for new programs and compiler implementations that strive to conform to the standards but still may not. Moreover, new programs also reuse existing library code. So, there is value in supporting sound GC behavior for all programs without imposing additional restrictions on the language itself.

4.3 Provenance-Based Garbage Collection

We address these problems by designing a sound and efficient *provenance-based* garbage collector. We will first motivate how a provenance-based garbage collector solves the soundness issue, and outline a simple-but-sound strawman GC. Then, we discuss optimizations to reduce the overhead of our strawman GC, leading to a provenance-based GC that is both fast and sound.

We assume that, aside from temporal safety errors, the given program is a valid C/C++ program and obeys the properties necessary for the compiler and hardware to guarantee well-defined behavior. This includes spatial memory safety [70] and data-race-freedom [89], which many prior works have addressed.

4.3.1 The Soundness of Provenance-Based GC

We propose that instead of using the *value* of a pointer to identify its points-to set, a GC can use the *provenance* of the pointer to soundly derive its dynamic points-to set.

The soundness of our provenance-based solution is based on the assumption that allocated heap memory addresses cannot appear *out-of-thin-air*. That is, without knowing the return value of an allocation function (e.g. `malloc` in C), it is impossible for the programmer to compute

the address of any dynamically allocated heap object. This assumption is true of most real type-unsafe languages, including C. Given this assumption, any well-behaved program must ensure that any heap addresses dereferenced by a load or store operation are derived from the return of heap allocation functions. Consequently, an object allocated within the heap is only reachable in the future if it has one or more live register or memory values (henceforth *values*) which draw provenance from its allocation function's return value.

Since a heap pointer value cannot appear *out-of-thin-air*, and all pointers must have a provenance to at least one valid allocation, it is therefore *sufficient* to track all points-to sets for all pointers to reconstruct all currently reachable objects.

4.3.2 A Simple Provenance-Based GC

To show how pointer provenance can construct a sound GC, we present a simple, strawman design of a provenance-based GC.

Terminology: We refer to registers and memory locations simply as *location*. A *pointer* refers to a location whose value is directly or indirectly derived from one or more allocation return values. The *points-to set* of a location refers to the set of object allocations from which the pointer is derived. The points-to set is the complete set of objects that the pointer *may be* used to access in the future. A memory location with empty points-to set is not a pointer.

Our strawman GC will naively track the points-to set for every register or memory location (henceforth *location*) in an execution by constructing a map from each location to the set of heap object allocations its value is derived from.

To dynamically track the points-to sets of all pointers, we apply a standard dynamic information flow (DIFT) policy, treating all heap object allocations as sources and using both data-flow and implicit-flow taint tracking. More specifically, the program begins with each location's points-to set empty. Whenever an allocation function returns, we add that allocation to the destination location's points-to set. Thereafter, whenever the program modifies a location, the points-to set of that location will be updated to contain the union of the points-to sets of any pointers it depends

on. We note that for this analysis to be sound, when a location is modified, it must consider not only data-flow dependencies, but also any implicit control-flow dependencies as well (e.g., when branch conditions depend on pointers).

A properly constructed DIFT analysis [90, 31] will, by construction, ensure that the points-to set of each pointer is conservative. That is, if the pointer could be used to dereference an object in the future, that object will be within that pointer's points-to set.

In order to avoid scanning memory to locate pointers, we also maintain a pointers-within set mapping for each allocated heap object. The pointers-within set will logically contain the memory location of every pointer within the allocated heap object. The pointers-within set for an object can be trivially maintained by initializing the set to empty when the object is allocated, then adding a pointer to the set whenever such a pointer is stored to a location within the object.

Once the collection phase of the GC actually begins, our GC only needs to iterate through the root set of the program (any locations statically reachable - globals, or reachable from any stack frames and registers), add these locations to a temporary set called the working set. Then, the GC will iterate through the working set and for each object in the working set identify all pointers with the object's pointers-within set. The GC then adds any objects in the points-to sets of those pointers to both the working set, and a set of live objects. This process iterates until the live set does not change. Any object not in the set of live objects at the conclusion of the algorithm may then be reclaimed.

While this straw-man solution provides a sound GC, tracking provenance metadata through all operations and through implicit flows will typically be very expensive. Fortunately, most instructions operate on non-pointers. Also, in the common case, pointers stay within object bounds, and do not propagate pointer data to other locations through control flow. We leverage these properties to significantly reduce the amount of provenance tracking required to construct a sound GC.

4.3.3 Optimizing Explicit Provenance

The strawman system described earlier requires dynamically inserting a monitor (to propagate points-to set) on nearly every instruction within the program. This would result in very high overheads [26]. However, we observe that for a significant fraction of instructions, its runtime monitors do not change their destination operands' points-to (taint). We use static analysis to elide these dynamic monitors without losing soundness. This section discusses three optimizations that elide such redundant monitors: 1) eliding non-pointer tracking, 2) eliding "safe-pointer" tracking, and 3) eliding monitors for pointers with the same operands.

Non-Pointer Tracking Elision

Within C programs, the vast majority of computation operates on data which is not logically derived from pointer values. If we can statically prove that a location within the program has an empty points-to set, then that location is a non-pointer, and the dynamic run-time system need not dynamically track the points-to set of that location. This detection can be accomplished by using static information flow analysis [54, 36] to compute a sound may points-to set, then eliding dynamic points-to set operations on locations with an empty may points-to set. For the example in Fig. 4.2, we can trivially elide any provenance tracking for line 5 since neither of its operands have data-flow from any pointers, in fact they are constants.

Safe Pointer Tracking Elision

Next, we observe that the vast majority of pointers in C programs (1) have exactly one object in their points-to sets (singleton set) and (2) have a value within the allocated memory range of that object (in-bounds). We call these *safe pointers*.

Safe pointers are handled correctly by value-based GC. Because the pointer value can be used to dereference only one object, and since the pointer value is in-bounds, we can use its value to determine its object. For the same reasons, we do not need to track the points-to set of any location in the program we know is a safe pointer, as we can identify its points-to set from its value at collection time.

```
1  void explicit_flow() {
2   unsigned int n = 10, o = 1000;
3   obj* A = malloc_1(n*sizeof(obj));   // PT(A) = {malloc_1}
4   char** B = malloc_2(n*sizeof(obj));// PT(B) = {malloc_2}
5   long z = o / n;      // elided by E1
6   char* p = A;         // elided by E2
7   long d = B - A;      // PT(d) = {malloc_2, malloc_1}
    ...
8   for (unsigned int i = 0; i < n * sizeof(obj); i++) {
9     char* q = p + d; // PT(q) = {malloc_1, malloc_2},   elided by E2 and line 10
10    *q = p;
11    p = p + 1;         // PT(p) = {malloc_1}, elided by E3
12  }
13 }
```

Figure 4.2: Explicit pointer provenance propagation

To perform this optimization, we perform a static data-flow analysis to identify which instructions in a program must define safe pointers. While statically identifying safe pointers precisely in a program is hard, we construct a sound but imprecise data-flow analysis as follows. A pointer defined by the assignment from an allocation function is clearly safe. Assignment from a safe pointer is also safe. The result of any operation is safe, provided it satisfies two conditions: (1) the operation has only one pointer operand, and that operand is a safe pointer, and (2) the operation is guaranteed to not modify the pointer to point outside the bounds of the object it references (provably in-bounds).

Leveraging Dereferences Our static safe-pointer identification methodology is conservative, and consequently will falsely identify many safe-pointers as non-safe. It is sound, but potentially introduces unnecessary dynamic checks. To help identify additional sources of safe pointers, we observe that any time an address is dereferenced, it must be an in-bounds pointer, otherwise the program would have undefined behavior (violating spatial memory safety). If we can additionally prove that the dereferenced pointer's points-to set is a singleton set, then we know the pointer is a safe pointer. To accomplish this, we construct another static taint analysis, with the goal of identifying singleton taint sets. To construct such an analysis, we observe that a points-to set can only

propagate from a singleton-set to a non-singleton-set when it depends on multiple pointers. Therefore, a static analysis can determine pointers that must have singleton points-to sets by checking if its transitive dependency set contains no operations with multiple pointer dependencies. We leverage this must-have singleton-points-to sets analysis with our observation about dereferenced pointers being in-bounds to identify an additional source of safe-pointers: dereferenced pointers with singleton points-to sets.

Equivalent Points-to Propagation

Our third optimization exploits the fact that many pointer redefinitions do not change the points-to metadata, and therefore they can be elided. This is trivially true when the source and destination pointer operands are the same. For the example in Fig. 4.2, the provenance (points-to) of `p` on line 11 cannot change. It is possible for arithmetic operation on a pointer to result in an out-of-bounds value. But in a well-defined (spatially memory safe) program, it cannot be dereferenced before it is reverted back to be within bounds. We use a static data-flow analysis that elides the monitor for an instruction when it can be proven that the provenance of its destination operand is same as its source either directly or transitively through data-flow.

A related optimization is that, if the provenance of a location remains constant within a loop, our analysis hoists it out of the loop through a loop invariant code motion [91].

In summary, we can elide provenance tracking operations when –

E1 All source operands have empty provenance.

E2 The resultant pointer is *safe*.

E3 The resultant pointer is assigned to the same identifier as the source operand, directly or via temporaries.

4.3.4 Optimizing Implicit Provenance

Although rare, it is both possible and legal for weakly-typed programs to deconstruct and reconstruct pointers through implicit flow operations, as shown in Fig. 4.3. Traditionally, implicit information flow DIFT is known to have severe limitations as the majority of locations can get tainted, and doing so, as proposed in our strawman solution would result in very poor heap object collection rates and slow provenance tracking performance. However, recall that the goal of a sound provenance-based GC is not to ensure that no taint is lost, as a security analysis would, but rather to ensure that a pointer cannot be reconstructed from any provenance data. In order for an address range to be reconstructed, there must be enough data about the pointer propagated implicitly to definitely reconstruct it. We observe that for many comparisons, the binary outcome of that comparison doesn't propagate enough information to reconstruct the pointer, even if the comparison were made many times.

We consider two specific comparison cases for a valid in-bounds pointer `ptr1`:

I1 == or != `NULL`

I2 == or != another valid in-bounds pointer `ptr2`

The outcome of the comparison determines the value of `ptr1` from two possible partitions – $\mathcal{S}_1 = \{\texttt{NULL}\}$ *or* $\{\texttt{ptr2}\}$ and $\mathcal{S}_2 =$ the set of all other valid pointers. When in $\mathcal{S}_1 = \{\texttt{NULL}\}$, `ptr1` is an invalid pointer. When in $\mathcal{S}_1 = \{\texttt{ptr2}\}$, `ptr1`'s (or `ptr2`'s) value cannot be deduced from their equality alone but must be explicitly carried in `ptr1` (or `ptr2`). When in $\mathcal{S}_2$, we have eliminated only one possible value and still need sufficient information to determine `ptr1`'s value.

```
1  long implicit_copy(long ptr) {
2    long hidden_ptr = 0;
3    for (int i = 0; i < sizeof(ptr) * 8; i++ {
4      long mask = 1 << i;
5      if (ptr & mask) {
6        hidden_ptr |= mask;  // set bit in hidden_ptr
7      }
8    }
9    return(hidden_ptr);
10 }
```

Figure 4.3: Copying a pointer via implicit flow

Sufficient information to recover the pointer must then propagate either via at least one explicit data flow, or via a series of $2^{64} - 1$ equality comparisons, which are unreasonable to do in any practical amount of time. So, we can safely elide tracking implicit provenance propagation via these constrained comparisons. For all other comparisons, our sound GC propagates the provenance set through implicit operations.

The above condition for in-bounds pointers can be guaranteed statically for safe pointers but must be checked dynamically for unsafe pointers. Prior work on memory safety has enabled efficient spatial bounds checking [70, 7], and checks required for pointer comparisons only incur a fraction of those costs. Note, we cannot assume that the pointers used in comparison are guaranteed to be in-bounds. They may be out-of-bounds, and later become in-bounds before being dereferenced.

```
1  obj* A = malloc1(n*sizeof(obj));    // PT(A) = {malloc1}
2  char** B = malloc2(n*sizeof(obj));//PT(B) = {malloc2}
3  unsigned int o = 1000;
4  bool flag = false;
5  long x = A + o;      // PT(x) = {malloc1}
6  long y = B - o;      // PT(y) = {malloc2}
7  char* p = A;
8  if (B != NULL) flag = true; // elided by I1
   ...
9  if (A == p) flag = true;      // elided by I2
   ...
10 if (x == y) {
11   p = A + 2*o;       // PT(p) = PT(A) ∪ PT(x) ∪ PT(y) = {malloc1, malloc2}
12 }
```

Figure 4.4: Implicit pointer provenance propagation

Consider the example code in Fig. 4.4. After the comparison on line 8, the `flag` being **`true`** simply indicates that the pointer `B` is non-`NULL` which cannot be used to recover a valid pointer within object B. Similarly after line 9, if `flag` is **`true`** (or **`false`**), you still need either (or both) of `A` and `p` to access the object(s). However, line 10 propagates sufficient information to recover a pointer value. When `x == y` succeeds, it encodes the distance between objects A and B, so that even if all pointers to B are discarded, a pointer to B can still be recovered as in line 11. To handle this information flow, we add the pointer provenance of line 10's comparison operands, `x` and `y`, to

the control-dependent line 11's result `p`.

4.3.5 Other Points-To Set Propagation Channels

Pointer information can escape the managed address space and leak through external channels, such as by writing them to the file-system and reading them back. Safely handling such channels would require elaborate mechanisms to preserve the pointer provenance of such escaping values, and treat them as always live to exclude from being collected. Such pointer propagation channels being practically rare, we conservatively disable GC when any value escaping the program's address-space has non-empty pointer provenance.

4.3.6 C Standard for Pointers

The C standard [74] places certain restrictions on the possible values of pointers, limiting acceptable pointer behaviors in correct programs with well defined behaviors on all platforms. The standard disallows arbitrary manipulations on pointers [§6.5.6], but allows arbitrary, implementation defined, conversions between integer and pointer types [§6.3.2.3]. As a result, pointer-typed values may be in one of three states: (1) **`in-bounds`**: well defined in-bounds values, (2) **`one-past-end`**: pointing to a location just past the end of an array, and (3) **`imp-def`**: an implementation-defined value converted from an integer, which is unknown in the general case. We will now show that PROV-GC can leverage these restrictions to expand the set of safe pointers (§4.3.3) to significantly reduce pointer provenance tracking.

If we ignore the `imp-def` case for now, then these properties are clearly highly advantageous to our garbage collector. Because, if all pointer typed values are in-bounds or one past the end of an array, then all pointer typed values are safe-pointers by definition. Thus, our GC can apply our optimization discussed in §4.3.3 for all pointer type values.

The `imp-def` case does not present an instance of an in-bounds pointer, as it allows an arbitrary value to be present in a pointer. Fortunately, however, the lack of definition between conversion from an integral to pointer-typed value disallows the program from reasoning about any

value stored in that pointer, except under very specific conditions covered shortly. As any pointer in `imp-def` instance is not defined by the standard, a program cannot portably rely on it to reconstruct a pointer later, and thus it cannot be used to legally dereference or construct a pointer in the future, allowing PROV-GC to conservatively treat it as a safe-pointer.

The one exception we referred to is a defined conversion from a pointer value to an integral value and back as defined in [§7.20.1.4] of the C standard. This conversion applies to `intptr_t` values. For these values a `void` pointer may be converted to an `intptr_t` type and back. The conversion is not defined, except when the value stored in `intptr_t` variable is unchanged. Thus, any manipulated value of the integer would not have a standard-defined mapping when converted back to a pointer, and therefore our earlier conclusion for `imp-def` applies. If the value is unchanged when it is stored as `intptr_t`, then when it is converted back to a pointer type, it has to be either `in-bounds` or `one-past-end`.

Note that our example of XOR linked list in Fig. 4.1 complies with the above restrictions because it only uses `intptr_t` type to convert pointers into integers, and it recovers the exact value of the original pointer using XOR operation before converting it back to a valid pointer.

We note that this optimization relies on the programmer writing strictly standard compliant portable C code. Many implementations of C compilers do define mappings when performing pointer to integer conversions, and many code-bases do legally (but not portably) rely on these facets of the compiler [92, 75]. As a result, we provide solutions with and without this optimization. Programs that strictly adhere to the C standard can take advantage of this optimization.

4.3.7 Cautiously Optimistic Program Analysis

Provenance-based GC relies heavily on dynamic taint tracking [31], and consequently can incur significant overheads. Fortunately, recent work has shown that dynamic taint tracking can significantly benefit from a technique known as optimistic hybrid analysis [36, 1]. Cautiously Optimistic Program Analysis (COPA) is a method of dynamic analysis optimization based on the insight that optimization should be done for the common case. Traditional hybrid analyses use a sound static

analysis to reason about *all possible* future executions (and many impossible ones, due to over-approximation). However, when optimizing a dynamic analysis, the optimization need only care about the execution that will actually happen. To help approximate this, COPA uses a *predicated* static analysis, which takes in a set of assumptions (called likely invariants), and only guarantees that the static analysis is sound for executions in which these likely invariants actually remain invariant. Assuming these invariants allows the static analysis to reason much more effectively about the analyzed program, dramatically improving its ability to reduce dynamic checks. In this case, it allows the dynamic taint analysis to be aggressively optimized by eliding taint tracking monitors along paths that do not propagate taints in the predicated static analysis. A runtime system then checks the likely invariants at runtime, and falls back to a conservative analysis if the invariants ever fail.

We leverage COPA to improve our provenance-based GC in two ways: (1) we use it to improve our static empty points-to analysis in §4.3.3, (2) we assume pointers used in comparisons are in-bounds, reducing the amount of implicit flow tracking done in §4.3.4. For our first use of COPA, we simply apply the same optimizations found in the Iodine tool [36] to improve our common-case identification of empty may-points-to sets. Our second use is slightly more subtle. A conservative analysis would require that we propagate implicit flow information for any pointer which may be out-of-bounds during the **I1** and **I2** implicit flow checks. However, it is *very* rare for a pointer used within a comparison to be out-of-bounds, so we assume the invariant that any pointer used in a comparison is in-bounds. Using this invariant, we can remove any implicit flow taint tracking that may occur in the common case, so long as we first dynamically verify that all pointers used in comparisons for branches are in-bounds. If the check fails before the comparison, we soundly switch to the conservatively optimized analysis that propagates taint through the implicit flow.

4.4 PROV-GC Implementation

We present an overview of PROV-GC's notable features, and its details are presented in [93, §4].

PROV-GC relies on three primary components: (1) a Static Pointer Provenance Analysis (PPA), (2) a Dynamic Pointer Provenance Tracking (PPT) instrumented on the target program, and (3) a Provenance-based GC library for use by the target program. Our static analysis, and dynamic analysis instrumentation is implemented in the LLVM 7.0 compiler infrastructure [37], and the Provenance-based GC library is a modification of the Boehm-Demers-Weiser GC [73] to use provenance metadata for GC.

4.4.1 Static Pointer Provenance Analyses

The static analyses classifies all LLVM static single assignment form [94] values in a program into three partitions– non-pointers, safe-pointers, and unsafe pointers. It uses the same underlying whole-program context-sensitive data-flow analysis described in §3.4, along with a control-flow dependence analysis. After creating the predicated program DUG graph, we assign an empty *static points-to* set to every value and initialize the set for values defined by pointer sources (e.g. `malloc`). It then traverses the whole-program DUG iteratively, accumulating the union of points-to sets of values that are used in a definition, until all points-to sets reach a fixed point.

Non-Pointers: At the end of this data-flow analysis, all values with empty points-to sets are definitely non-pointers, and the rest are may-be pointers. No instrumentation is required for non-pointers.

Safe Pointers: We further classify may-be pointers into safe and unsafe sets as defined in §4.3.3 by constructing a conservative data-flow analysis identifying must-be safe-pointer operations as operations which may not pass through unsafe operations. Once we compute this analysis, we elide points-to set tracking for any safe pointers.

Unsafe Pointers: For the remaining may-be unsafe pointers, we elide equivalent points-to propagations when a manipulated pointer value is assigned back to same source-level identifier as its

source operand. This is done by mapping LLVM SSA values to their source-level identifiers and checking if values are assigned transitively to the same identifier along the define-use chain. Remaining unsafe pointer operations require both pointers-in and points-to tracking.

Implicit Flow Optimizations Finally, the static analysis optimizes implicit provenance tracking. When any branch conditional on an equality or inequality comparison has any pointers in the comparison, we instrument the spatial safety invariant check for the pointer operands and then optimistically elide the implicit PPT operations. Any other logical comparison used in conditional branches are always instrumented for all control-dependent value definitions.

4.4.2 Dynamic Pointer Provenance Tracking Instrumentation

After the static provenance analysis, we instrument the target program for dynamic Pointer Provenance Tracking (PPT) operations. This entails three types of pointer metadata operations, and COPA invariant checks.

Root-set Pointers: We instrument the main entry function to record the locations of all global values which may hold pointers. Every function, at entry, adds to this record the locations of all local pointer values and removes them before returning. This ensures that the GC can always locate statically identified safe pointer locations, and we only need to track the rest.

Pointer Metadata Creation: Any location that could not be statically identified as a non-pointer has two pieces of metadata associated with it- A pointer flag indicating that it has a safe pointer value maintained in a shadow memory, or its *dynamic points-to* set when it's unsafe. The points-to set is kept in a splay-tree [95] indexed by the pointer value location to serve range queries efficiently.

Pointer Metadata Tracking: The shadow memory taint creations, lookups and transfers are efficiently handled using LLVM DFSan [38] instrumentation. For explicit propagation of unsafe pointers, the points-to set metadata is computed as union of the sources. For implicit propagation, we use Iodine's recovery mechanism to create two paths, with and without the implicit tracking, based on the spatial safety invariant.

COPA Invariant Checks: In addition to all likely invariants used for static data-flow analysis, our implicit provenance optimizations add additional spatial bounds checks for pointer comparison operands.

4.4.3 Garbage Collection

PROV-GC keeps an *allocation table* of active allocations with their base and bound addresses, indexed by their base address. This is exposed to the provenance tracking mechanism for computing the dynamic points-to and enforcing the spatial safety invariant checks.

PROV-GC uses the dynamic points-to and taint metadata maintained by PPT to compute the pointers-in set within an object's bounds, using the splay tree to efficiently search all unsafe pointers, and leveraging the one-to-one mapping of heap locations to taint bits in shadow memory to efficiently lookup safe pointers using bitwise arithmetic.

Garbage collection begins by pushing the root-set of pointers maintained by our root-set tracking into a set known as the GC root-set. Then it queries the allocation table to locate remaining pointers within the bounds of the global data segment and the current stack to include in the GC root-set. Then marking continues by transitively performing the range-queries into the bounds of the objects in their points-to set. The range-based query techniques quickly locate all pointer values within an object's bounds, much faster than value-based scanning for large objects. Upon collecting objects, PROV-GC removes the associated metadata through range-deletion operations.

4.4.4 Source Transformations for GC

Running target programs with GC require some source-level changes to communicate between the collector and the client program. To convey applications' allocation requests, we replace all allocation calls e.g. `malloc()` with corresponding `GC_malloc()`'s, and remove all `free()`'s. Some applications like `redis`, need to notified by the GC for special handling of deallocated objects, for which we use BDW-GC interface to register the application specific finalization code.

4.5 Evaluation

We compare the performance of PROV-GCwith state-of-the-art value-based GC BDW-GC[73]. Our evaluation shows the following:

- Dynamic Pointer Provenance Tracking incurs reasonably low overheads, even for memory-intensive benchmarks.
- PROV-GC reduces scanning overheads so that individual GC invocations run faster compared to BDW-GC.
- PROV-GC is the only sound garbage collector, and yet improves memory reclamation rates over value-based GC and yields benefits similar to other unsound GC solutions [88].

Experimental Setup

We evaluate PROV-GC over several real-world applications including the `nginx` web server, `postfix` mail server, `redis` database server, `vim` and SPEC C benchmark programs. The profiling methodology and run configurations for the test programs are identical to that used in §3.5. For each target program, we create 3 versions: (1) **`base`** without GC, (2) **`bdw-gc`** with value-based BDW-GC and (3) **`prov-gc`** with our sound PROV-GC. The `base` versions use `glibc 2.26` allocator, except for `nginx` and `redis` which use `jemalloc 5.1.0`. All programs are compiled with `clang 7.0` at the `-O3` optimization level. We use BDW-GC version `7.4.16` with the parallel and incremental collection being disabled (`GC_MARKERS=1 GC_DISABLE_INCREMENTAL`). The benchmark programs and the configured GC heap limits are listed in Table 4.1. Finally, `prov-gc` compiles the programs with our static analysis that instruments them with the provenance tracking mechanism, and run with the PROV-GC allocator using the same configurations.

4.5.1 Provenance Tracking Overheads

To understand how static analysis can significantly reduce the overhead of provenance tracking, we run PROV-GC configured only to track provenance (i.e. collection disabled), and then selectively

Table 4.1: PROV-GC Benchmark configurations

Program	base	bdw-gc	
	Peak Memory	Heap Limit	# Collections
perlbench	580 MB	1024 MB	2
bzip2	856 MB	1024 MB	3
gcc†	940 MB	1024 MB	×
mcf	832 MB	1024 MB	3
gobmk	32 MB	32 MB	2
hmmer	60 MB	48 MB	1
sjeng	180 MB	256 MB	2
libquantum	108 MB	128 MB	2
h264ref	68 MB	48 MB	2
nginx*	26 MB	16 MB	2
redis*	316 MB	512 MB	3
postfix	588 MB	1024 MB	4
vim	244 MB	512 MB	3

†we're unable to run gcc with bdw-gc, *nginx and redis employ their own custom GC allocators

enable optimizations within PROV-GC. Our results can be found in Fig. 4.5: each benchmark shows 4 different overheads normalized to base– 'Cons' uses the sound static analyses described in §4.3.3 and §4.3.4 to optimize provenance tracking; 'Opt' further optimizes using optimistic hybrid analysis as described in §4.3.7; the two '+C' versions then use the specific optimization in §4.3.6 leveraging the C Standard. We find that the overhead of provenance tracking, including implicit flow tracking, for our benchmarks is actually quite reasonable, with an average overhead of 16% (11% excluding gcc). This result is actually quite surprising, as this number includes the cost of implicit flow tracking, which is known for dramatically increasing taint tracking overhead due to over-tainting. However, with our combination of static analyses, and optimistic hybrid analyses, we are able to dramatically reduce this result to only 16%. Note that this solution requires strict adherence to the C standard, which is stronger than spatial memory safety. While spatial memory safety only checks that pointers are in-bounds when dereferenced, the standard requires that pointers be in-bounds *always* for well-defined behavior. Therefore, the design point that does not assume the C Standard is also quite useful in supporting sound GC for legacy non-portable C programs that do not follow this strict standard. Programs that violate the standard [92, 75] can still employ sound GC, although incurring a higher overhead of $\sim 60\%$ (37% excluding gcc). Note that this cost is still comparable to that of Temporal Memory Safety checking solutions (60%

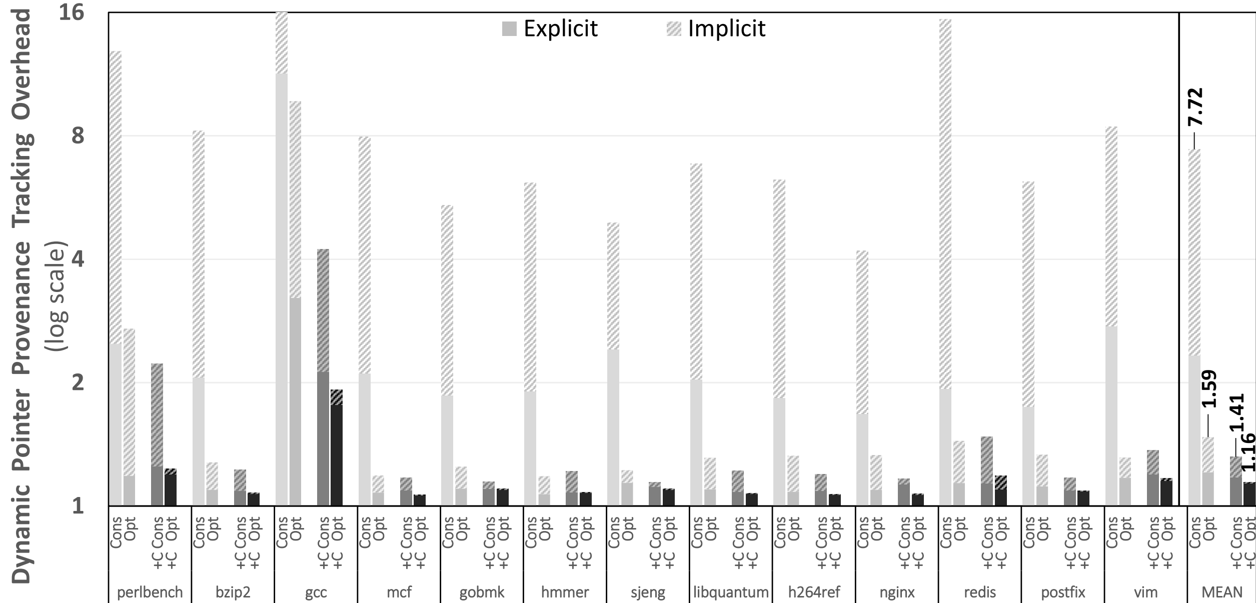


Figure 4.5: Result: PROV-GC dynamic pointer provenance tracking overheads

Cons: optimizes dynamic provenance tracking using sound static analyses, Opt: uses optimistic hybrid analysis.
+C: optimizes for C Standard-compliant programs as discussed in §4.3.6.
The solid portions of bars represent the overheads of tracking provenance via explicit flows, and the striped portions represent that for implicit flows. Execution times are normalized to `base`, i.e. without GC.

overhead [62]).

`gcc` is a very large program for which whole program context-sensitive pointer analysis does not scale, even when predicated using optimistic hybrid analysis. The context-insensitive pointer analysis for this program results in much less precise may-alias relations. Consequently, non-pointers can be imprecisely classified as may-be pointers, and safe pointers to be unsafe. This induces much weaker static optimizations resulting in severe dynamic overheads. The average overhead of provenance tracking excluding `gcc` is 11%.

We further study the various sources of our provenance tracking overheads in detail. For explicit provenance tracking, $\sim 21\%$ of its overhead attributes to the provenance metadata creation– i.e. when a safe pointer becomes unsafe, we compute the provenance metadata from the value of the source safe pointer. This substantiates that very few pointer values become tainted as unsafe and the remainder of the overhead is in tracking their provenance propagation. On the contrary, for implicit provenance tracking, the overhead is entirely in validating the invariant of spatial bounds checking for pointer comparisons, as we discussed in §4.3.4, and none of our tested programs

ever violate these checks. The framework overheads of checking the other optimistic invariant assumptions are negligibly low.

Summary : Optimistic hybrid analysis combined with our optimizations in §4.3.3 significantly elides tracking operations for most non-pointers and safe pointers, and tracks only few unsafe pointers. Pointer provenances do not propagate implicitly via common pointer comparisons as we reasoned in §4.3.4, and checking for that involves a subset of spatial memory safety check overheads. The cost of soundness for GC in our Pointer Provenance Tracking is $\sim 16\%$. This is significantly lower than the cost of providing Temporal Memory Safety ($\sim 60\%$ [62]).

Note that, to the best of our knowledge, these benchmarks do not exercise unsafe pointer propagations and BDW-GC also works correctly for them. For the programs that exercise unsafe pointer manipulations, BDW-GC might break, but PROV-GC still works correctly. However, programs with pathological pointer manipulations may incur higher overheads with PROV-GC. So, PROV-GC provides soundness in all cases without hurting performance significantly in the common case.

4.5.2 GC Overheads

Next, we show how the presence of dynamic pointer provenance information can be used to achieve an efficient GC solution. To study this effect, we compare the collection times of a single GC invocation with `prov-gc` against that of `bdw-gc`. Since our pointer provenance metadata is maintained separately outside the GC managed heap, the first GC invocation of a program happens at the same execution point under consistent configurations. But, since `bdw-gc` can reclaim unsoundly and retain imprecisely, the subsequent GC invocations can happen at different program states. So for equivalent comparison, we only measure collection statistics upon the first GC invocation and then terminate the program.

Fig. 4.6 plots the overhead of a GC invocation with `prov-gc` normalized to that with `bdw-gc`. `prov-gc` generally improves the GC collection times compared to `bdw-gc` and completes collecting $\sim 13\%$ faster. While `bdw-gc` performs expensive value-based scanning over large alloca-

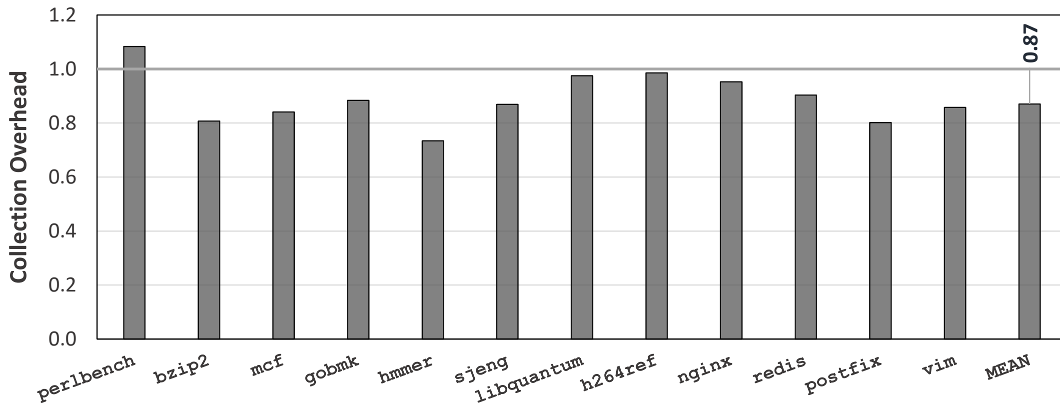


Figure 4.6: Result: PROV-GC reduces overhead of a single GC invocation; collection time is normalized to that of `bdw-gc`

tions to locate potential pointers for marking, `prov-gc` can locate values with pointer provenance using the fast range queries over its metadata. This benefit can compound while marking large allocations with sparsely located pointers.

We study the effectiveness of provenance-based GC in terms of its memory retention rate, i.e. the fraction of heap size after and before a GC invocation, once again in our previous single GC invocation setup. Fig. 4.7 plots the mean memory retention for each program. `prov-gc` is strictly more precise than `bdw-gc` and reclaims as much as 21% more memory in the case of `vim`, and $\sim 6\%$ more on average. This benefit varies with the programs' memory usage patterns, and is low for programs with a stable working set like `mcf`, `hmmer` and `sjeng`. Prior works on Precise GC [88] for C report better reclamation (up to $\sim 70\%$) benefits on a different set of applications, although they do not guarantee that the reclaimed objects will not be used in the future.

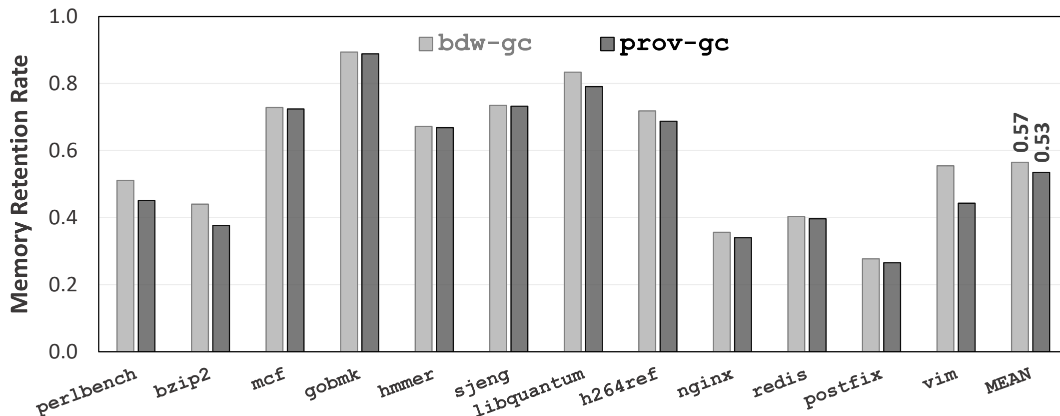


Figure 4.7: Result: PROV-GC reclaims more memory per GC invocation

4.5.3 GC Heap Size Sensitivity

Finally, we look at the end-to-end performance of `prov-gc` and its space-time trade-off with varying heap sizes. Fig. 4.8 shows the execution times of `prov-gc` and `bdw-gc` normalized to that of `base` for four benchmark programs with varying heap size limits on the x-axes; the labeled numbers on the plots indicate the number of GC invocations per benchmark configuration. Overall, `prov-gc` runs slower than `bdw-gc` and the difference between their execution time plots is attributed to the dynamic Pointer Provenance Tracking overheads. We observe that more frequent GC invocations at lower heap size limits lead to higher execution time overheads. This behavior is consistent for both `bdw-gc` and `prov-gc` although interestingly `prov-gc`'s improved reclamation rate results in fewer GC invocations for `vim` at lower heap limits.

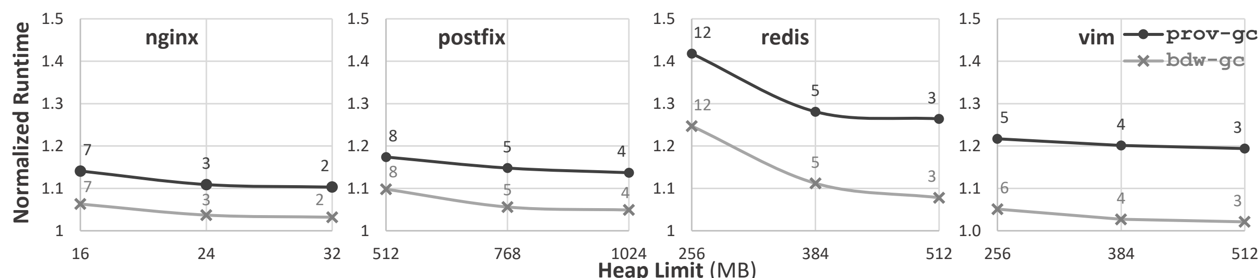


Figure 4.8: Result: PROV-GC performance with varying heap limits

4.5.4 Memory Overheads

To evaluate the additional memory overheads of maintaining the pointer metadata, we measure the sizes of the two metadata structures- shadow memory, and splay tree, at collection time. Programs running with PROV-GC have 30.8% more memory footprint on average, with `perlbench` seeing the highest overhead of 35.1%. Of this memory overhead, $\sim 27.1\%$ is due to the shadow memory structure, and $\sim 3.6\%$ is occupied by the splay tree structure.

Tracking pointer provenance metadata incurs some additional memory overheads, and hence may not be suitable for applications with large memory footprints in memory constrained environments. However, the advantages of sound garbage collection are much prominent in contrast.

Limitations

Possible leaks: All GCs are imprecise and can lead to memory leaks. PROV-GC too may suffer from memory leaks in pathological cases where it finds benign data-flows from pointer values, e.g. when certain bits of `malloc` return addresses are checked for bookkeeping purposes. In practice we do not observe such behavior.

Thread safety: Prov-GC's implementation currently supports single-threaded programs. Future work can address this limitation by combining prior work on data-flow analysis for concurrent programs [96] with our optimistic hybrid analysis techniques to construct a static pointer provenance analysis for concurrent programs. The C11 standard, by requiring data-race freedom, allows extending the sequential reasoning of many program analyses to concurrent programs [97]. Note that the provenance metadata accesses do not introduce any new races, and the per-word taint metadata is already covered by the program's existing synchronization for shared objects.

4.6 Related Work

In §4.2, we discussed the limitations of well-known value-based GC for C w.r.t soundness, precision, and GC performance. Below we relate relevant GC work that address each of these problems to our work. We also discuss work that relates to the techniques (provenance) we use and the added benefits of GC (temporal safety).

GC Soundness: Prior work has developed compiler checks to reject C/C++ programs that may violate soundness of value-based GC. Precise GC solutions check that programs do not store pointer values into integral types [87, 88] during compilation. Conservative GC [73] checks its assumptions (e.g., that integers are not converted to heap pointers), and preserve original pointer values around compiler optimizations of pointer arithmetic [98, 99]. However, these solutions can reject legal C/C++ programs, as they essentially make pointer manipulations and casting illegal.

GC Efficiency: Traditional mark-sweep collection [73] has been optimized using *parallel* marking algorithms [100, 101], by collecting *incrementally* [102, 103], by treating *generations* of objects separately [104, 105], or by organizing the heap into regions and performing mark-region GC [106, 101]. The fragmentation problem when dealing with ambiguous pointers in uncooperative environments like C has been addressed by *mostly-copying* collectors that move heap objects with no direct references from the root set [107, 108, 105, 109, 110]. These optimizations are orthogonal to our goal to realize sound GC for C, but they can be integrated into our GC.

GC Precision: Precise GC techniques disambiguate pointers from non-pointers to some degree, relying on programmer annotations to register live pointers in a shadow stack to be managed by GC libraries [111, 112, 113], or with cooperation from the compilers [114]. When compiling high-level languages to C [115, 116, 117], a virtual machine with its own stack and registers convey necessary type information to the GC, but this complicates code generation and makes systems fragile and non-portable. As inefficiencies of conservative collection arise mostly due to their conservative treatment of the root pointers [118], type-accurate GC [87] accurately locate pointers in a shadow stack through extensive source transformations. Later systems [119, 120, 88] improve upon this by optimizing metadata storage, and using static liveness analysis assuming the programs obey several constraints. Such techniques are primarily motivated to solve the leakage and fragmentation issues by enabling copying collection, although they incur significant additional framework overheads [87, 88] and their reduced retention and compaction benefits are marginal [121]. But importantly, these systems are more unsound than conservative GC in assuming that pointer values are only stored into pointer types.

Reference Counting: In contrast to above reachability-based tracing collectors, Reference Counting keeps count of incoming references to each object [122] and reclaims an object when its reference count falls to zero. This is natively supported in many languages like- Objective-C, Perl, PHP, and Swift, and also in C++ via 'smart pointers'. This requires the compiler to identify all pointer updates and account reference counts which can become expensive; so it is deferred and performed periodically and incrementally [102]. Reference counting is inefficient compared to pre-

cise GC [84, 85], and moreover cannot collect cyclic garbage requiring separate cycle collectors [123, 124] or forbidding cycles altogether [125]. Importantly, they inherently rely on type safety and cannot handle pointer information flowing into non-pointers. Our dynamic pointer provenance could be used to maintain reference counts to objects, and thereby realize a sound reference-count based GC for C.

Taint and Provenance Analysis: There is a significant body of work on static and dynamic information-flow analysis by tracking taints. They have been largely used to ensure that private data do not leak through untrusted channels [16]. Static analysis [54] has also been used to reduce the overhead of dynamic-taint tracking. Our COPA-based static analysis elides runtime monitors by leveraging optimistic assumptions, and by taking advantage of special properties of the C language standards that are beneficial in the context of dynamic pointer provenance. Also, unlike classical taint solutions, we show how implicit-flow tracking is necessary and feasible to track in our context.

Recent work has introduced static pointer provenance for C [71] in order to improve static alias analysis. This static analysis was used by C compilers to improve compiler optimizations. In contrast, we discuss dynamic pointer provenance, and optimistic hybrid analysis to optimize it. Recent work used dynamic pointer provenance for implementing capability checks [126] to improve security. In contrast, we use dynamic pointer provenance to construct sound GC for weakly typed languages, and optimize that using optimistic hybrid analysis.

Memory Safety: *Temporal memory safety* ensures that programs access only allocated memory, and *Spatial memory safety* ensure that all accesses are within allocated object bounds. Spatial MS is ensured by dynamically checking that intermediate pointer arithmetic do not cross valid object boundaries [127], and further tracking their intended objects for out-of-bounds pointers [128]. This approach has been improved by allocating memory in pools and storing object bounds more efficiently [65], by restricting memory allocation sizes and layout to efficiently compute bounds checks [129], eliminating redundant checks through static analysis [130, 7], and hybrid solutions combining static analysis with hardware support [70]. Temporal MS requires tracking liveness of

objects and checking for erroneous uses of uninitialized objects and dangling pointers (use-after-free, double-free), which has also been heavily optimized using static analysis [6, 7, 62].

Another approach to MS, adopted by Cyclone [131], CCured [3], SafeCode [65], Checked C [132], and Managed C++ [133], is to enforce constraints through a strong type system and then perform sound analysis to check for memory errors, but the language becomes much restricted than C, making porting programs hard. A contrasting approach is to combine heuristics, programmer annotations, and unsound analyses in designing tools [134, 135, 136, 137, 138, 139] that detect most memory usage errors in practice.

Our sound GC guarantees temporal MS for legacy C/C++ code by automatically collecting safe garbage that cannot be accessed later, and we show this can be done more efficiently.

Dynamic Type Inference: Types can be inferred from binaries during execution [140, 141], optionally aided with static analysis [142], for many applications including- decompilation, binary rewriting, vulnerability detection, memory forensics, etc. The typed binary can be executed with dynamic type-safety checks [143]. Our dynamic tracking infers more than pointer types, as it also tracks the pointer provenance when type-safety is violated.

Summary

Traditionally, Garbage Collectors have relied only on values to identify pointers in uncooperative environments like C. C being weakly typed, this is unsound for several legal pointer manipulations. We show that tracking pointer provenances using Dynamic Information Flow Tracking can soundly identify all pointer information, even those hidden by their values, and a Provenance-based GC will therefore safely collect only objects which *cannot* be accessed later.

To realize a practical *Pointer Provenance Tracking* solution, we leverage Optimistic Hybrid Analysis, and identify properties that allow us to elide tracking for most common pointer operations. Our tool PROV-GC tracks pointer information propagations with only $\sim 16\%$ overhead, even via implicit control-flows, and also improves the overhead and effectiveness of collection.

CHAPTER 5

OPT-SC: Efficient Sequential Consistency for `Java`

Modern concurrent languages such as C++/Java guarantee sequential consistency (SC) for data-race-free programs. Data-races, however, are a common form of programmer error. For programs with such data-races, their memory model guarantees are either undefined (C++) or too esoteric for most programmers (Java).

A practically viable solution for guaranteeing language-level SC for all programs (SC-for-all), even those with data-races, remains elusive. Current solutions are either too expensive, requires custom SC hardware or imposes significant language-level restrictions.

We address this need for a low-cost SC-for-all solution by using a precise static data-race detector, so that only a small set of potentially racy instructions need to be guarded by the costly fence operations. Conventional sound static data-race detectors, however, are too imprecise and/or do not scale to large programs. We address this problem using a new Cautiously Optimistic Program Analysis (COPA) that induces a significantly more precise and scalable whole-program static analysis by assuming likely program invariants. By checking likely invariants at runtime, and recovering when any of them fails, SC is guaranteed for all executions. We realize language-level SC for Java on commodity x86 hardware at only $\sim 5\%$ overhead for Spark.

5.1 Enforcing Strong Concurrency Semantics is Challenging

A *memory consistency model* defines the semantics of a concurrent programming language by specifying the order in which one thread's accesses to shared memory objects become visible to other threads in the program. *Sequential consistency* (SC) [144] provides an intuitive memory model by ensuring that the program's instructions (appear to) execute in a global total order con-

sistent with the per-thread program order. While programmers would benefit from such a simple memory model with strong guarantees about the behavior of their programs, it can significantly reduce the scope of compilers and the underlying hardware from performing optimizations. The popular data-race-free-0 (DRF0) model [89] attempts to strike a balance between simplicity for programmers and flexibility for compilers and hardware performance. In the DRF0 model, programmers explicitly annotate synchronized accesses (e.g. using `volatile` in Java and `atomic` in C++) and the compiler and hardware limits in their optimizations to respect the semantics of synchronized accesses declared by the programmer. The DRF0 model guarantees SC for correctly synchronized programs that are free of data races, but leaves the semantics undefined for programs with data-races. The current memory model for C++ [145] is based on the DRF0 model and its undefined semantics in the presence of data races complicates debugging tasks and reasoning for program safety. The Java memory model [146] provides semantics for racy programs, but is weaker than SC. However, this weaker semantics is too complex, and reasoning the correctness of programs and various compiler and hardware optimizations remains challenging [147, 148].

The goal of our work is to make SC-for-all practical, i.e. all programs with or without data races are guaranteed sequential consistency. Providing language-level SC semantics, however, remains prohibitively expensive. In order to execute a given program under strict SC semantics, all transformations or optimizations made by the compiler and the underlying hardware must preserve the natural SC orderings of the source program. Existing SC-preserving compilers [149] ensure the SC behavior of the source program is preserved by restricting certain compiler transformations while retaining much of the performance gains of the optimizing compiler. However, modern commodity hardware (x86, ARM, PowerPC) that aggressively optimize memory accesses do not guarantee SC behavior at runtime. So, systems requiring SC guarantees must additionally emit expensive hardware fence operations around *all* shared-memory accesses to restrict hardware optimizations and re-orderings that can potentially violate SC. This naive approach is very expensive and shared-memory concurrent programs thus perform poorly under the SC model on commodity hardware.

Providing SC semantics, however, does not require all shared-memory accesses to be guarded via expensive fences. If shared memory accesses can be *proven to be data-race-free* statically, then fences around such operations can be safely omitted. Since data-races are rare in production software, a precise static data-race detector would be able to eliminate almost all the fences, thereby resulting in near-zero overhead for SC.

However, a precise and scalable static data-race detection analysis remains elusive. Statically proving memory accesses to be data-race-free is hard. Static race detection tools used in practice often resort to unsound heuristics [150], do not reason about pointers altogether [151], or use an imprecise context- and field-insensitive pointer analysis [152]. Recently proposed *volatile-by-default* (VBD) semantics for Java [153] can provide SC, while allowing limited optimizations using function-local static analyses and speculative compilation[154]. But, the cost of providing SC remains quite high (24% on average and 64% maximum overhead for Spark).

In this work, we leverage Cautiously Optimistic Program Analysis (COPA) [1] to construct a **precise whole program static data-race-detector**, which in turn allows us to realize, a practically viable, low-overhead SC-for-all solution on commodity hardware.

COPA is a powerful technique that learns likely program invariants from profiled dynamic behaviors, and then predicates the static analysis state space assuming such learned invariants to induce a significantly more precise analysis. A COPA-induced static data-race detector is far better at identifying non-racing shared-memory accesses than prior techniques. Our SC-compiler optimistically compiles with fences only around memory accesses identified by this analysis as potentially racy. The programs execute with additional lightweight runtime checks that validate the assumed invariants. In the rare event that an invariant fails during a dynamic execution, we leverage the JVM's just-in-time compilation features to soundly recover the execution to a program version that is conservatively optimized without the likely invariants.

Our work presents the first low-overhead SC-for-all solution for Java on commodity hardware, that works for legacy programs without incurring language restrictions or requiring programmer annotations, using a significantly precise static data-race detection analysis. Our optimistic static

data-race analysis eliminates about 85% of false data-races compared to conventional static data-race analysis. Our SC-compiler can then elide fences for these accesses, irrespective of underlying hardware's memory model (x86 or ARM). This allows us to realize SC-for-all for Java on x86 for just 5% overhead on average and 13% maximum overhead for Spark.

5.2 Memory Consistency Models

Our goal is to provide SC for all programs at the language-level on commodity hardware. This section presents the necessary background on the challenges with current memory models and in providing language-level SC for all programs.

5.2.1 Data-Race-Free Memory Model and Its Limitations

A *memory consistency model*, or simply *memory model*, defines the semantics of a concurrent programming language by specifying the order in which one thread's accesses to shared memory objects become visible to other threads in the program. A data-race-free (DRF) memory model requires programmers to explicitly annotate synchronization variables that can race (using `volatile` in Java, `atomic` in C++). There exists a data-race between any two data accesses to a memory location, if they are from different threads, at least one of them is a write, and if they are unordered by synchronization accesses.

For data-race-free programs, DRF languages provide sequential consistency (SC). *Sequential consistency* (SC) [144] guarantees that a program's instructions (appear to) execute in a global total order consistent with the per-thread program order. This intuitive memory model is easily understood and has well known benefits [89].

DRF, however, treats programs with data races in one of two ways– (*Strict*) treats data races as errors leading to undefined behavior (C++[145, 155]), or (*Weak*) defines feasible program execution semantics, but they are too complex for programmers to reason about (Java[146]).

While it is naturally desirable to write data-race-free programs, many programs unfortunately

contain data-races, sometimes rather intentionally [156] and they are present across a wide range of Java applications [153]. It is challenging for programmers to reason about these program behaviors in the presence of data-races and debug them. Java's semantics for data races are so complex that even building correct compilers can be challenging [147, 148, 157].

5.2.2 Sequential Consistency For All

SC is easily understood and has well known benefits [89]. It is therefore desirable to guarantee SC semantics at the language-level for all programs, even those with data-races.

However, in spite of its well understood benefits, SC is not supported in practice, due to its prohibitively high performance overhead. To preserve SC at the language-level, we must ensure that the observable ordering of memory accesses is preserved both at the compiler level and at the hardware level. Prior work has constructed a SC-preserving compiler [149] that ensures the SC behavior of the source program while retaining much of the performance gains of the optimizing compiler (less than 2% overhead).

However, efficiently guaranteeing SC at the hardware level remains challenging since commodity hardware (x86, ARM) perform several memory reordering optimizations that violate SC. To guarantee SC behavior on such modern commodity hardware, the compilers, in addition to being SC-preserving, need to insert expensive memory ordering operations, most commonly in the form of fence operations, to restrict the hardware from reordering shared-memory accesses around them. This severely restricts hardware optimizations, leading to high performance overhead.

Not all shared-memory accesses, however, need to be guarded by expensive fence operations. If a memory access can be statically proven to be data-race-free by the compiler, the compiler can safely elide the unnecessary fence for that access. Fewer the fences, lower the SC overhead.

Proving data-race-freedom statically is, however, challenging. The only prior work that comes close to eliminating a vast majority of fences using advanced concurrency analysis require significant language restrictions on a dialect of Java [158]. These techniques do not translate to current C++ and Java language standards, and therefore practical SC for all programs remains a challenge.

Volatile-by-default (VBD) semantics

VBD is a recent solution [153] that treats all shared variables as `volatile` unless explicitly marked as otherwise (data-race-free) by the programmer. VBD guarantees SC for programs with correct annotations of data-race-free accesses. Memory accesses that are incorrectly annotated would have weaker guarantees as data-races in Java.

VBD also conservatively provides SC for programs with no annotations. It uses function-local static analysis to conservatively prove a subset of thread-local accesses to be data-race-free. Recent work [154] improved this solution by speculatively assuming all accesses are thread-local, and then dynamically recompiling the program as shared memory accesses are discovered. The resulting system is currently the state-of-the-art SC-for-all solution (assuming no annotation). Its overhead for x86 hardware is still too prohibitive for practical adoption: $\sim 24\%$ on average, and maximum 65% for Spark.

5.2.3 SC-for-all Using Precise Datarace Detection

In most correct programs, a large fraction of accesses to shared-memory locations are in fact data-race-free. A precise data-race detection analysis can identify memory accesses that can potentially be racy and the SC-compiler can then selectively apply fences only around such operations.

A scalable and precise sound static data-race detection analysis, which in turn relies on sound memory alias analysis, is challenging as it must reason about all possible executions. Static data-race detection tools used in practice therefore often resort to heuristics [150], do not reason about pointers altogether [151], or use a weaker context- and field-insensitive pointer analysis [152]. Next, we describe how our Optimistic Hybrid Analysis approach solves these fundamental limitations of traditional sound static data-race analyses to realize SC at a low-overhead.

5.3 Precise Predicated Static Datarace Detection

Conventional sound static data-race detectors are imprecise and/or not scalable to large programs. They report a large number of potential data-races, which we refer to as **may-race** accesses, most of which are actually data-race-free in practice. Since all may-race accesses need to be guarded with fence, lower the precision, higher the overhead for SC.

We use *predicated static data-race analysis* [1], which allows us to classify a large fraction of may-race accesses into **likely-race-free** memory accesses. This is achieved by restricting the program states analyzed by an otherwise sound static analysis to a set wherein *likely invariants* hold. These *likely invariants* are properties that are likely to remain true for almost all executions, but are hard to prove. They can be learnt through profiling representative program executions. The predicated static analysis guarantees that the likely-race-free accesses are race-free for program executions, as long as the likely invariants hold true.

In the next section §5.4, we discuss our COPA-SC compiler that optimistically elides fences for these *likely-race-free* accesses in addition to the provably **race-free** accesses determined by a conservative static analysis. A program execution, however, can violate a likely invariant. Therefore, a key challenge is in safely recovering and guaranteeing SC, even in the presence of likely invariant violations. Our OPT-SC solution, discussed later in §5.4, addresses this key problem.

In this section, we first present an overview of traditional static data-race analyses and motivate the challenges. Then we present the design of our significantly more precise *predicated static data-race analysis*.

5.3.1 Conventional Conservative Data-race Analysis

A static data-race detector computes the pairs of program instructions that may lead to data-races in an execution. Typically, a static race detector works in two phases:

May-Happen-in-Parallel (MHP) analysis [159] statically determines whether two instructions (or two instances of the same instruction) in a program may execute in parallel or not. The

MHP analysis uses a whole-program context-insensitive flow-sensitive analysis to compute the set of all load and store instructions that can dynamically happen in parallel [160, 161]. A whole-program context-insensitive pointer-alias analysis is then performed to identify pointers in the program that may be used to access the same memory locations. The MHP analyses then combines this information to further identify those instruction pairs that may concurrently read and write to the same memory location. These determine the set of all potentially racy memory accesses.

Lockset-based pruning then excludes potentially racy instruction pairs from the above set that are correctly synchronized. It uses the pointer analysis information to compute the lockset for each memory access, i.e. the set of lock variables that alias with the lock that guards the memory access. It then excludes those pairs of racy memory accesses that are guarded by the same locks.

Challenges

Statically proving memory accesses to be data-race-free remains intractable in practice. The MHP analysis and underlying pointer analysis use whole-program data-flow analyses that are fundamentally imprecise and unscalable for reasonably large concurrent programs, because they must be overly conservative to be sound in the presence of dynamic features such as dynamic class loading, class redefinitions, etc. Moreover during the lockset-based pruning phase, the weaker *may-alias* relations cannot be effectively used to prune racy instructions guarded by a common lock which requires a stronger *must-alias* reasoning [160]. So, they are unable to prove many memory accesses as data-race-free. Using call- and object-sensitive analysis instead can improve precision [161], but such analyses do not scale for large programs as distinguishing the contexts leads to the state-space explosion problem. Traditional static data-race analyses either do not scale to large programs, or are too imprecise. Consequently as seen in Fig. 5.1, traditional static race detectors can identify only a small set of provably *race-free* accesses and conservatively treat that majority of accesses *may-race*, leaving behind fences for a large number of memory accesses that never actually encounter a race during any execution.

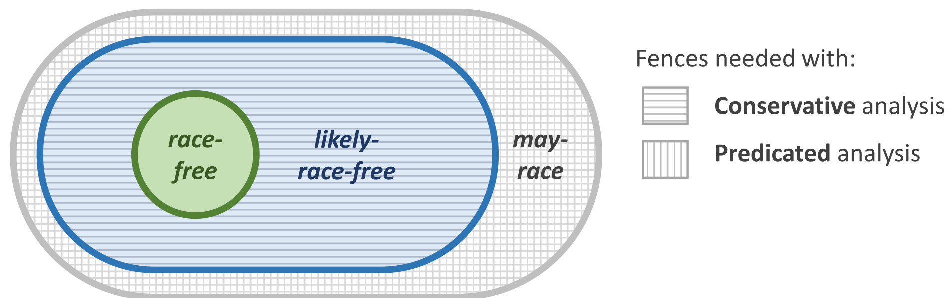


Figure 5.1: Benefits of predicated over conservative data-race analysis: While conservative analysis can identify only few *race-free* accesses thus emitting fences for the remaining majority of *may-race* accesses, predicated analysis can additionally identify a large number of *likely-race-free* accesses and then speculatively elide fences around them.

5.3.2 Precise Predicated Data-race Analysis

In order to tackle the fundamental problem of imprecision of static analyses, predicated static analysis [1] uses likely program invariants to predicate the static analysis to induce a significantly more precise reasoning as follows.

Likely Guarding Locks invariants dynamically identify the set of objects locked at each lock site and infers a *must-hold-same-lock* relation for pairs of lock sites that *always* only lock the same object. These relations inferred from the invariants enable the lockset-based pruning phase of the datarace detection to exclude many pairs of correctly synchronized memory accesses that are otherwise treated as potentially racy. In a conservative analysis without this dynamic information, the lockset-based pruning phase cannot remove many racy pairs since it cannot derive the required *must-hold-same-lock* relations from the weaker *may*-alias relations of a conventional pointer analysis.

Likely Singleton Thread invariants identify thread creation sites that only ever create a single instance of a thread. These *singleton-thread* instances greatly benefit the MHP analysis, since by assuming this behavior, the MHP analysis can reason that all memory accesses within the singleton thread are ordered thus allowing it to prune memory access pairs for singleton threads.

```java
1  class Data { int data = 0; }
2  class Clazz extends Thread {
3    static Data global = new Data();
4    public void run() {
5      Data local = new Data();
6      try {
7          helper(local);
8      } catch (Exception e) {
9          // executes rarely
10         helper(global);
11     }
12   }
13   static void helper(Data arg) {
14     arg.data++;                    // add fence
15   }
16 }
```

(a) **Conservative** Static Analysis

```java
1  class Data { int data = 0; }
2  class Clazz extends Thread {
3    static Data global = new Data();
4    public void run() {
5      Data local = new Data();
6      try {
7          helper(local);
8      } catch (Exception e) {
9          copa_check_recover();
10         helper(global);            // likely unreachable
11     }
12   }
13   static void helper(Data arg) {
14     arg.data++;                    // elide fence
15   }
16 }
```

(b) **Predicated** Static Analysis

Figure 5.2: Example Java program benefiting from predicated data-race analysis: (5.2a) Conservative analysis must add fence on line 14 for the rare case when `global` is passed to the `helper()` function on line 10. (5.2b) Optimistic analysis elides the fence on line 14 assuming the exception handler on line 10 is never called, and adds `copa_check_recover()` just before line 10 to detect and recover when this assumption is violated.

Example of COPA enhanced static data-race analysis

Consider the example program in Fig. 5.2a: multiple running threads operate on their thread-local data by passing `local` to the `helper()` function on line 7 during the normal execution, and only

operate on the shared data `global` on line 10 during an exception. The conservative sound analysis reasons that since on line 14, `arg` may indeed update the shared `global` data, it may race in some execution, and must be guarded by a fence. As a result, line 14 is always dynamically protected by the expensive fence even though it operates on the `local` data during normal execution, unnecessarily slowing down the execution. On the other hand, COPA in Fig. 5.2b is able to infer that line 10 is likely unreachable code just by observing few dynamic executions. Assuming this invariant, the predicated analysis reasons that since at line 14, only the `local` data reaches the update, it cannot race and need not be guarded by the fence. This reasoning holds for all correct executions of the program. However, during an exceptional execution on line 9, COPA is able to first detect and recover the execution before `helper()` is actually called with the `global` data.

5.4 COPA for Efficient Sequential Consistency

OPT-SC uses Cautiously Optimistic Program Analysis (COPA) to guarantee SC-for-all. As we discussed so far, our offline predicated static data-race detector can identify many more *likely-race-free* memory accesses in addition to those proven *race-free* using conservative analysis. Next, our OPT-SC compiler optimistically elides fences for these *likely-race-free* accesses and only emits fences for the remaining few *may-race* accesses. During most program executions when the assumed likely invariants hold, then the likely-race-free accesses are guaranteed to be data-race-free for that execution, and so it entails an SC execution. If these invariants fail, OPT-SC will safely add additional fences to guarantee SC. This speculative optimization allows OPT-SC to provide language-level sequential consistency for Java at only $\sim 5\%$ overhead on commodity x86 hardware.

Fig. 5.3 illustrates the complete workflow of our system. In §5.3.2, we already covered how we identified strong invariants through profiling to induce a more precise predicated data-race analysis. In the remainder of the section, we elaborate on how OPT-SC uses the precise predicated data-race detector to construct a compiler and runtime system that enforces SC behavior at runtime at a low

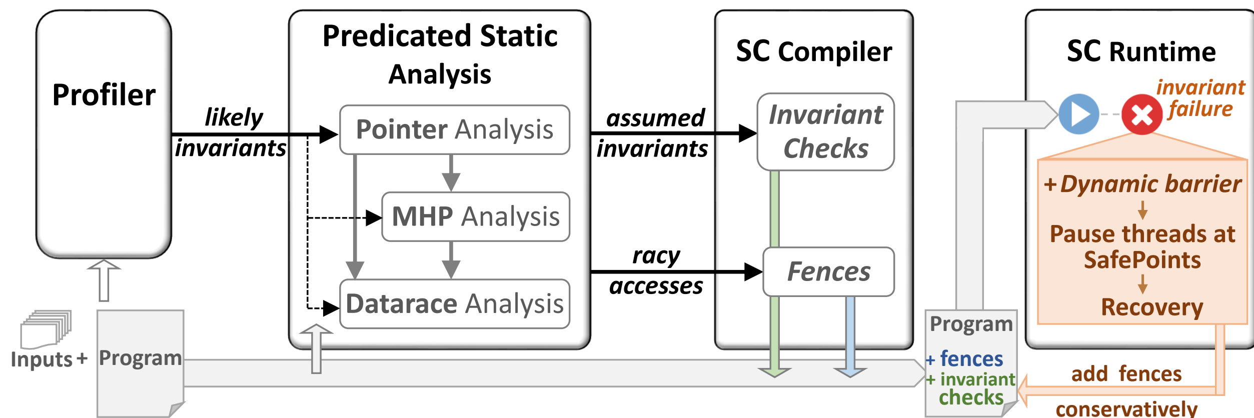


Figure 5.3: Workflow of OPT-SC

cost. The OPT-SC compiler enforces SC behavior by adding required fences, and the runtime system detects and recovers the execution to soundly provide SC in the rare event of a likely invariant violation.

5.4.1 OPT-SC Soundness For SC Guarantees

The key to the soundness of our system is guaranteeing that the program is data-race free. That is, we require an ordering *fence* operation between two accesses to the same memory location by different threads if at least one access is a write. If our system can guarantee a fence is placed between all such memory accesses, it guarantees memory access ordering, and the DRF0 system that it is built on top of will provide sequential consistency.

Most systems which use this model to provide SC-for-all simply elide fences around operations they can prove are data-race free. This trivially results in a well ordered operation, as any non-provably racy operation has a fence. However, OPT-SC removes fences speculatively, depending on the observed execution states. Consequently, for some executions one set of fences is sufficient, and for others another set may be required. We now discuss how OPT-SC guarantees orderings in all executions.

Trivially, predicated analysis promises that race-free memory accesses and likely-race-free accesses cannot cause races in executions for which likely invariants hold. OPT-SC can safely elide their fences for these execution.

When a likely invariant fails, OPT-SC recovers by transitioning the execution to a conservatively optimized program, by re-inserting fences at any likely-race-free memory access that were previously elided. Once these fences are added, all memory operations will be totally ordered under all conditions, guaranteeing safety. However, this transition must be handled with care to ensure data race freedom.

In summary, OPT-SC ensures correct SC behavior under all executions as follows:

- **Predicated Analysis** identifies likely invariants to prove many more memory accesses to be likely-race-free, and then speculatively elides fences around such memory accesses. The resulting execution is guaranteed to be SC when the assumed invariants hold.
- If an invariant ever fails, all accesses before the invariant failure are strictly ordered with all accesses after the invariant failure as follows:
  - **Eager Invariant Checks** detect an execution outside of those analyzed in the predicated analysis just *before* an invariant is actually violated.
  - **Ordered Transition** ensures that memory accesses from before the transition are strictly ordered before those from the conservatively recovered execution. This is achieved by inserting a *dynamic barrier* at the invariant check failure path and waiting for all threads to pause at *SafePoints*.
  - **Recovery** then switches the program to a version that conservatively re-inserts the elided fences, thereafter ensuring correct SC behavior.

5.4.2 OPT-SC Compiler

OPT-SC's design is based on the guarantees already provided by a language with the popular DRF0 memory model [89]. Additionally, we rely on some modern language features– particularly, we use dynamic de-optimization provided by Just-in-time compiled languages for constructing the COPA recovery, and use the notion of SafePoints in managed garbage-collected languages to reason for its correctness. We present OPT-SC's design for the Java language, although our system can be

extended for other managed languages like C#, Ruby, etc., however, our system's overheads and overall benefits may vary.

Once our predicated static data-race detection analysis has determined the set of memory accesses that can potentially race, our modified SC-compiler must satisfy two requirements–

First, it must insert fence operations only around the remaining *may-race* accesses, and speculatively elide fences for the *likely-race-free* accesses. To this end, we adapt the existing Java JIT compiler framework to identify the potentially racy instructions and selectively only emit the platform-specific hardware fence operations, as we describe later in §5.5.

Secondly, the compiler must respect the additionally imposed ordering constraints throughout all optimization passes. During the JVM compilation phase, Java bytecode is translated to a graph-based intermediate representation called the Ideal Graph. We modify the Ideal graph construction phase to create special memory-barrier nodes around the statically determined set of may-race memory accesses. All downstream compiler optimizations respect the semantics of the memory-barrier nodes and do not reorder memory accesses around them, thus ensuring SC at the compiler level.

5.4.3 Runtime Invariant Checks

Although the static datarace detection analysis elides fences assuming likely program invariants, this reasoning holds for the vast majority of dynamic executions that do not violate the assumed invariants. The invariants may, however, be invalidated in a rare execution. A key advantage of COPA is that even though an invariant violation renders the predicated static analysis to be unsound, the soundness of the final dynamic execution can still be recovered as long as the invariant violation is detected immediately and the dynamic execution is then recovered. So, the OPT-SC compiler additionally inserts checks that validate the likely invariants at runtime to ensure that the dynamic execution is within the set of executions that were statically analyzed in the predicated static analysis.

Our COPA invariants satisfy the property in **O2** as they are simple enough to be checked dynamically at a very low cost: violations of likely unreachable code blocks are discovered as soon as the code block is visited; validating likely guarding locks require the aliasing lock-sites to perform a quick check that they lock the same dynamic object; likely singleton thread creation sites require an inexpensive check on the number of spawned threads.

The other key property of our invariants as stated in **O4** is that– *the invariant checks detect before the invariants are violated*. This is critical to guarantee the soundness of our system upon recovery. All of our invariant checks satisfy this property by construction. A violation of likely unreachable code invariant is detected immediately upon entering the code block and before the code block is actually executed. A dynamic check for aliasing locks detects violations of likely guarding locks invariant before the locks are actually acquired. A simple check on the dynamic number of spawning threads at thread creation sites detects a violation of likely singleton thread invariant before the threads are actually invoked. Consequently, when a likely invariant violation is detected during execution, the dynamic program state is one where the violation is just about to occur. So, we detect invariant violations immediately *before* they take affect.

5.4.4 Recovering SC Upon Likely Invariant Violation

When an invariant fails during a rare execution scenario, it implies that the dynamic execution is about to exit from the set of states for which the predicated static analysis holds sound. Note that the execution state so far is one that satisfies the COPA predicates, and so adheres to SC even using the optimistic reasoning. However, future execution may violate SC as the predicated reasoning is no longer guaranteed to be correct. To ensure sound SC behavior, it is therefore necessary that the execution be recovered to a version that ensures SC by conservatively guarding memory accesses with fences without assuming the COPA invariants. Furthermore, it is essential that the SC orderings are adhered during the recovery process itself when the program threads are switching to the conservative version. We elaborate below how the recovery process satisfies this safety requirement in **O5**.

We introduce a *dynamic barrier* during the program recovery to ensure SC behavior while the program threads are undergoing the recovery to their conservative versions. Unlike a regular barrier which always enforces program threads to be synchronized and prevents operations from being reordered across, the dynamic barrier provides this synchronization semantic only on-demand when invoked by the OPT-SC recovery. This dynamic barrier effectively prevents memory accesses from being reordered by the compiler in the instance of an invariant violation. First, the offending thread that detects the invariant violation invokes a special recovery function that waits until all other threads are paused at *SafePoints*. Managed languages provide the notion of a VM *SafePoint* [162] where the state of the VM is well-defined so that threads can be safely paused and resumed. We extend the SafePoints construct to additionally require that invariant check violations are also SafePoints. This ensures the offending thread can be stopped safely and immediately. Additionally, all other running threads are paused at their nearest SafePoints; loop back-edges also being SafePoints [163], all threads pause quickly.

The recovery process is safe and preserves SC due to two reasons – Firstly, compilers prevent several unsafe optimizations including memory reordering across such SafePoints in order to preserve program semantics in the presence of dynamic language features such as code de-optimization, instrumentation, etc. And secondly, since the offending thread is paused and the invariant has not yet been violated, the existing fences instrumented using the COPA-induced predicated reasoning suffice to enforce the required SC orderings during the recovery itself. Moreover, the recovery process is *deadlock-free*, since any wait operation is also a SafePoint.

Once all threads are safely paused, the program needs to re-insert all optimistically elided fences. This can be achieved using a static compilation technique that maintains two separate code versions [36]. However for efficient recovery, we leverage the just-in-time compilation framework to iteratively invalidate all previously cached compiled code. All subsequent code invocations compile using a conservative approach for SC behavior that inserts fences around all shared memory accesses. This is a one-time recovery, and we do not switch back to the program version with optimistically inserted fences.

In summary, OPT-SC ensures SC behavior throughout program executions in all cases. When invariants hold, the fences induced using its precise predicated data-race analysis suffice to ensure SC. When an invariant rarely fails, it is detected before the invariant violation has taken effect, so that all threads can be safely paused and recovered to a version that ensures SC behavior by conservatively inserting fences.

5.4.5 Likely Invariant Violations Are Rare

Naturally, there is a strong dependency between the quality of profiling and the rate of invariant violations. Poor profiling would induce unstable invariants that fail often and thus would invalidate the benefits of COPA. In practice, our invariants meet the stability property in **O3** and are violated extremely rarely.

We leverage the extensive test suites that often ship with mature production software systems for the purpose of profiling for COPA invariants. These test suites are carefully designed to exercise a wide range of program behaviors, including representative common-case inputs as well as possible erroneous and anomalous behaviors. As such, violations of COPA invariants inferred on such test suites would indicate weaknesses in the software testing methodology and would make the case for improving these test suites. Failed invariants can even provide useful hints for generating better test cases. Profiling on existing test suites is thus adequate, and invariant failures are quite rare in practice.

One resulting design choice was not to recover the execution upon invariant failure to a conservatively optimized version as in prior COPA works [36], i.e. a program version with fences added to those potentially racy instructions as determined using a traditional sound data-race analysis without assuming the COPA invariants. Instead, we recover by switching to the `volatile`-by-default semantics [153] which conservatively guards all shared memory accesses with necessary fences, and further reduce the runtime costs associated with this recovery by leveraging the just-in-time compilation features of the modern JVM, which we further elaborate later in §5.5. This design choice not only simplifies the implementation, but is further justified as the large analysis

time spent in a conservative static analysis does not yield significant performance advantage compared to the conservative VBD version in our experience. Moreover, since invariant failures are rare in practice, we do not further optimize the associated cost of COPA recovery.

5.5 OPT-SC Implementation

In this section, we discuss the implementation of OPT-SC which primarily consists of two components: (1) the COPA static data-race analysis framework, and (2) the SC-compiler within the JVM. Our COPA analysis is implemented in the Chord analysis framework [160, 161], and the SC-compiler modifies Oracle's HotSpot JVM [164] to instrument and preserve necessary fences as well as perform COPA recovery if needed. Our SC-compiler is closely based on the VBD-HotSpot [153] that introduces the `volatile`-*by-default* semantics for Java, and we later compare against this VBD-HotSpot baseline in our evaluation.

5.5.1 Static Data-race Analysis

The target Java program is first compiled using the OpenJDK `javac` compiler and the resulting Java bytecode is used for analyses in Chord, which further uses the Joeq compiler [165] to convert the bytecode into a suitable intermediate representation for ease of analysis.

Profiling Likely Invariants The first step of COPA gathers likely invariants by observing profiling executions of the program. Here we use Chord's bytecode manipulation framework to log the necessary dynamic information for inferring our invariants in §5.3.2 by instrumenting profiling code into the original program at relevant program locations such as basic block entries, lock sites and thread creation sites. The instrumented program is then run on a set of profiling inputs and the dynamic information is recorded. In addition to profiling COPA invariants, the dynamic information collected from these executions are also used for Reflection Resolution [166]. We use this dynamic reflection resolution in subsequent analyses, as we find Chord's internal static reflection analysis to be unsound.

Predicated Data-race Analysis Chord facilitates easy extension of existing and new analyses via an interface that describes analysis operations using a declarative logic-programming language called Datalog. The datalog analyses represent various program *domains*, e.g. the set of all fields in the program, and each analysis induces *relations* among multiple program domains. For example, a typical pointer analysis computes a relation on the tuple of program variables or fields to heap memory locations. An analysis specifies rules that conditionally apply on a set of input relations to compute the output relations. Chord then iteratively applies these rules to compute the analyses relations using an efficient BDD representation-based solver.

Our predicated data-race analysis extends the default context-insensitive lockset-based data-race detector in Chord [160] as follows. First, the profiled dynamic information is imported as relations to infer the COPA likely invariants discussed in §5.3.2. Then Chord's context-insensitive pointer analysis is predicated to assume the COPA likely unused code invariants, thereby making it much more precise. The MHP analysis is then enhanced to use the precise pointer analysis results and additionally exclude false race pairs by assuming the likely singleton thread invariants. The datarace analysis then assumes the likely guarding lock invariants to deduce must-aliasing lock-sets relations that remove well-synchronized memory accesses. The resulting COPA-optimized data-race detection analysis produces a significantly more precise list of potentially racy memory accesses. The list of such memory accesses are then passed to our SC-compiler which emits the necessary fences around them to enforce SC behavior at runtime.

Likely Invariant Checks Since COPA assumes likely invariants to induce a more precise static analysis, these assumptions must be validated at runtime. So, we use the ASM bytecode manipulation framework [167] to instrument checks that dynamically validate our likely invariant assumptions. Since the bytecode is modified here, we adjust the bytecode index (BCI) accordingly in the static analysis results that are used to convey potentially racy instructions to the SC-compiler. The predicated static analysis is sound as long as the invariant checks hold and guarantees SC behavior. When an invariant rarely fails, the execution is recovered to a conservative SC approach as we describe later.

5.5.2 OPT-SC JVM Compiler

Our SC-compiler modifies the Oracle HotSpot JVM in two ways to ensure the program's SC behavior at runtime.

Instrumenting Fences Although OPT-SC operates at the Java bytecode level, the widely used OpenJDK `javac` compiler [164] that compiles Java source code to bytecode performs no optimizations that effectively reorder memory accesses [153]. So, compiling a Java program with `javac` and then executing with OPT-SC provides SC semantics for the source program at the Java language level.

The Java Virtual Machine typically executes *new* code in the interpreted mode. The interpreter executing one operation at a time naturally preserves SC at the compiler level, and we instrument fences to further ensure that the underlying hardware respects SC semantics. Moreover, bytecode-rewriting optimizations performed by the JVM interpreter never reorder memory accesses, and Java intrinsic functions never write to shared memory [153], thus ensuring SC.

Once the JVM identifies a 'hot spot' in the code, it is compiled to native code. During this compilation phase, we modify the JVM c2 (server) compiler to load the results of our static analysis and identify all potentially racy instructions. The compiler then adds the special memory-barrier nodes around them after constructing the Ideal Graph representation. This prevents subsequent optimizations from reordering memory accesses across the memory-barrier nodes. The code generation then emits the platform-specific hardware fence instructions to enforce the necessary ordering constraints during the actual execution. Enforcing this on the x86 total store order (TSO) semantics [168] requires no additional fences for load operations and requires a *StoreLoad* barrier after store operations that may potentially race to ensure that the store commits before any subsequent loads [169]. For simplicity, we only modify the c2 server compiler and invoke the JVM with the `-XX:-TieredCompilation` flag to disable tiered compilation and skip the c1 client compiler.

Recovery Using JVM De-optimization In rare execution scenarios, a COPA invariant may be violated, requiring that the execution be recovered to a conservative version adding back the fences that were removed using the optimistic reasoning. The recovery process mainly provides two functions– (1) it implements the *dynamic barrier* discussed in §5.4.4 that ensures ordering requirements are maintained during the recovery itself, and (2) it implements the mechanism that re-inserts the fences conservatively into the program.

The dynamic barrier implementation leverages the semantics of JVM's SafePoints [163]. Function call boundaries being SafePoints, the invocation of our recovery function acts as a barrier that the compiler cannot reorder memory accesses around in order to preserve the SafePoint state. The recovery mechanism is implemented as a "VM operation" which causes all running threads to be paused at JVM SafePoints before proceeding with the recovery.

Next, we switch the program version to one with conservatively inserted fences. As discussed earlier in §5.4.5, we choose to recover by switching to the `volatile`-by-default semantics [153]. To do so, we leverage the HotSpot JVM's existing de-optimization mechanism [170, §6.2]. Since we use a whole-program static analysis, the scope of our COPA optimizations is not limited, and consequently we must recover by switching the entire program to the conservative VBD semantics, which simplifies the recovery. The JVM de-optimization mechanism is invoked iteratively for each remaining class, which invalidates the compiled code cache. A global flag is set to indicate the switch to conservative mode. And all subsequent invocations as well as newly loaded classes compile conservatively with the VBD semantics. Once the de-optimization is complete, execution can resume safely since the program semantics remains unchanged and the JVM safepoints preserve the mapping from the bytecode level to the JVM execution state.

5.6 Evaluation

In this section, we show that OPT-SC can significantly reduce the performance cost of providing SC for a wide range of Java server applications. We measure the performance overheads of OPT-

SC over the unmodified JVM on several benchmark applications. Then we compare its overheads to that of VBD-Hotspot [153], a JVM compiler that enforces the volatile-by-default semantics, and S-VBD-HotSpot [154] that additionally improves this cost using speculative compilation. Our evaluation shows that-

- OPT-SC provides SC to a wide range of Java applications at only marginal overheads- avg. 9% and 5% for Dacapo benchmarks and Spark applications respectively compared to 31% and 28% respectively with VBD.
- The performance benefits of OPT-SC stem from the significantly more precise static datarace analysis using Optimistic Hybrid Analysis, which removes guarding fences from 84% more memory accesses compared to a conservative datarace analysis.
- The benefits of Optimistic Hybrid Analysis are realized with a reasonable effort in profiling, and the application performance does not suffer from invariant violations.

All experiments are run on 8 cores of an Intel Xeon E5-2620 v4 processor with hyper-threading, which provides total 16 processing units, sharing 64GB RAM and running Linux 4.18.

5.6.1 Runtime Overheads of Providing SC

Dacapo Benchmarks

We first evaluate the effectiveness of OPT-SC on the Dacapo [171] benchmark suite, a set of open-source Java applications from a wide range of application domains that is widely used to evaluate Java performance. We run our experiments on multi-threaded applications from the benchmarks suite which are compatible with the underlying Chord [160] static analysis framework.

The Optimistic hybrid analysis [1] requires a profiling phase to learn invariants. To this end, we construct a corpus of profiling and test sets, each consisting 64 inputs for each benchmark, by using the following large input sets:

- `sunflow` – curated inputs by sweeping the parameter space (e.g. input size, number of threads, pseudo-random seed).

- `lusearch` – Search novels from Project Gutenberg[172].
- `pmd` – Run the `pmd` source code analysis tool across source files in our benchmarks.
- `luindex` – Index novels from Project Gutenberg[172].
- `xalan` – Convert xhtml versions of pydoc 2.7 Webpages to XSL-FO files.

We run OPT-SC as a programmer would typically on a large set of regression tests. We first increasingly profile more executions, until the number of learned program invariants stabilizes. The static analysis is then predicated to assume the learned invariants from the profiling phase. The performance evaluations then run the default workload for each benchmark along with all inputs in our testing set. We run 10 JVM invocations for each benchmark and report the average of the 10 invocations. Upon each JVM invocation, we first run each benchmark for 5 warm-up iterations, and calculate the average runtime for the next 5 iterations.

Note that our implementation, as well as the VBD-HotSpot that we compare against, adapts only the JVM server compiler. Consequently, we disable JVM's tiered compilation feature (using `-XX:-TieredCompilation` flag), and all experiments use this configuration. The runtime overhead numbers are normalized to that of the unmodified JVM also disabling tiered compilation.

Fig. 5.4a presents the relative execution times normalized over that using the baseline unmodified HotSpot JVM. For each benchmark program, the group of bars in left-to-right order represents the overhead of the naive VBD compiler [153], the improved VBD compiler using speculative compilation optimization [154], and our OPT-SC. The naive VBD compiler incurs an average 31% overhead and upto a maximum of 78% overhead. The speculative compilation technique only brings this overhead to an average 21% and still upto 42% overhead. OPT-SC significantly reduces the overhead of enforcing SC to only 9% on average. Excluding the `luindex` benchmark for which OPT-SC incurs a slowdown of 31%, the average overhead is only $\sim 4\%$. We suspect this program sees limited benefits due to its frequent array accesses which present to be challenging to our static analysis. Furthermore, we observe that while S-VBD's speculative compilation only marginally benefits all benchmarks ($\sim 1.4\times$ speedup over VBD), OPT-SC's benefits are significantly higher ($\sim 4.4\times$ speedup over VBD).

Spark Benchmarks

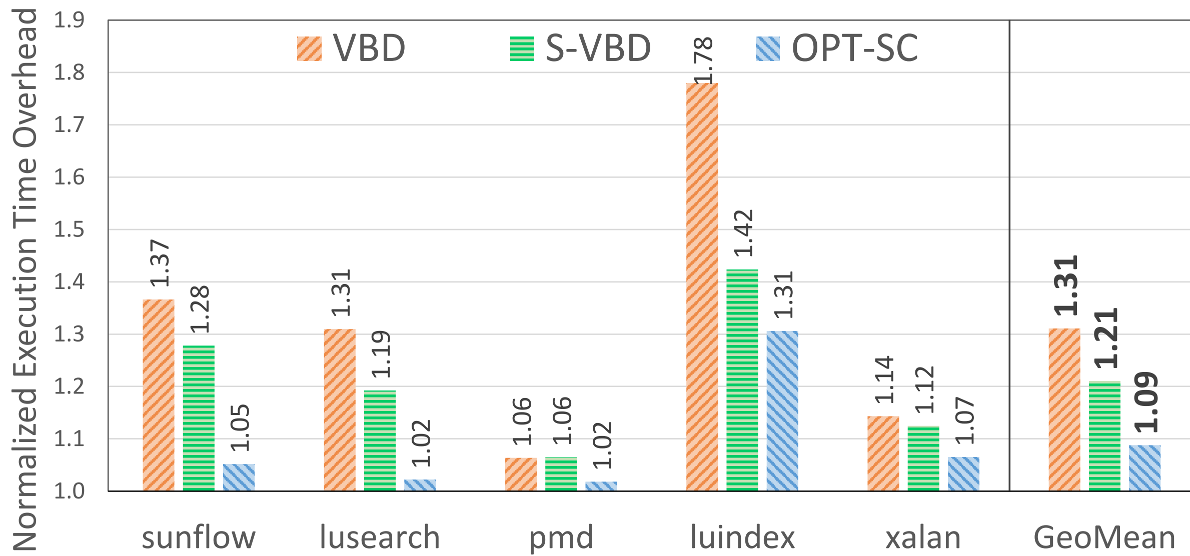
Next, we evaluate OPT-SC on Apache Spark [173], a widely used framework for big-data analytics and machine-learning tasks. Note that Spark is written in the Scala language but it compiles to the Java bytecode. So, our analysis can also be applied to such systems to extend SC guarantees. We run OPT-SC on the `spark-tests` benchmarks provided by Databricks representing several big-data analytics applications. We run Spark in standalone mode on a single machine, i.e. the driver and executors all run locally as separate processes communicating through specific ports. The spark-perf benchmarking framework runs a benchmark multiple times and reports the median execution time. We run spark-perf framework for 10 invocations and report the average of the median execution times.

Fig. 5.4b presents the relative execution times for the `spark-tests` benchmarks normalized over that with the baseline HotSpot JVM disabling tiered compilation using three JVM configurations-VBD, S-VBD, and OPT-SC respectively from left-to-right. Once again, OPT-SC significantly improves upon VBD and S-VBD bringing down the runtime overheads from an average 28% with VBD down to only 5%.

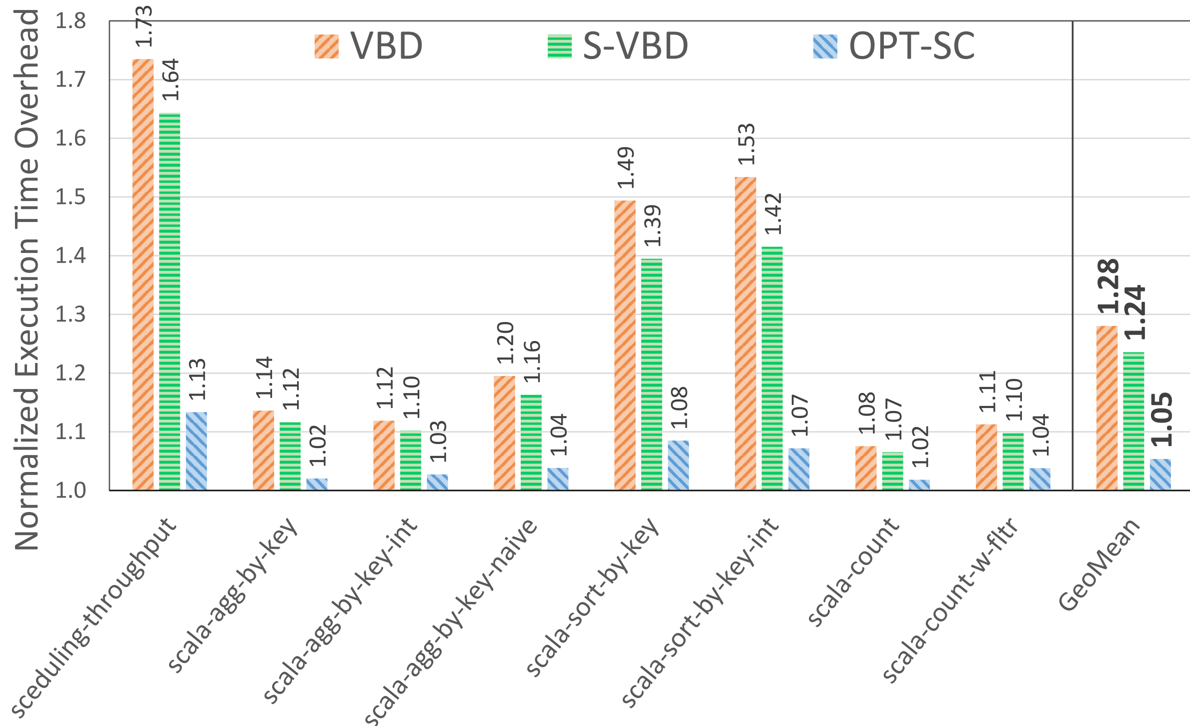
Interestingly, when comparing S-VBD over naive VBD, we note that the performance benefits of S-VBD are much less pronounced for the long-running Spark benchmarks ($\sim 1.2\times$ speedup over VBD), while OPT-SC still yields significant performance ($\sim 5\times$ speedup over VBD). S-VBD is ineffective since its speculation at the class level only pays off initially and quickly diminishes for large parts of multi-threaded programs over the course of the execution. On the other hand, OPT-SC's fine-grained invariants rarely fail and the programs enjoy the benefits of COPA during steady-state execution.

5.6.2 Precision of Static Data-race Detection

Next, to understand the sources of COPA's efficiency, we investigate how COPA's predicated analyses assuming the likely invariants affect the precision of results throughout the various phases of static data-race detection. Table 5.1 reports the total time spent in static data-race detection, the



(a) DaCapo benchmarks



(b) `spark-tests`

Figure 5.4: Result: OPT-SC reduces execution time overhead compared to VBD and S-VBD

size of the analyses result sets for the underlying pointer and MHP analyses, and the final number of potential races for each benchmark program using two configurations– the baseline **Cons** version uses the conservative analyses, and the **COPA** version predicates each analyses using the

Table 5.1: COPA improved precision of intermediate static analyses and data-race detection

Benchmark	Static Analysis Time		Pointer Analysis #pointer aliasing pairs			MHP Analysis #unordered instruction pairs			Datarace Analysis #racy instructions		
	Cons	COPA	Cons	COPA		Cons	COPA		Cons	COPA	
sunflow	3m 01s	4m 07s	47.2K	3.9K	91.6%	43.1M	140.3K	99.7%	15,440	1,589	89.7%
lusearch	1m 19s	1m 57s	26.8K	3.0K	88.8%	9.1M	56.1K	99.4%	8,369	867	89.6%
pmd	1m 08s	2m 24s	31.3K	7.2K	76.8%	7.5M	116.2K	98.4%	10,309	2,142	79.2%
luindex	1m 12s	2m 03s	26.9K	4.8K	82.3%	9.1M	1.0M	88.8%	9,029	3,189	64.7%
xalan	1m 02s	1m 30s	23.9K	3.5K	85.3%	8.8M	28.8K	99.7%	7,023	68	99.0%
Apache Spark	3m 49s	3m 13s	42.1K	2.9K	92.9%	65.0M	118.2K	99.8%	9,098	1,162	87.2%
GeoMean					**86.1%**			**97.6%**			**84.2%**

COPA invariant assumptions. The right columns (highlighted in blue) for the three results is a measure of COPA's precision over conservative analyses indicating the percentage reduction in the analysis result sets. We see that COPA's predicated analyses significantly improves the precision at each stage of the static data-race detection: by an average 86% for the pointer analysis, by 98% for the MHP analysis, and by 84% for the number of identified races. This explains the reason for COPA's improved efficiency in ensuring SC, as the precise reasoning using the COPA invariants requires 84% fewer memory accesses to be dynamically guarded by fences.

5.6.3 COPA Framework Overheads

Invariant Checking Overhead The runtime overhead incurred in checking COPA invariants is only 0.8% relative to the baseline execution on average, having virtually no effect on the performance. The likely unreachable code invariants are checked at no cost as their violations are detected upon visiting these code blocks dynamically, and checking likely singleton thread invariants incur a rather inexpensive check upon thread creation. Checks for likely guarding locks are slightly more involved but are invoked relatively infrequently upon programmer written synchronization code.

Recovery Overhead The overhead of recovering the program to conservative SC version via the JIT de-optimization mechanism is only incurred upon a rare scenario that violates an COPA invariant. In fact, none of our test programs ever violated an invariant after adequate profiling

during any dynamic execution. This indicates that our profiling methodology effectively captures the common-case dynamic execution behaviors of the programs. Furthermore, even if a one-time recovery occurs during an execution, the recovery cost gets amortized over the entire length of execution time and the performance can be no worse than that of the best available conservative SC approach.

5.6.4 Profiling Effort

COPA incurs an additional cost of profiling to learn its program invariants during the offline static analysis phase. Profiling time depends largely on the number of executions needed to profile stable invariants and the programs' typical execution times. Our test programs require profiling times ranging from 16 minutes to about 2 hours. This profiling is a one-time cost, and is part of the rigorous software testing efforts as we explained in §5.4.5 and employed for evaluating Apache Spark. Fig. 5.5 shows how the profiling phase affects the overall benefits of OPT-SC for Spark by plotting its overall runtime overhead and invariant violation rates after varying stages of profiling.

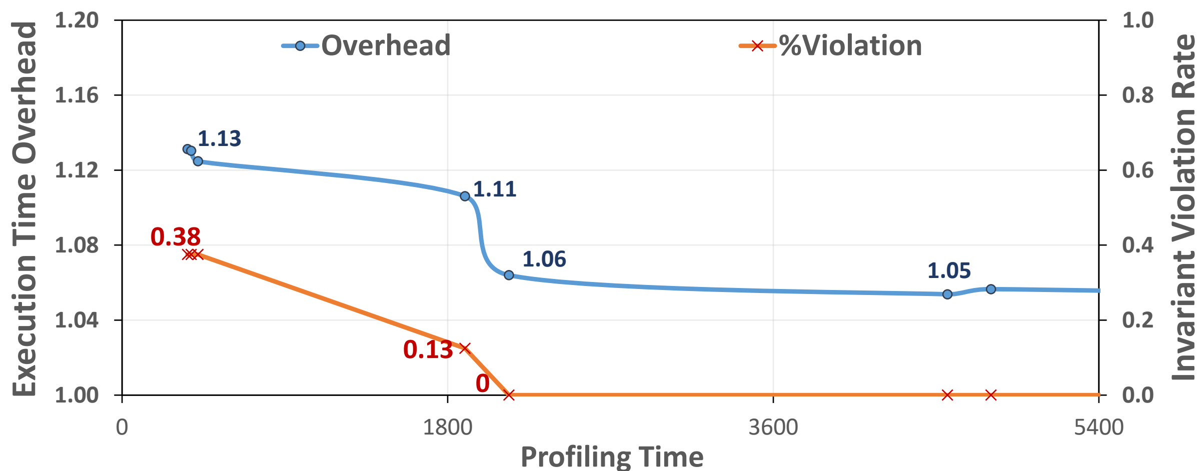


Figure 5.5: Result: OPT-SC benefits for Spark with profiling using its test suite: We divide the `mllib-tests` inputs into several batches and profile incrementally over each test input batch with the X axis showing cumulative time spent in profiling. At the end of each batch, we invoke OPT-SC with all profiled invariants. The blue line along the left vertical axis plots the normalized execution time overhead; the orange line along the right vertical axis plots the fraction of executions that encounter an invariant violation. More profiles infer stable invariants, reducing invariant violation rates, and thereby improving OPT-SC's runtime overheads.

We observe that the invariant violation rate is quite high in the beginning but quickly diminishes to zero after 36 minutes of profiling. This drop in invariant violation means that the program spends most or all of its execution time in the optimized version with optimistically elided fences. This results in a significant reduction in runtime overheads starting from 13% after 6 minutes of profiling down to only 5% in the steady-state when no invariants are violated.

Discussion

OPT-SC vs. `volatile`-by-default for Java Our work differs from the recent proposal of VBD semantics for Java [153, 154] in two distinct ways.

First, VBD ensures SC using a conservative safe-by-default approach which incurs a high performance overhead of $\sim 30\%$. We bring down the cost of providing SC using static whole-program data-flow analyses. While such heavyweight analyses notoriously do not scale well for large concurrent programs using traditional static analyses techniques, we enable this using Optimistic Hybrid Analysis thereby inducing much stronger optimizations for providing SC.

Second, the performance overheads of VBD were improved recently using a speculative compilation technique [154]. This approach first elides the fences based on a temporary assumption that object instances of a given class will not be accessed from multiple threads, and then falls back to the program version with fences added as soon as any object belonging to the class sees an access from a different thread. This saves on the synchronization costs for objects of those classes that are always accessed from a single thread. While VBD speculates at the JVM level, OHA, on the other hand, optimizes in an offline static analysis framework by assuming a much richer set of likely program invariants to improve the static reasoning in datarace detection. Although the scope and cost of OHA recovery is much higher compared to the just-in-time re-compilation for VBD, an OHA-optimized program execution rarely encounters invariant violations in the steady-state, whereas every execution of a speculatively compiled program with VBD typically sees several re-compilations in a multi-threaded execution.

5.7 Related Work

Language-Level Sequential Consistency

Recent work has demonstrated the overhead of providing SC semantics for the Haskell programming language to be negligible on commodity hardware [174]. A purely functional programming language naturally restricts conflicting memory accesses among threads and therefore supporting and reasoning for SC does not incur high overheads. The results and techniques however do not extend to imperative languages like Java.

Prior work has achieved end-to-end SC guarantees for Java [175, 176] and C [149, 177] with high efficiency by combining a cooperative compiler and specialized hardware support. Our work aims to provide SC at low costs on off-the-shelf commodity hardware by using advanced whole-program static analysis.

SC compilers for Java [178] and C [179] previously used whole-program delay-set analysis [180] to determine the required barriers to guarantee SC for a given program, but the performance overheads remained high. The technique achieved good performance, nearly that of a relaxed consistency model, when applied to a parallel dialect of Java called Titanium [158] which induces several language restrictions in order to simplify the reasoning in statically proving data-race-freedom for many memory accesses. However, the solution is not viable for the vast majority of programs that do not adhere to the additional language restrictions. Our work brings down the cost of providing SC for legacy and standard-compliant Java programs without inducing any language restrictions.

Language-Level Region Serializability

Region serializability provides a stronger memory consistency semantics than SC, whereby the program is partitioned into disjoint regions, each of which is guaranteed to execute atomically. Region serializability for C has achieved good performance, however with special-purpose hardware support [181, 182, 183]. Several works have explored region serializability for Java [184, 185, 186,

187] achieving good performance, also optimizing using a whole-program static data-race detector [185]. But these techniques incur high implementation complexity requiring code transformations to guarantee safe restarts in the event of a deadlock. In contrast, the OHA recovery is rare and fully leverages the just-in-time compilation features of the JVM.

Summary

We realize an efficient solution to enforcing language-level SC semantics for all programs, even ones with data-races, without imposing any language-level restrictions. Our approach relies on a precise static data-race detector to identify the potentially racy instructions and only guard them via hardware fence operations. Designing a precise static data-race detector remained challenging due to the fundamental imprecision of traditional static analysis. We address this problem using a new Cautiously Optimistic Program Analysis (COPA) that induces a significantly more precise static analysis by assuming likely program invariants. With reasonable profiling effort, the assumed invariants hold almost during all executions, and the execution can be recovered to a conservative SC version in the rare instances when they fail. Using this technique, we design a significantly more precise data-race detector which our OPT-SC compiler for Java uses to enforce SC during runtime at only $\sim 5\%$ overhead.

CHAPTER 6

Conclusion

With our critical infrastructures increasingly being modernized, it is essential that the underlying computer systems provide strong security and reliability guarantees. Well-known dynamic analyses techniques can enforce these security and reliability properties, but the additional work in doing so often incur prohibitively high performance overheads. So today, most of the industrial software systems run with relaxed security and reliability guarantees. As software systems grow in scope and complexity, the need for strong guarantees is over-shadowed by its performance demands.

However, performance is not necessarily at odds with security and reliability guarantees. In fact, well-behaved software in their correct executions will satisfy all security and safety properties. Such correct executions should then not require much additional work to check for these properties. So, the compiler or the language runtime should be able to remove much of the unnecessary work. Unfortunately, today's program analyses techniques are fundamentally conservative in their reasoning, as they reason for all possible execution states, correct and erroneous alike, along with many infeasible and rare execution states. This imprecise reasoning means that they cannot effectively optimize the dynamic analyses overheads.

This dissertation addresses this problem by combining static and dynamic program analyses in a novel construction of Cautiously Optimistic Program Analysis (COPA). It is founded on two principles:

Optimistic Analysis leverages assumptions about programs' dynamic behaviors to significantly improve the precision of static analyses and thereby reduce the overheads of dynamic analyses.

It first gathers *likely program invariants* from dynamic observations, which are properties that almost always hold in useful dynamic executions but are hard to prove statically. These likely invariants are then assumed in a *predicated* static analysis which reasons much more precisely, thereby identifying and eliding many more unnecessary runtime monitors from the final *optimized* dynamic analysis. The resulting system is fast and ensures the correct analysis guarantees in all executions where the assumed invariants hold true.

Cautious Reasoning additionally ensures analysis soundness in the rare event that an assumed likely invariant is violated during an execution. The precise results of the predicated static analysis are carefully used to induce only *safe elision* optimizations that do not change the analysis metadata state. Consequently, as long as the invariants hold, the optimized analysis still holds the exact analysis metadata state as in a conservative analysis. Another important property of the likely invariants is that their violations can be *detected eagerly* before they actually take effect. The analysis state is thus guaranteed to be correct when an invariant violation is detected early, so that the analysis can recover by simply switching to a conservative analysis and continue forward.

We design a simple forward recovery mechanism for the C language that statically instruments both the optimistic and conservative analyses versions in separate control flow domains. Later, we improve this mechanism for Java to leverage just-in-time compilation features.

We demonstrated the utility of COPA in three key results–

- *Live Information Flow-based Security Monitoring* : Continuous runtime monitoring of information flow can enforce several security and privacy policies. However their use is limited to offline post-mortem analysis due to the prohibitive runtime overheads ($> 500\%$ for web/email servers) of information flow tracking. COPA dramatically reduces this cost (to $\sim 9\%$) and eliminates the possibility of rollbacks to make it practical for online security analyses on live software.
- *Sound Garbage Collection (GC) for C* : Prior GCs for C only work correctly for well-behaved programs belonging to a subset of the C language, and can incorrectly reclaim memory objects that are still reachable. We design the first sound GC for C by explicitly tracking *provenance* of all pointer information during runtime. The runtime costs of tracking pointers in this manner is

greatly reduced using COPA. Our PROV-GC tool provides sound GC at only $\sim 16\%$ overhead for standard-compliant C programs.

- *Sequential Consistency (SC) for Java* : Current language standards provide weak or no semantics to a vast majority of concurrent programs with data-races. This severely complicates reasoning for correctness of programs and compilers leading to obscure bugs. The runtime cost of enforcing strong SC semantics however remained high. We leverage COPA to construct a precise static data-race detector which identifies many likely race-free memory accesses, and our OPT-SC system applying fences only around the remaining memory accesses incurs a modest $\sim 5\%$ runtime overhead on x86 hardware.

Future Directions

COPA makes an important contribution– it enables optimistic dynamic analysis without ever incurring a rollback. For well-tested software, invariants should rarely fail as profiles would have captured the common-case program states. However for moderately large software with diverse features, optimistically gathered invariants may eventually fail when the program encounters unprofiled behavior. If this happens, COPA incurs a one-way switch to a conservative hybrid analysis. So, even in the worst case, COPA is still as fast as the best available conservative hybrid technique.

We envision the following strategies to tackle the remaining challenges in deploying COPA.

Continuous Profiling : Although we demonstrate that most of COPA's benefits can be achieved from adequate profiling on regression test suites, that may not be viable in many industrial settings. Moreover, production behaviors may be very different from that explored during in-house testing. To address these constraints, COPA may be deployed in an active feedback-loop, starting with a minimal 'boot-strapping' process of initial invariants learnt from light-weight profiling. Thereafter, invariant violations in production can trigger a 'learning' phase to include the new behavior and re-analyze the program. Such a setup removes the need for an extensive a priori profiling phase and opens the possibility to actively learn new invariants and re-optimize the analysis.

Incremental Re-analysis : Upon an invariant-failure, instead of switching to the most conservatively optimized analysis, COPA can switch to a less aggressive optimistic analysis that excludes the failing invariant. If COPA could map assumed invariants to the set of induced optimizations, it could selectively disable only those optimizations induced by the violated invariant, essentially re-instrumenting the monitors that were elided by assuming that invariant. However, computing and then succinctly encoding this invariant-to-optimization mapping is challenging.

Another approach to gracefully handle invariant violations could re-analyze the program without the offending invariant. At first, it would appear to be impractical to recompile for live executions given the long time spent in static whole-program analysis for complex programs. However, for many useful static analyses, this can be done incrementally rather than redoing from scratch [188, 189]. For a dataflow analysis, this boils down to adding new nodes and edges to the programs' definition-use graph, and recomputing the transitive closure. Recent work has used incremental pointer analysis to scale whole-program analysis in the context of program modifications [190], e.g. with dynamic class loading, and this has also been applied to improve performance of an incremental data-race detector [191]. We believe our COPA analyses can similarly leverage an incremental construction in the context of invariant failures. And this re-compilation process can continue in the background while the monitored program runs slowly. Upon completion of the re-compilation process, the program can switch to the newly optimized analysis at a pre-determined safe program point. This would enable fast incremental re-analysis of the program upon an invariant violation, so the execution can be recovered to a newly optimized optimistic version instead of falling back to a conservative version.

Cautiously Optimistic Program Analysis provides a way to realize useful dynamic analyses at significantly lower overheads without sacrificing analyses' correctness. Applying this technique and building upon it in the future can realize practical tools that improve security, reliability and semantic guarantees for the rapidly growing software systems.

BIBLIOGRAPHY

[1] D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy, "Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018* (X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, eds.), pp. 348–362, ACM, 2018.

[2] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[3] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: type-safe retrofitting of legacy code," in *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pp. 128–139, 2002.

[4] D. Rhodes, C. Flanagan, and S. N. Freund, "Bigfoot: static check placement for dynamic race detection," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017* (A. Cohen and M. T. Vechev, eds.), pp. 141–156, ACM, 2017.

[5] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008* (P. Ning, P. F. Syverson, and S. Jha, eds.), pp. 39–50, ACM, 2008.

[6] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, pp. 31–40, 2010.

[7] M. S. Simpson and R. Barua, "Memsafe: ensuring the spatial and temporal memory safety of C at runtime," *Softw., Pract. Exper.*, vol. 43, no. 1, pp. 93–128, 2013.

[8] B. Dufour, B. G. Ryder, and G. Sevitsky, "Blended analysis for performance understanding of framework-based applications," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007* (D. S. Rosenblum and S. G. Elbaum, eds.), pp. 118–128, ACM, 2007.

[9] B. Dufour, B. G. Ryder, and G. Sevitsky, "A scalable technique for characterizing the usage of temporaries in framework-intensive java applications," in *Proceedings of the 16th ACM*

SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008 (M. J. Harrold and G. C. Murphy, eds.), pp. 59–70, ACM, 2008.

[10] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers, "Improving program slicing with dynamic points-to data," in *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*, pp. 71–80, ACM, 2002.

[11] S. Wei and B. G. Ryder, "Practical blended taint analysis for javascript," in *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013* (M. Pezzè and M. Harman, eds.), pp. 336–346, ACM, 2013.

[12] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA* (W. Tracz, M. Young, and J. Magee, eds.), pp. 291–301, ACM, 2002.

[13] C. Csallner, Y. Smaragdakis, and T. Xie, "Dsd-crasher: A hybrid analysis tool for bug finding," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, pp. 8:1–8:37, 2008.

[14] S. Moore and S. Chong, "Static analysis for efficient hybrid information-flow control," in *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pp. 146–160, 2011.

[15] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, pp. 85–96, 2004.

[16] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.

[17] J. Kong, C. C. Zou, and H. Zhou, "Improving software security via runtime instruction-level taint checking," in *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, ASID 2006, San Jose, California, USA, October 21, 2006*, pp. 18–24, 2006.

[18] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "LIFT: A low-overhead practical information flow tracking system for detecting security attacks," in *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA*, pp. 135–148, 2006.

[19] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, pp. 175–185, 2006.

[20] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Recent Advances in Intrusion Detection, 8th International Symposium, RAID 2005, Seattle, WA, USA, September 7-9, 2005, Revised Papers*, pp. 124–145, 2005.

[21] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *Security and Privacy in the Age of Ubiquitous Computing, IFIP TC11 20th International Conference on Information Security (SEC 2005), May 30 - June 1, 2005, Chiba, Japan*, pp. 295–308, 2005.

[22] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *21st Annual Computer Security Applications Conference (ACSAC 2005), 5-9 December 2005, Tucson, AZ, USA*, pp. 303–311, 2005.

[23] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pp. 393–407, 2010.

[24] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "RIFLE: an architectural framework for user-centric information-flow security," in *37th Annual International Symposium on Microarchitecture (MICRO-37 2004), 4-8 December 2004, Portland, OR, USA*, pp. 243–254, 2004.

[25] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: protecting sensitive data leaks using application-level taint tracking," *Operating Systems Review*, vol. 45, no. 1, pp. 142–154, 2011.

[26] J. A. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, pp. 196–206, 2007.

[27] H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pp. 116–127, 2007.

[28] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "Shadowreplica: efficient parallelization of dynamic data flow tracking," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pp. 235–246, 2013.

[29] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, "Straighttaint: decoupled offline symbolic taint analysis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pp. 308–319, 2016.

[30] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The world's fastest taint tracker," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection RAID 2011*, 2011.

[31] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pp. 317–331, 2010.

[32] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation (awarded best paper!)," in *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pp. 321–336, 2004.

[33] T. Leek, G. Baker, R. Brown, M. Zhivich, and R. Lippmann, "Coverage maximization using dynamic taint tracing," Tech. Rep. TR-1112, MIT Lincoln Laboratory, 2007.

[34] W. Masri, A. Podgurski, and D. Leon, "Detecting and debugging insecure information flows," in *15th International Symposium on Software Reliability Engineering (ISSRE 2004), 2-5 November 2004, Saint-Malo, Bretagne, France*, pp. 198–209, 2004.

[35] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot - A technique for cheap recovery," in *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pp. 31–44, 2004.

[36] S. Banerjee, D. Devecsery, P. M. Chen, and S. Narayanasamy, "Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pp. 490–504, IEEE, 2019.

[37] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pp. 75–88, 2004.

[38] "DFSan. Clang DataFlowSanitizer." `http://clang.llvm.org/docs/DataFlowSanitizer.html`.

[39] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 1999.

[40] "Google desktop - privacy policy." `http://desktop.google.com/en/privacypolicy.html`.

[41] "VRoom." `https://github.com/google/vroom`.

[42] "Run Vim Tests." `https://github.com/inkarkat/runVimTests`.

[43] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC 2006), 26-29 June 2006, Cagliari, Sardinia, Italy*, pp. 749–754, 2006.

[44] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "RAIN: refinable attack investigation with on-demand inter-process information flow tracking," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pp. 377–390, 2017.

[45] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th International Conference on Virtual Execution Environments, VEE 2012, London, UK, March 3-4, 2012 (co-located with ASPLOS 2012)*, pp. 121–132, 2012.

[46] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta, "Dynamic information flow tracking on multicores," in *Proceedings of the 2008 Workshop on Interaction between Compilers and Computer Architectures*, 2008.

[47] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis, "A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[48] J. Lee, I. Heo, Y. Lee, and Y. Paek, "Efficient dynamic information flow tracking on a processor with core debug interface," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pp. 79:1–79:6, 2015.

[49] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. P. Ryan, "Parallelizing dynamic information flow tracking," in *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*, pp. 35–45, 2008.

[50] A. Quinn, D. Devecsery, P. M. Chen, and J. Flinn, "Jetstream: Cluster-scale parallelization of information flow queries," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pp. 451–466, 2016.

[51] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn, "Parallelizing security checks on commodity hardware," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pp. 308–318, 2008.

[52] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards practical provenance tracing by alternating between logging and tainting," in *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[53] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pp. 228–241, 1999.

[54] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[55] M. D. Ernst, "Static and dynamic analysis: synergy and duality," in *ICSE WORKSHOP ON DYNAMIC ANALYSIS (WODA 2003)*, pp. 24–27, 2003.

[56] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "Taintpipe: Pipelined symbolic taint analysis," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pp. 65–80, 2015.

[57] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 30, Research Triangle Park, North Carolina, USA, December 1-3, 1997*, pp. 259–269, 1997.

[58] M. Mock, M. Das, C. Chambers, and S. J. Eggers, "Dynamic points-to sets: a comparison with static analyses and potential applications in program understanding and optimization," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, pp. 66–72, 2001.

[59] M. G. Burke, J. Choi, S. J. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The jalapeño dynamic optimizing compiler for java," in *Java Grande*, pp. 129–141, 1999.

[60] C. Chambers and D. Ungar, "Customization: Optimizing compiler technology for self, A dynamically-typed object-oriented programming language," in *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*, pp. 146–160, 1989.

[61] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pp. 133–143, 2012.

[62] T. Zhang, D. Lee, and C. Jung, "Bogo: Buy spatial memory safety, get temporal memory safety (almost) free," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), pp. 631–644, ACM, 2019.

[63] J. Vilk and E. D. Berger, "Bleak: automatically debugging memory leaks in web applications," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pp. 15–29, 2018.

[64] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *In Proc. of the Winter 1992 USENIX Conference*, pp. 125–138, 1991.

[65] D. Dhurjati, S. Kowshik, V. S. Adve, and C. Lattner, "Memory safety without runtime checks or garbage collection," in *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03). San Diego, California, USA, June 11-13, 2003*, pp. 69–80, 2003.

[66] E. D. Berger and B. G. Zorn, “Diehard: Probabilistic memory safety for unsafe languages,” in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’06, (New York, NY, USA), pp. 158–168, ACM, 2006.

[67] D. Dhurjati and V. S. Adve, “Efficiently detecting all dangling pointer uses in production servers,” in *2006 International Conference on Dependable Systems and Networks (DSN 2006), 25-28 June 2006, Philadelphia, Pennsylvania, USA, Proceedings*, pp. 269–280, 2006.

[68] J. Cohen, “Garbage collection of linked data structures,” *ACM Comput. Surv.*, vol. 13, pp. 341–367, Sept. 1981.

[69] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith, “Java without the coffee breaks: A nonintrusive multiprocessor garbage collector,” in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI ’01, pp. 92–103, 2001.

[70] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, “Softbound: highly compatible and complete spatial memory safety for c,” in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pp. 245–258, 2009.

[71] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell, “Exploring c semantics and pointer provenance,” *Proc. ACM Program. Lang.*, vol. 3, pp. 67:1–67:32, Jan. 2019.

[72] H. Boehm, “Space efficient conservative garbage collection,” in *Proceedings of the ACM SIGPLAN’93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pp. 197–206, 1993.

[73] H. Boehm, “Space efficient conservative garbage collection,” *SIGPLAN Not.*, vol. 39, pp. 490–501, April 2004.

[74] C. S. C. (WG14), “Programming languages — c (iso/iec 9899:201x),” Tech. Rep. N2310, International Organization for Standardization, Geneva, Switzerland, 2018.

[75] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, “Into the depths of C: elaborating the de facto standards,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016* (C. Krintz and E. Berger, eds.), pp. 1–15, ACM, 2016.

[76] M. Rudafshani and P. A. S. Ward, “Leakspot: detection and diagnosis of memory leaks in javascript applications,” *Softw., Pract. Exper.*, vol. 47, no. 1, pp. 97–123, 2017.

[77] G. H. Xu, M. D. Bond, F. Qin, and A. Rountev, “Leakchaser: helping programmers narrow down causes of memory leaks,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pp. 270–282, 2011.

[78] J. Clause and A. Orso, “Leakpoint: Pinpointing the causes of memory leaks,” in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE ’10, (New York, NY, USA), pp. 515–524, ACM, 2010.

[79] G. Novark, E. D. Berger, and B. G. Zorn, “Efficiently and precisely locating memory leaks and bloat,” in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pp. 397–407, 2009.

[80] M. Jump and K. S. McKinley, “Cork: Dynamic memory leak detection for garbage-collected languages,” in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’07, (New York, NY, USA), pp. 31–38, ACM, 2007.

[81] M. D. Bond and K. S. McKinley, “Bell: bit-encoding online memory leak detection,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pp. 61–72, 2006.

[82] M. Hauswirth and T. M. Chilimbi, “Low-overhead memory leak detection using adaptive statistical profiling,” in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, pp. 156–164, 2004.

[83] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, “Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[84] R. Shahriyar, S. M. Blackburn, and D. Frampton, “Down for the count? getting reference counting back in the ring,” in *International Symposium on Memory Management, ISMM ’12, Beijing, China, June 15-16, 2012*, pp. 73–84, 2012.

[85] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley, “Taking off the gloves with reference counting immix,” in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pp. 93–110, 2013.

[86] “A memory-efficient doubly linked list.” `https://www.linuxjournal.com/article/6828`, 2004.

[87] F. Henderson, “Accurate garbage collection in an uncooperative environment,” in *Proceedings of The Workshop on Memory Systems Performance (MSP 2002), June 16, 2002 and The International Symposium on Memory Management (ISMM 2002), June 20-21, 2002, Berlin, Germany*, pp. 256–263, 2002.

[88] J. Rafkind, A. Wick, J. Regehr, and M. Flatt, “Precise garbage collection for C,” in *Proceedings of the 8th International Symposium on Memory Management, ISMM 2009, Dublin, Ireland, June 19-20, 2009*, pp. 39–48, 2009.

[89] S. V. Adve and M. D. Hill, “Weak ordering - A new definition,” in *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990* (J. Baer, L. Snyder, and J. R. Goodman, eds.), pp. 2–14, ACM, 1990.

[90] T. H. Austin and C. Flanagan, “Efficient purely-dynamic information flow analysis,” in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS ’09, pp. 113–124, 2009.

[91] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[92] D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann, “Beyond the PDP-11: architectural support for a memory-safe C abstract machine,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’15, Istanbul, Turkey, March 14-18, 2015* (Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, eds.), pp. 117–130, ACM, 2015.

[93] S. Banerjee, D. Devecsery, P. M. Chen, and S. Narayanasamy, “Sound garbage collection for C using pointer provenance,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 176:1–176:28, 2020.

[94] B. Alpern, M. N. Wegman, and F. K. Zadeck, “Detecting equality of variables in programs,” in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pp. 1–11, 1988.

[95] D. D. Sleator and R. E. Tarjan, “Self-adjusting binary search trees,” *J. ACM*, vol. 32, pp. 652–686, July 1985.

[96] R. Chugh, J. W. Voung, R. Jhala, and S. Lerner, “Dataflow analysis for concurrent programs using datarace detection,” in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008* (R. Gupta and S. P. Amarasinghe, eds.), pp. 316–326, ACM, 2008.

[97] L. Effinger-Dean, H. Boehm, D. R. Chakrabarti, and P. G. Joisha, “Extended sequential reasoning for data-race-free programs,” in *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI ’11, San Jose, CA, USA, June 5, 2011* (J. S. Vetter, M. Musuvathi, and X. Shen, eds.), pp. 22–29, ACM, 2011.

[98] H. Boehm and D. Chase, “A proposal for garbage-collector-safe c compilation,” *The Journal of C Language Translation*, vol. 4, pp. 126–141, December 1992.

[99] H. Boehm, “Simple garbage-collector-safety,” in *Proceedings of the ACM SIGPLAN’96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, pp. 89–98, 1996.

[100] H. Boehm, A. J. Demers, and S. Shenker, "Mostly parallel garbage collection," in *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pp. 157–164, 1991.

[101] T. Endo, K. Taura, and A. Yonezawa, "A scalable mark-sweep garbage collector on large-scale shared-memory machines," in *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1997, November 15-21, 1997, San Jose, CA, USA*, p. 48, 1997.

[102] L. P. Deutsch and D. G. Bobrow, "An efficient, incremental, automatic garbage collector," *Commun. ACM*, vol. 19, no. 9, pp. 522–526, 1976.

[103] H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," in *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pp. 55–59, 1977.

[104] D. M. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*, pp. 157–167, 1984.

[105] G. M. Yip, "Incremental, generational mostly-copying garbage collection in uncooperative environment," Tech. Rep. Technical Report 91/8, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, June 1991.

[106] S. M. Blackburn and K. S. McKinley, "Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pp. 22–32, 2008.

[107] J. F. Bartlett, "Compacting garbage collection with ambiguous roots," *SIGPLAN Lisp Pointers*, vol. 1, pp. 3–12, Apr. 1988.

[108] J. F. Bartlett, "Mostly-copying garbage collection picks up generations and C++," Tech. Rep. Technical Note TN. 12, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, October 1989.

[109] F. Smith and J. G. Morrisett, "Comparing mostly-copying and mark-sweep conservative collection," in *International Symposium on Memory Management, ISMM '98, Vancouver, British Columbia, Canada, 17-19 October, 1998, Conference Proceedings*, pp. 68–78, 1998.

[110] A. L. Hosking, "Portable, mostly-concurrent, mostly-copying garbage collection for multi-processors," in *Proceedings of the 5th International Symposium on Memory Management, ISMM 2006, Ottawa, Ontario, Canada, June 10-11, 2006*, pp. 40–51, 2006.

[111] D. R. Edelson, "Dynamic storage reclamation in C++," Tech. Rep. Technical Report UCSC-CRL-90-19, UCSC, June 1990.

[112] D. R. Edelson and I. Pohl, "A copying collector for C++," in *Proceedings of the C++ Conference. Washington, D.C., USA, April 1991*, pp. 85–102, 1991.

[113] W. Schreiner, "RT++ – higher order threads for C++, tutorial and reference manual," Tech. Rep. Technical Report 96-9, RISC-Linz, 1996.

[114] S. L. Peyton Jones, N. Ramsey, and F. Reig, "C–: A portable assembly language that supports garbage collection," in *Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings*, pp. 1–28, 1999.

[115] D. Tarditi, P. Lee, and A. Acharya, "No assembly required: Compiling standard ML to C," *LOPLAS*, vol. 1, no. 2, pp. 161–177, 1992.

[116] S. L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler, "The glasgow haskell compiler: a technical overview," in *Proceedings of Joint Framework for Information Technology Technical Conference, Keele*, pp. 249–257, March 1993.

[117] F. Henderson, Z. Somogyi, and T. Conway, "Compiling logic programs to c using gnu c as a portable assembler," in *Proceedings of The ILPS'95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming Languages, Portland, Oregon*, 1995.

[118] M. Hirzel, A. Diwan, and J. Henkel, "On the usefulness of type and liveness accuracy for garbage collection and leak detection," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 6, pp. 593–624, 2002.

[119] D. Jung, S. Bae, J. Lee, S. Moon, and J. K. Park, "Supporting precise garbage collection in java bytecode-to-c ahead-of-time compiler for embedded systems," in *Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2006, Seoul, Korea, October 22-25, 2006*, pp. 35–42, 2006.

[120] J. Baker, A. Cunei, F. Pizlo, and J. Vitek, "Accurate garbage collection in uncooperative environments with lazy pointer stacks," in *Compiler Construction, 16th International Conference, CC 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 26-30, 2007, Proceedings*, pp. 64–79, 2007.

[121] R. Shahriyar, S. M. Blackburn, and K. S. McKinley, "Fast conservative garbage collection," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pp. 121–139, 2014.

[122] G. E. Collins, "A method for overlapping and erasure of lists," *Commun. ACM*, vol. 3, no. 12, pp. 655–657, 1960.

[123] R. D. Lins, "Cyclic reference counting with lazy mark-scan," *Inf. Process. Lett.*, vol. 44, no. 4, pp. 215–220, 1992.

[124] D. F. Bacon and V. T. Rajan, "Concurrent cycle collection in reference counted systems," in *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, pp. 207–235, 2001.

[125] Apple, "Transitioning to ARC release notes." `https://developer.apple.com/library/archive/releasenotes/ObjectiveC/RN-TransitioningToARC`, 2013.

[126] B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, J. E. Maste, A. Mazzinghi, E. T. Napierala, R. M. Norton, M. Roe, P. Sewell, S. D. Son, and J. Woodruff, "Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pp. 379–393, 2019.

[127] R. W. M. Jones and P. H. J. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," in *AADEBUG*, pp. 13–26, 1997.

[128] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*, 2004.

[129] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pp. 51–66, 2009.

[130] R. Bodík, R. Gupta, and V. Sarkar, "ABCD: eliminating array bounds checks on demand," in *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pp. 321–333, 2000.

[131] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, pp. 275–288, 2002.

[132] D. Tarditi, A. S. Elliott, A. Ruef, and M. Hicks, "Checked c: Making c safe by extension," in *IEEE Cybersecurity Development Conference 2018*, pp. 53–60, IEEE, September 2018.

[133] Microsoft, "Managed Extensions for C++." `https://docs.microsoft.com/en-us/cpp/build/reference/microsoft-extensions-to-c-and-cpp`, 2004.

[134] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, pp. 290–301, 1994.

[135] J. Sparud, "Fixing some space leaks without a garbage collector," in *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, pp. 117–124, 1993.

[136] D. E. Evans, "Static detection of dynamic memory errors," in *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, pp. 44–53, 1996.

[137] N. Dor, M. Rodeh, and S. Sagiv, "Detecting memory errors via static pointer analysis (preliminary experience)," in *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE '98, Montreal, Canada, June 16, 1998*, pp. 27–34, 1998.

[138] C. Ding and Y. Zhong, "Compiler-directed run-time monitoring of program data access," in *Proceedings of The Workshop on Memory Systems Performance (MSP 2002), June 16, 2002 and The International Symposium on Memory Management (ISMM 2002), June 20-21, 2002, Berlin, Germany*, pp. 1–12, 2002.

[139] D. L. Heine and M. S. Lam, "A practical flow-sensitive and context-sensitive C and C++ memory leak detector," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pp. 168–181, 2003.

[140] J. Lee, T. Avgerinos, and D. Brumley, "TIE: principled reverse engineering of types in binary programs," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[141] J. Caballero and Z. Lin, "Type inference on executables," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 65:1–65:35, 2016.

[142] K. Elwazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pp. 51–60, 2013.

[143] M. Burrows, S. N. Freund, and J. L. Wiener, "Run-time type checking for binary programs," in *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pp. 90–105, 2003.

[144] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Computers*, vol. 28, no. 9, pp. 690–691, 1979.

[145] H. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008* (R. Gupta and S. P. Amarasinghe, eds.), pp. 68–78, ACM, 2008.

[146] J. Manson, W. Pugh, and S. V. Adve, "The java memory model," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005* (J. Palsberg and M. Abadi, eds.), pp. 378–391, ACM, 2005.

[147] W. Pugh, "Fixing the java memory model," in *Proceedings of the ACM 1999 Conference on Java Grande, JAVA '99, San Francisco, CA, USA, June 12-14, 1999* (G. C. Fox, K. E. Schauser, and M. Snir, eds.), pp. 89–98, ACM, 1999.

[148] W. Pugh, "The java memory model is fatally flawed," *Concurr. Pract. Exp.*, vol. 12, no. 6, pp. 445–455, 2000.

[149] D. Marino, A. Singh, T. D. Millstein, M. Musuvathi, and S. Narayanasamy, "A case for an sc-preserving compiler," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011* (M. W. Hall and D. A. Padua, eds.), pp. 199–210, ACM, 2011.

[150] D. R. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003* (M. L. Scott and L. L. Peterson, eds.), pp. 237–252, ACM, 2003.

[151] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey, "Racerd: compositional static race detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 144:1–144:28, 2018.

[152] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: static race detection on millions of lines of code," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007* (I. Crnkovic and A. Bertolino, eds.), pp. 205–214, ACM, 2007.

[153] L. Liu, T. D. Millstein, and M. Musuvathi, "A volatile-by-default JVM for server applications," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 49:1–49:25, 2017.

[154] L. Liu, T. D. Millstein, and M. Musuvathi, "Accelerating sequential consistency for java with speculative compilation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019* (K. S. McKinley and K. Fisher, eds.), pp. 16–30, ACM, 2019.

[155] C. S. C. (WG21), "Programming languages — c++," Tech. Rep. N4849, International Organization for Standardization, Geneva, Switzerland, 2020.

[156] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007* (J. Ferrante and K. S. McKinley, eds.), pp. 22–31, ACM, 2007.

[157] H. Boehm, "How to miscompile programs with "benign" data races," in *3rd USENIX Workshop on Hot Topics in Parallelism, HotPar'11, Berkeley, CA, USA, May 26-27, 2011* (M. McCool and M. Rosenblum, eds.), USENIX Association, 2011.

[158] A. Kamil, J. Su, and K. A. Yelick, "Making sequential consistency practical in titanium," in *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA, CD-Rom*, p. 15, IEEE Computer Society, 2005.

[159] E. Duesterwald and M. L. Soffa, "Concurrency analysis in the presence of procedures using a data-flow framework," in *Proceedings of the Symposium on Testing, Analysis, and Verification, TAV 1991, Victoria, British Columbia, Canada, October 8-10, 1991* (W. E. Howden, ed.), pp. 36–48, ACM, 1991.

[160] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006* (M. I. Schwartzbach and T. Ball, eds.), pp. 308–319, ACM, 2006.

[161] M. Naik and A. Aiken, "Conditional must not aliasing for static race detection," in *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007* (M. Hofmann and M. Felleisen, eds.), pp. 327–338, ACM, 2007.

[162] A. Ragozin, "Safepoints in HotSpot JVM." `http://blog.ragozin.info/2012/10/safepoints-in-hotspot-jvm.html`, 2012.

[163] A. Gupta, "Under the hood JVM: Safepoints." `https://medium.com/software-under-the-hood/under-the-hood-java-peak-safepoints-dd45af07d766`, 2017.

[164] "OpenJDK." `http://openjdk.java.net`, .

[165] "Joeq Java compiler framework." `http://joeq.sourceforge.net`, .

[166] V. B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for java," in *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings* (K. Yi, ed.), vol. 3780 of *Lecture Notes in Computer Science*, pp. 139–160, Springer, 2005.

[167] "ASM Java bytecode manipulation and analysis framework." `https://asm.ow2.io`, .

[168] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: x86-tso," in *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings* (S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, eds.), vol. 5674 of *Lecture Notes in Computer Science*, pp. 391–407, Springer, 2009.

[169] "The JSR-133 Cookbook for Compiler Writers." `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`, .

[170] T. Kotzmann, *Escape Analysis in the Context of Dynamic Compilation and Deoptimization*. PhD thesis, Johannes Kepler University Linz, 10 2005.

[171] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: java benchmarking development and analysis," in *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA* (P. L. Tarr and W. R. Cook, eds.), pp. 169–190, ACM, 2006.

[172] "Project Gutenberg." `http://www.gutenberg.org`, .

[173] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: a unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[174] M. Vollmer, R. G. Scott, M. Musuvathi, and R. R. Newton, "Sc-haskell: Sequential consistency in languages that minimize mutable shared heap," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017* (V. Sarkar and L. Rauchwerger, eds.), pp. 283–298, ACM, 2017.

[175] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: bulk enforcement of sequential consistency," in *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA* (D. M. Tullsen and B. Calder, eds.), pp. 278–289, ACM, 2007.

[176] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J. Lee, X. Fang, S. P. Midkiff, and D. C. Wong, "Bulkcompiler: high-performance sequential consistency through cooperative compiler and hardware support," in *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA* (D. H. Albonesi, M. Martonosi, D. I. August, and J. F. Martínez, eds.), pp. 133–144, ACM, 2009.

[177] A. Singh, S. Narayanasamy, D. Marino, T. D. Millstein, and M. Musuvathi, "End-to-end sequential consistency," in *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, pp. 524–535, IEEE Computer Society, 2012.

[178] Z. Sura, X. Fang, C. Wong, S. P. Midkiff, J. Lee, and D. A. Padua, "Compiler techniques for high performance sequentially consistent java programs," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA* (K. Pingali, K. A. Yelick, and A. S. Grimshaw, eds.), pp. 2–13, ACM, 2005.

[179] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, "Don't sit on the fence - A static analysis approach to automatic fence insertion," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings* (A. Biere and R. Bloem, eds.), vol. 8559 of *Lecture Notes in Computer Science*, pp. 508–524, Springer, 2014.

[180] D. E. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, pp. 282–312, 1988.

[181] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H. Boehm, "Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races," in *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France* (A. Seznec, U. C. Weiser, and R. Ronen, eds.), pp. 210–221, ACM, 2010.

[182] D. Marino, A. Singh, T. D. Millstein, M. Musuvathi, and S. Narayanasamy, "DRFX: a simple and efficient memory model for concurrent programming languages," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010* (B. G. Zorn and A. Aiken, eds.), pp. 351–362, ACM, 2010.

[183] A. Singh, D. Marino, S. Narayanasamy, T. D. Millstein, and M. Musuvathi, "Efficient processor support for drfx, a memory model with exceptions," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011* (R. Gupta and T. C. Mowry, eds.), pp. 53–66, ACM, 2011.

[184] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia, "Valor: efficient, software-only region conflict exceptions," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015* (J. Aldrich and P. Eugster, eds.), pp. 241–259, ACM, 2015.

[185] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni, "Hybrid static: Dynamic analysis for statically bounded region serializability," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015* (Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, eds.), pp. 561–575, ACM, 2015.

[186] A. Sengupta, M. Cao, M. D. Bond, and M. Kulkarni, "Toward efficient strong memory model support for the java platform via hybrid synchronization," in *Proceedings of the Principles and Practices of Programming on The Java Platform, PPPJ 2015, Melbourne, FL, USA, September 8-11, 2015* (R. Stansifer and A. Krall, eds.), pp. 65–75, ACM, 2015.

[187] M. Zhang, S. Biswas, and M. D. Bond, "Avoiding consistency exceptions under strong memory models," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management, ISMM 2017, Barcelona, Spain, June 18, 2017* (C. M. Kirsch and B. L. Titzer, eds.), pp. 115–127, ACM, 2017.

[188] B. G. Ryder, "Incremental data flow analysis," in *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983* (J. R. Wright, L. Landweber, A. J. Demers, and T. Teitelbaum, eds.), pp. 167–176, ACM Press, 1983.

[189] M. G. Burke, "An interval-based approach to exhaustive and incremental interprocedural data-flow analysis," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 341–395, 1990.

[190] B. Liu, J. Huang, and L. Rauchwerger, "Rethinking incremental and parallel pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 1, pp. 6:1–6:31, 2019.

[191] Y. Li, B. Liu, and J. Huang, "SWORD: a scalable whole program race detector for java," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019* (J. M. Atlee, T. Bultan, and J. Whittle, eds.), pp. 75–78, IEEE / ACM, 2019.