Enabling Program Analysis Through Deterministic Replay and Optimistic Hybrid Analysis

by

David Devecsery

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
2018

Doctoral Committee:

Professor Peter M. Chen, Chair
Professor Jason Flinn
Professor Stéphane Lafortune
Associate Professor Satish Narayanasamy

David Devecsery

ddevec@umich.edu

ORCID iD: 0000-0001-7574-9321

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

As software continues to evolve, software systems increase in complexity. With software systems composed of many distinct but interacting components, today's system programmers, users, and administrators find themselves requiring automated ways to find, understand, and handle system mis-behavior. Recent information breaches such as the Equifax breach of 2017, and the Heartbleed vulnerability of 2014 show the need to understand and debug prior states of computer systems.

In this thesis I focus on enabling practical entire-system retroactive analysis, allowing programmers, users, and system administrators to diagnose and understand the impact of these devastating mishaps. I focus primarly on two techniques. First, I discuss a novel deterministic record and replay system which enables fast, practical recollection of entire systems of computer state. Second, I discuss optimistic hybrid analysis, a novel optimization method capable of dramatically accelerating retroactive program analysis.

Record and replay systems greatly aid in solving a variety of problems, such as fault tolerance, forensic analysis, and information providence. These solutions, however, assume ubiquitous recording of any application which may have a problem. Current record and replay systems are forced to trade-off between disk space and replay speed. This trade-off has historically made it impractical to both record and replay large histories of system-level computation. I present Arnold, a novel record and replay system which efficiently records years of computation on a commodity hard-drive, and can efficiently replay any recorded information. Arnold combines caching with a unique process-group granularity of recording to produce both small, and quickly recalled recordings. My experiments show that under a desktop workload, Arnold could store 4 years of computation on a commodity

4TB hard drive.

Dynamic analysis is used to retroactively identify and address many forms of system mis-behaviors including: programming errors, data-races, private information leakage, and memory errors. Unfortunately, the runtime overhead of dynamic analysis has precluded its adoption in many instances. I present a new dynamic analysis methodology called optimistic hybrid analysis (OHA). OHA uses knowledge of the past to predict program behaviors in the future. These predictions, or *likely invariants* are speculatively assumed true by a static analysis. This creates a static analysis which can be far more accurate than its traditional counterpart. Once this *predicated static analysis* is created, it is speculatively used to optimize a final dynamic analysis, creating a far more efficient dynamic analysis than otherwise possible. I demonstrate the effectiveness of OHA by creating an optimistic hybrid backward slicer, OptSlice, and optimistic data-race detector OptFT. OptSlice and OptFT are just as accurate as their traditional hybrid counterparts, but run on average 8.3x and 1.6x faster respectively.

In this thesis I demonstrate that Arnold's ability to record and replay entire computer systems, combined with optimistic hybrid analysis's ability to quickly analyze prior computation, enable a practical and useful entire system retroactive analysis that has been previously unrealized.

CHAPTER I

Introduction

Today, software is ubiquitous, controlling and managing many important aspects of our daily lives. It drives critical aspects of modern society, such as financial transactions, medical devices, and transportation. The internet has become so ingrained into our daily lives that in 2014 when Facebook experienced a brief outage multiple people called 911 [78]. Given the tasks we entrust to software, the importance of having reliable software systems is critical. Beyond the potentially life-threatening consequences of software failures in systems such as airplanes, software failures can also have devastating fiscal effects. According to a report by Gartner Group[99] the average cost of an hour of downtime for a financial company exceeds six million US dollars.

As software continues to evolve, software systems increase in complexity. With software systems composed of many distinct, but interacting components, today's system programmers, users, and administrators find themselves requiring automated ways to find, understand, and handle system mis-behaviors.

Unfortunately, as we have come to rely on software more, we have also demanded more complexity from our software systems. Chou et al. [30] found that over a seven year time period, the size of the Linux kernel source grew by a factor of 16. Jim Gray notes a 20% growth per year in his classic software reliability study [51]. With this rapid growth in code size, the complexities of maintaining, managing, and understanding how complex software

systems operate become impractical for software users, developers, and administrators. Li et al. [70] state that over 80% of bugs they study come from programmers' misunderstanding how their code works in the larger system. Lu et al. [73] note that 74% of concurrency bugs are incorrectly fixed on their first attempt, indicating programmers don't understand exactly how the code is actually operating.

Fortunately, today there are many techniques to automate the process of understanding and detection of system mis-behaviors. In this thesis we explore two such techniques: deterministic record and replay and dynamic program analysis. This thesis focuses on the challenges of practically applying these techniques toward solving the problem of retroactive analysis. Thus, the following statement summarizes my thesis:

Deterministic record/replay and optimistic hybrid analysis make retroactive program analysis practical for entire computer systems.

We additionally observe deterministic record and replay, and dynamic analysis enjoy a mutualistic relationship. Deterministic record and replay allows analysis to be run offline, enabling many dynamic analyses which would otherwise be too expensive. Furthermore, dynamic analyses can observe and constrain execution environments, making it possible to record executions which would otherwise be too expensive.

1.1 Deterministic Record and Replay

There are many instances where computer systems can strongly benefit from knowledge of the past. Debugging, system crashes, and security violations are common problems in today's complex systems. When these issues occur, users, programmers, and administrators often wish to identify the problem, recover, and learn the consequences of the system misbehavior. Unfortunately, by the time an issue is discovered, the computation required to fully understand the consequences of the issue is often discarded or overwritten. Today's debugging systems frequently employ ad-hoc methods to preserve a limited history, such as logging, but these systems are limited to pre-determined sets of data, often inadequate to

diagnose serious problems. A more comprehensive solution is that of deterministic record and replay.

Deterministic record and replay records an entire system's execution, allowing any prior state to be faithfully reproduced and analyzed. In many instances, deterministic record and replay has been used to solve many complex problems such as crash consistency [22, 43], forensic analysis [63, 64, 39], debugging [47, 67, 13, 11], and information providence [64]. Unfortunately, in order for solutions using a record and replay system to be adopted, it must be practical to both record and replay any execution needing analysis.

While it is practical to record and replay small systems, or isolated executions, recording the large computing environments used by many modern systems remains impractical. Record and replay systems must select a granularity of recording, for instance recording each process independently. This process-level record and replay system would record each process on a system independently, allowing a user to replay any recorded process in isolation. This results in a reasonable replay time, but recording processes in isolation often consumes considerable disk space. Furthermore, tracing bugs through the system becomes challenging, as the recording does not preserve inter-process communication dependencies. Instead a record and replay system could choose to record at the granularity of the entire system. This would resolve the inter-process communication, and disk problems, but would do so at the cost of replaying the data. To get a single byte of information from a single process the entire system would have to be replayed, resulting in large amounts of wasted effort. This trade-off between recording space and replaying time has limited the use of deterministic record and replay on large systems where both recording overhead and replay time must both be reasonable.

We introduce the idea of *eidetic systems*. Eidetic systems are motivated by eidetic visions; an eidetic vision is a vision so realistic and lifelike that it can be recalled as if it is actually being re-lived. An eidetic system is similarly a computer system which can record and replay any prior computation, just as if it were happening again. In this chapter we

discuss *Arnold*, the first practical eidetic system. Arnold is a novel record and replay system which efficiently records and replays years of computation for an entire desktop system on a commodity hard-drive. It combines caching with a unique process-group granularity of recording to produce both small, and quickly recalled recordings. Our experiments show that under a desktop workload, Arnold can store and recall 4 years of computation on a commodity 4TB hard drive.

1.2 Optimistic Hybrid Analysis

Dynamic analysis is another tool which has often been used to solve issues relating to system reliability and security. Dynamic analyses improve reliability by checking for common program errors such as memory errors [87, 14, 60, 88], concurrency bugs [98, 42, 48, 74, 75], and many other common programming errors [27, 46]. Additionally, dynamic analyses are used to help programmers debug and reason about programs though techniques like program slicing [111, 105], and profiling [26, 81, 104]. Finally, dynamic analyses have proven useful in aiding with security enforcement with tools such as dynamic taint trackers [44, 63, 89, 33], and malware detection [20, 41].

Although these tools automatically identify many issues, their use is often precluded by the unreasonable overhead associated with the analysis. Li et al. [70] note that of the bugs they study which are not semantic bugs, the majority could be detected with a simple dynamic memory checker. In order for these tools to be useful, they must be practical for users to run.

The most optimized dynamic analyses today are *hybrid static/dynamic analyses*. These analyses statically analyze a program's source code to prove properties about an execution, then use these properties to optimize some runtime checks during dynamic analysis. Conventionally hybrid analyses require sound static analysis to guarantee that any removed checks do not sacrifice the soundness of the final result. Unfortunately, due to the conservative nature of the static analysis, this is often still not enough to make dynamic analysis

practical to run ubiquitously.

In this thesis, we present a new dynamic analysis methodology called *optimistic hybrid analysis* (OHA). OHA uses knowledge of the past to predict program behaviors in the future. These predictions, or *likely invariants* are speculatively assumed true by a static analysis. This creates a static analysis which can be far more accurate than its traditional counterpart. Once this *predicated static analysis* is created, it is speculatively used to optimize a final dynamic analysis, much as is done in traditional hybrid analysis, creating a far more efficient dynamic analysis than otherwise possible. We demonstrate the effectiveness of OHA by creating an optimistic hybrid backward slicer, OptSlice and data-race detector, OptFT. OptSlice and OptFT are just as accurate as a traditional dynamic backward slicer, but runs on average 8.3x and 1.6x faster respectively.

1.3 Optimizing Rollbacks for Optimistic Hybrid Data-Race Detection

OHA suffers from poor worst-case performance, limiting its applicability to long-running applications, or to production code. This high-overhead worst-case scenario stems from OHA's inability to bound rollbacks on likely invariant mis-speculation. In this work, we focus on the challenges of identifying points within an execution an OHA can rollback to, and we discuss methods to help enable this bounded rollback.

Unfortunately, bounding rollbacks generally is very challenging with many naive solutions the overhead for bounding rollbacks approaching that of an unoptimized (traditional hybrid) query. To overcome many of these challenges, we focus at an analysis specific layer, leveraging properties of the analysis along with general techniques to bound the rollback of OHA.

This section introduces OptFT-BR, an extension of my optimistic hybrid data-race detector, OptFT, which optimizes and reduces the duration OptFT must rollback on mis-speculations. OptFT-BR uses three techniques to efficiently identify potential rollback points. First, OptFT-BR is analysis specific, and leverages properties from data-race de-

tection to quickly identify program properties that enable rollback. Second, OptFT-BR leverages a weaker form of rollback recovery we call analysis equivalence. Finally, OptFT-BR identifies potential rollback and recovery segments we call *recovery regions*. Recovery regions are segments of code over which an OHA will recover an equivalent analysis state when executed. By identifying recovery regions for race-detection, OptFT-BR is able to identify far more potential rollback locations than if it were to just look for individual rollback points. Using these techniques, OptFT-BR shows considerably faster rollback recovery, than OptFT, with with very little effect to common-case performance.

CHAPTER II

Related Works

Throughout this thesis I build upon two powerful tools used in many systems today: deterministic record and replay, and dynamic program analysis. In this chapter I discuss a brief history of these techniques, as well as their uses.

2.1 Deterministic Record and Replay

The ultimate goal of a deterministic record and replay system is to allow a computer system to recreate any state, such as memory and register states, present in a recorded execution. This is accomplished by logging all non-determinism entering the execution, such that by re-running the same computation, with the same set of non-deterministic inputs, computation produces the same outputs. The observation behind this technique is that the vast majority of computation performed on a computer system is deterministic, with only a few non-deterministic events which must be recorded. In this section I will briefly describe the evolution of record and replay systems, along with a summary of their uses.

2.1.1 Uses of Deterministic Replay

Deterministic record and replay systems have been used to solve a diverse set of problems. In this section I will discuss some of the general problems solved by them in the areas of debugging, security, and reliability.

Debugging Debugging is naturally aided by record and replay systems, as they provide the ability to reliably reproduce a bug. Researchers, however, have also used deterministic record and replay to iteratively recreate prior states, presenting the illusion of reverse execution. This allows programmers to examine when prior state changed [47, 67]. Replay has also been used to help debug configuration errors [13, 11].

Security Replay has also been applied to the realm of security. ReVirt uses virtual machine record and replay to log and audit system compromises [39]. Retro uses deterministic record and replay to help recover from intrusions [64]. Systems such as POIROT use deterministic record and replay to minimize web-server security auditing times [63]. Record and replay is also used to audit untrusted computation, for instance, game cheat detection systems [62].

Reliability Finally, deterministic record and replay has been used in creating reliable systems. Early virtual machine record and replay systems were used for virtual machine replication [22]. Record and replay has also been used for rollback-recovery systems [43], and to accelerate virtual machine migration [71].

2.1.2 Prior Record and Replay Systems

Due to their wide array of uses, many researchers have worked on and developed many deterministic record and replay systems. Due to the breadth of depth of the field, it would be impractical to cover all deterministic record/replay systems here. Instead I overview these works, organized by the type of execution (e.g. single process vs entire machine) the record and replay system was designed to handle.

2.1.2.1 Single Process Record/Replay

Single Thread The earliest record and replay systems focused on reproducing states within individual processes for debugging [47]. These systems were typically focused at

recording and replaying a single-threaded process on a uni-core processor, recording any non-deterministic input to that process.

Multi-Thread As multi-threaded programs became more prevalent, systems began focusing on the added challenging of capturing the non-determinism created by interacting threads. Researchers resolved these problems by recording thread interaction for individual processors either at the data granularity [91], or at the thread-interleaving level [103, 96], and replaying on a uni-core.

Unlike uni-core processors, multi-core processor memory interleavings cannot easily be summarized by thread schedules. With the onset of multi-core architectures, record/replay systems began to adopt a wide array of techniques to tackle the challenge of recording and replaying programs with data-races between memory accesses. Multi-core record and replay systems today adopt a wide array of techniques to record a single multi-threaded process, such as memory access monitoring [68], relaxing replay constraints [9], using speculation to emulate uni-core processors [107], and using static analysis to optimize memory access recordings [69].

2.1.2.2 Entire Machine Record/Replay

The second most common record and replay granularity in literature is that of an entire machine. This is typically preformed by recording all of the non-determinism, such as disk inputs, network inputs and interrupt timings, through a virtual-machine manager (VMM), or hypervisor, [22, 39] for uni-core processors. Unfortunately, memory access orderings on multi-core processors are non-deterministic, and existing software-only multi-core record and replay systems suffer significant overheads attempting to record them [40]. As software only solutions for recording memory orderings on multi-processor systems are very expensive, researchers have focused on a variety of hardware solutions [15, 112, 59, 82, 86, 108].

2.1.2.3 Other Record/Replay Granularities

Finally, researchers have looked at recording and replaying at other granularities within software systems. Some researchers have recorded at sub-process levels, such as recording within the Java Virtual Machine [8].

Researchers in the area of distributed systems have focused on recording and replaying groups of communicating processes for fault-tolerance purposes [43]. These systems record process groups by maintaining determinant logs of communication between collaborating processes, using replay to recover from selected checkpoints on system crashes.

2.1.2.4 Eidetic Systems

Prior systems have often focused first and foremost at recording some set of prior computation, be it a process, or a whole machine. However, none of these systems evaluate the trade-off between the recording granularity and the space and time it takes to recall any individual program state. For instance, an entire machine record-replay system must replay the entire virtual-machine's state in order to recover a single byte of data within a process. A process based recording can be replayed more efficiently (a single process instead of all processes), however, it is fundamentally larger, as all inputs non-deterministic to the *process* must be recorded. Even the distributed protocols, which record at a process group granularity, only consider one granularity, only record a limited history of the process, and use complex garbage collection algorithms to compensate for the space their checkpoints and determinant logs consume. In this thesis I focus on a record and replay system which aims to record large systems, with thousands of processes, yet still maintains both reasonable recording disk-space and replay overhead.

2.2 Optimistic Hybrid Analysis

Dynamic analysis is the analysis of computer software that is performed by executing programs. It is among the most foundational, widely used, and powerful computing techniques we employ today. Typically a dynamic analysis will add instrumentation to an executing program, allowing the analysis code to analyze and determine specific properties about a programs execution. For example a memory error checker may add bounds checks before each array access. Unfortunately, the weakness of dynamic analysis is the overhead introduced by the instrumentation required to analyze the program. In our memory checking example, performing repeated bounds checks may be an unacceptable overhead, causing some languages, like c, to forego this critical safety analysis. Throughout the remainder of this section I will present an overview of how dynamic analysis has been used, as well as common techniques used to accelerate dynamic analyses.

2.2.1 Dynamic Analysis Uses

As dynamic analysis is such a general technique, it would be impractical to discuss all dynamic analyses in this section. Since the goal of this work is to improve the efficiency of dynamic analysis, I will focus on some of the most popular analyses, whose use is often limited due to the high overhead of instrumentation.

Common Programming Errors Many dynamic analyses aim to detect common programming errors. They include memory error detectors [87, 14, 60, 88]. These systems typically introduce runtime instrumentation around memory accesses to ensure they are safe. This includes protections from use-after-free, double-free, buffer overflow, and null-pointer dereferences. Other common runtime checkers include type checking [27, 58, 5]. These checkers ensure that the runtime system follows a valid typing system, lest object accesses may have unintended consequences. Additionally, programmers have used dynamic analysis to both discover, and check program invariants [46]. These invariants can be either

verified by programmers, or dynamically checked on future runs to detect other errors.

Concurrency Bugs Concurrent shared-memory programs introduce an exponential number of program-state access interleavings not possible within a single-threaded application. This explosion of states can make a programmer's job of reasoning about, even relatively simple seeming, applications unreasonable. To help handle dealing with this complexity, programmers can employ dynamic concurrency tools.

Data-race detectors attempt to find any data races, or unordered memory operations, at least one of which is a write, within a program. There are two primary detector types, lockset [98, 42], and vector-clock based [48]. Vector-clock based detectors monitor all synchonization and memory operations, verifying that no two memory accesses, at least one of which is a write, are unordered. These detectors have no false positives, but often run more slowly than their lockset based counterparts. Lockset detectors, instead of detecting data-races directly, verify that shared variables are synchronized with a common locking pattern. This detection method has several benefits, such as traditionally lower runtime overhead and the ability to find possible races which don't happen in the current execution, but they can produce false positives.

Beyond data-race detection, others have looked at detecting atomicity violations [74, 75]. These are violations in which all memory accesses are properly ordered, but the ordering is not restrictive enough. This means that by re-ordering protected regions the program will exhibit buggy behavior.

Program Understanding There are entire bodies of research devoted to how dynamic analysis helps programmers understand how their programs operate [34]. In this section, I will focus on two similar, information-flow based approaches, slicing and flowback analysis.

Slicing was originally developed by Mark Weiser [111]. Slicing takes as input a minimum set of program behaviors, and reduces the program into a minimal program required

to produce that behavior. This subset of the original program is more succinct than the original, and often easier to understand. While Weiser discusses the notion of finding a slice statically, there are many works which find slices dynamically [80, 6, 105]. Flowback analysis is another similar analysis [16], which reports the entire dynamic program flow up to a specified instruction. While both of these dynamic tools can greatly aid programmers in understanding code and bug finding, they are not typically used in practice, largely due to their very high overhead.

Security Dynamic analysis has long been applied to the realm of security. Beyond the added security that comes from improved memory checking [35]. The security community has developed and deployed several dynamic techniques to detect and mitigate many dangerous behaviors.

Private information leakage has long been a primary security concern. The security community has developed several techniques which observe the difference in dynamic executions to find and remove information leaks [113, 61]. Another approach for tracking the flow of private information is dynamic information flow tracking, or DIFT. DIFT traces how information flows though a program, the most common implementation of DIFT is taint tracking [44, 63, 89, 33]. A taint tracking analysis attempts to determine if some set of outputs is "tainted," or causally affected, by some set of inputs. Unfortunately, taint tracking has often proven to have prohibitive overheads, resulting in a relatively low adoption in deployed systems.

Dynamic analysis systems have also been used to identify malicious applications, or malware [20, 41]. These analyses typically observe the program's API calls, attempting to identify behavior which fits the profile of an attacker.

Optimization Dynamic analysis has even been used to optimize code. Profile-guided optimization uses dynamic observations, or "semi-invariants" to create common-case fast-paths though regions of code. In the instance the semi-invaraints do not hold, the execution

falls back to a slower execution path [26, 81, 104]. These optimizations have proven particularly effective when used within a JIT system [25].

2.2.2 Analysis Optimization

As the use of program analysis is often dictated by the overhead associated with the analysis, and its accuracy. Dynamic analysis systems use a wide variety of techniques to reduce their runtime overheads. In this effort we focus on optimization techniques which combine static and dynamic program analysis to reduce the overhead of that analysis.

2.2.2.1 Sound Static Then Dynamic Analysis

One common approach to reduce the runtime overhead of dynamic analysis is that of a traditional *hybrid analysis*. A traditional hybrid analysis first performs a sound static analysis, and then uses the results of that analysis to prune dynamic checks. An excellent example of this is CCured [87]. CCured uses type inference to remove the vast majority of memory checks, only ultimately dynamically checking a few accesses for which the type inference fails. This allows CCured to have nearly an order of magnitude lower runtime overhead than systems which use strictly dynamic methods. Other examples of this include race detection [42, 94, 29], hybrid type checking systems [58, 83], hybrid taint tracking systems [28], and other memory safety soltuions [84].

2.2.2.2 Dynamic then Unsound Static

Unfortunately, as many static analyses are fundamentally unsolvable, they must operate very conservatively, and approximate many program properties. These approximations often lead to unacceptable imprecision, resulting in limited dynamic check reduction. To get even lower overheads, researchers have attempted to approximate dynamic analyses by first running a fast, but incomplete dynamic analysis, such as Daikon [45], to gather likely invariants, and then feeding those results into a static analysis.

These analyses have been applied to enable reasoning about web-based frameworks [37, 38, 110], enable slicing of large complex programs [53, 80], to identify bugs in programs [97, 90], and to reconstruct program control properties [65]. Unfortunately, these results lack both the soundness of a sound static analysis, and the precision of a precise dynamic analysis.

2.2.2.3 Dynamic then Unsound Static then Unsound Dynamic

A few systems attempt to accelerate dynamic analysis with a three phase analysis, similar to OHA. They learn likely invariants, then use these invariants in an unsound static analysis to produce a faster final dynamic analysis, often for bug-finding purposes [55, 36]. However, these systems do not compensate for the unsoundness introduced by their unsound static analysis, so the final dynamic analysis is unsound. Conversely, OHA solves the unsoundness introduced in the static phase with speculative execution, and a carefully designed predicated static analysis.

Unfortunately, even after years of effort, many dynamic analyses are not employed due to their runtime overhead. For example, most c programs are not checked for memory safety at runtime. Some researchers even argue that for "safe" languages, such as Java, more dynamic enforcement should be used, even at the cost of performance, to guarantee sane application behaviors [79].

In this thesis I focus on improving the overhead associated with dynamic analysis. I propose a new form of hybrid analysis called *optimistic hybrid analysis*, which combines static analysis, dynamic analysis, and speculation in chapter IV.

2.3 Summary

Both deterministic record and replay, and dynamic analysis have proven to be powerful tools in developing complex systems. Unfortunately, due to prohibitive overheads, there are many scenarios today where there tools are not used. By reducing the overhead of these

tools, we can enable many critical dynamic guarantees over applications.

CHAPTER III

Eidetic Systems

Deterministic record and replay systems have long been valuable tools for solving a variety of problems related to debugging, forensics, reliability, and more. One of the great benefits of a record and replay system, is its ability to retroactively analyze any prior state, without the need to predict which states are important before execution. However, in order for this to be generally true, it must be practical for the record and replay system to be capable of both recording and replaying *all* state on a system over a period of years.

Unfortunately, prior record and replay systems do not realize this vision; they present either prohibitively large recording overheads, limiting the amount of information which can be recorded, or excessive replaying overheads, removing the utility of the data once recorded. In this work I argue for and describe a system capable of both recording and recalling all computation on a computer system over a period of years. I call such a system an *eidetic system*.

I describe an eidetic system called Arnold that provides the above properties for personal computers and workstations with reasonable storage requirements and runtime overheads. The key technologies that enable Arnold to provide the properties of an eidetic system efficiently are deterministic record and replay [23], model-based compression, deduplicated file recording, operating system tracking of information flow between processes [66], and retrospective binary analysis of processes [31, 89].

Arnold uses deterministic record and replay to efficiently reproduce past computations. Reproducing past computations enables Arnold to recall any state and to track the lineage of that state within a replaying entity. Arnold uses numerous optimizations to reduce the amount of data that must be recorded. As a result, the log data required for years of operation of a personal computer or workstation can fit on a commodity hard drive.

To avoid the need to replay the entire system to recover any state, Arnold divides the system into units, called *replay groups*, that can be replayed independently. To track information flow between replay groups, Arnold records dependency information for each communication between replay groups, forming a *dependency graph*. In addition to enabling information flow to be tracked across groups, the dependency graph also allows Arnold to treat as a cache the log of data sent between groups. To conserve space, Arnold can discard this data and regenerate it later by replaying the group that produced it, a technique I call *cooperative replay*.

To analyze execution within a replay group, Arnold uses retrospective binary analysis, in which it deterministically re-executes the process within the group and analyzes this re-executed computation. This technique offers several benefits over a traditional on-line analysis. First, it can retroactively run any query over the computation, even queries which were not anticipated at the time of recording. Second, it moves the analysis overhead from the original execution time to the time of the query. I will demonstrate the utility of this technique by applying it towards lineage queries, in which Arnold tracks the relationship between inputs and outputs.

I have run an experiment in which myself and my collaborators have run Arnold continuously on our workstations for several weeks. Our results show that its storage requirements for 4 or more years of operation could be satisfied by adding a $150 4TB hard drive. On almost all benchmarks we ran, Arnold's performance overhead is less than 8%. I also report on several case studies in which I use Arnold to reproduce past state and trace lineage over many applications and workflows.

3.1 Motivation

The vast majority of state produced by a typical computer is generated, consumed, then lost forever. Lost state includes process address spaces, deleted files, interprocess communication, and input received from the network. With lost state comes lost value: users cannot recover detailed information about past computations that would be useful for auditing, forensics, debugging, error tracking, and many other purposes.

Prior approaches try to retain some of this information via a variety of techniques, such as file backup, packet logging, and process checkpointing, but these approaches preserve only the subset of information that someone anticipates may be useful. A more comprehensive approach is needed: one that preserves the values and lineage of *all* state that has ever existed on the system. We call such a system an *eidetic computer system*.

An eidetic computer system can recall any past state that existed on that computer, including all versions of all files, the memory and register state of processes, interprocess communication, and network input. Further, an eidetic computer system can analyze the history each byte of current and past state.

In this work I motivate the useage of eidetic systems by applying Arnold toards a series of lineage queries developed by Michael Chow [32]. Lineage describes *how* state was derived. With such information, the user of an eidetic system can often infer *why* the data was derived. For instance, a colleague might point out to a user that a citation in a paper draft is incorrect. Using an Arnold with these lineage queries, the user could trace back from the binary document through all the steps used to create that document and recreate the browser screen displaying the Web page from which the data was derived. On seeing that Web page, the user would realize that he cited the wrong paper from a conference session. The user could then trace forward from that mistake and reveal all current documents and data that reflect the mistake, as well as any external output (e.g., e-mail) containing mistaken information.

Or consider an example in which someone runs a malicious application on a shared

computer. The malicious program exploits a privilege escalation vulnerability, gives itself privileged access, and installs a backdoor for future access. A lineage query enabled eidetic system could trace forward from the malicious program, trace through the privilege escalation vulnerability, and determine that the malicious software installed a backdoor. The system could then trace any future executions of the vulnerable program and determine if the backdoor was ever used, and exactly what was done by the attacker during the vulnerable window. In these and similar examples, recall and lineage are tightly coupled; they are useful in isolation but more powerful when combined.

3.2 Design goals

The design of Arnold was guided by several goals. First, we wanted to support the widest possible range of queries about user-level state and the history of that state. Arnold reproduces and tracks state of all user-level processes at the level of the instruction set architecture. We wanted to support queries about the history of a computation both within a replay group (Section 3.3.6) and between replay groups (Section 3.3.5). We also wanted to support queries not anticipated at the time of recording, which we accomplish via retrospective binary analysis (Section 3.3.6).

Second, we wanted to minimize the time and space overhead of recording, since we intend for Arnold to continuously record computer usage. We wanted the time overhead of recording to be low enough to support interactive workloads and the space overhead to be small enough to record several years of execution of workstations and personal computers on a commodity hard drive. We reduce the time overhead of recording through deterministic record and replay (Section 3.3.1) and retrospective binary analysis (Section 3.3.6). We reduce space overhead through techniques such as model-based compression (Section 3.3.2), deduplicated file recording (Section 3.3.3), and cooperative replay (Section 3.3.4).

Third, we wanted to reduce the cost of answering queries by not requiring the reexecution of processes unrelated to the state being queried. We accomplish this by dividing

the system into multiple replay groups, each of which can be replayed independently. To preserve lineage between replay groups, we track the dependencies cause by inter-group communication in a dependency graph (Section 3.3.5).

3.3 Design and implementation

3.3.1 Record and replay

Deterministic record and replay enables two important features of Arnold. First, it allows Arnold to efficiently reproduce the complete architectural state (register and address space) of user-level processes. Second, it allows Arnold to defer the work needed to track lineage from the time of execution to the time of querying [31].

To enable reproduction of all architectural state, Arnold records and replays execution at the level of processes. Our modified Linux kernel records all nondeterministic data that enters a process: the order, return values, and memory addresses modified by a system call; the timing and values of received signals; and the results of querying the system time.

Dealing with multiple threads/processes that write-share memory requires special care. Record and replaying individual threads/processes would shrink the scope of replay needed to answer a query, but this would require Arnold to record all nondeterministic reads of shared memory. Instead, Arnold records all threads/processes that share memory as a single *replay group*, then seeks to replay the interleavings of events from the replay group deterministically.

To enable deterministic replay of a replay group, Arnold records all synchronization operations and atomic hardware instructions (such as `atomic_inc`, or `atomic_dec_and_test`). A modified version of `libc` logs the order and memory addresses of synchronization operations between threads, including low-level atomic instructions and high-level synchronization operations such as `pthread_lock`. Such logging inserts an additional two atomic instructions for each event logged (to order the start and end of the

operation). In the absence of data races, this information is sufficient to faithfully replay the recorded execution of a replay group involving multiple threads or processes—each replayed thread will execute the same sequence of instructions and system calls, observe the same values read, and produce the same results as during recording [95].

In the presence of data races, the replayed execution may diverge from the recorded one. We deal with programs with data races by identifying the races and adding additional instrumentation to eliminate them on subsequent runs. Veeraraghavan et al. [106] observed a synergy between deterministic replay and data race detection: if the only reason that a replayed execution may diverge from a recorded execution is the presence of a data race, then the replay system can act as a very efficient data-race detector. Arnold supports the ability to instrument and observe the execution of replayed recordings (Section 3.3.6), and we use this to run a standard vector-clock data race detector [93] when a replay divergence is detected. This is guaranteed to detect at least the first pair of racing instructions (it may also detect subsequent pairs). We then either statically instrument the code to record the outcome of the data race, or dynamically instrument the binary when it runs to cause the racing pair of instructions to trap to the kernel (via an `INT 3` instruction), where we record the order of the racing instructions. Static instrumentation is preferred since it is more efficient, but dynamic instrumentation allows us to support applications for which we do not have source code.

In practice, we have detected few data races that affect replay in the programs we run on our workstations. It has been relatively simple for a small team of users to add the necessary instrumentation to record these instances. Interestingly, many of the races we found were already documented, for example by developers who ran ThreadSanitizer [101] or similar tools. Since races are very infrequent, we suspect that it should usually be possible to search through all possible interleavings of the racing instructions to find an interleaving that is indistinguishable from the recorded execution [9, 92].

When a process executes the `exec` system call, Arnold creates a new replay group

(with a unique 64-bit identifier) consisting solely of that process. Arnold also saves a small checkpoint for the new group, which allows replay to begin from the creation of that process. The checkpoint consists of the arguments and environment variables passed to `exec`, other nondeterministic information used during the system call (e.g., seeds used to randomize address spaces), and a *reference* to the file containing the executable image—the image usually resides in a deduplicated file store described in Section 3.3.3.

Arnold creates new replay groups on `exec` rather than on `fork` because the initial address space at `exec` is more amenable to deduplication than the address space at the time of `fork`. It stores a *split record* that contains the unique identifier of the new replay group in the log of the replay group that performed the `exec`. Infrequently, two replay groups need to be merged (e.g., because they establish a write-shared memory segment). In such instances, Arnold merges the processes from one group into the other and inserts a *merge record* into their logs.

Arnold replays recorded execution on a per-group basis. It creates a new process from the group's checkpoint and deterministically reexecutes the process by supplying values from the group's log in lieu of performing any nondeterministic action. As additional threads and processes are created within the replay group, Arnold also replays those entities. Each process executes until it exits or the execution reaches a split record. Arnold can replay multiple groups concurrently—this allows it to parallelize lineage queries that span groups.

3.3.2 Reducing storage utilization

Arnold uses several optimizations to reduce the size of its replay logs. The first optimization is model-based compression. The order and results of many of the system calls and synchronization operations that Arnold logs are highly predictable. For instance, many system calls usually return zero (success); the `write` system call usually returns the number of bytes in the input buffer; and `pthread_cond_lock` usually returns a value specifying

that the lock has been obtained. Arnold constructs a model for predictable operations and records only instances in which the returned data differs from the model. Thus, the log size used for each type of operation is proportional to the number of deviations, which can be much less than the number of executed operations.

Some operations such as `poll` exhibit considerable locality in the data they return (e.g., the set of ready file descriptors is often the same from call to call within a short window). For these operations, Arnold caches the most recent 8 values returned on both record and replay and replaces the actual values in the log with a small cache index (when the value hits in the cache) in order to save space. Arnold also uses model-based compression to reduce the amount of ordering information in the log. It predicts that there are no ordering constraints and no signals delivered between two successive logged operations, and records only when the execution deviates from the model.

After applying model-based compression, we determined that the most significant source of log usage on our systems was messages sent from the X server to applications. A small fraction of this data comes from user events (button presses, mouse movements, etc.). Most data consisted of responses to application requests. Since such responses included nondeterministic data such as identifiers and window properties, the responses needed to be recorded to faithfully replay each application.

We observed, however, that with the exception of actual user input, the behavior of the X server is mostly deterministic. Arnold avoids logging most data from the X server by using the X server to help regenerate data during replay. We insert a proxy between applications and the X server that records only a small subset of the data sent from the X server, such as identifiers and window properties generated nondeterministically by the X server. During replay, the application again connects to an X server via the proxy. The proxy translates the nondeterministic values, and the replay process generates GUI state using the live X server, but on a separate display. The proxy also inserts the recorded user events at the appropriate point in the stream. In combination with the proxy translation, the

X server produces the same sequence of responses during the replayed execution as during recording. With deterministic X recording, Arnold can make the display of X windows visible during replay. As we will describe, this is useful for showing users application displays that correspond to the results of lineage queries and for allowing users to specify queries by clicking on recreations of windows displaying data they observed in the past. By recording only nondeterministic response values and user input, the proxy substantially reduces the amount of information in the logs of GUI applications.

After applying the above optimizations, we noticed that time queries constituted a substantial portion of the remaining log size. To reduce the amount of nondeterminism that needs to be logged, Arnold uses a *semi-deterministic clock*. The value returned by a semi-deterministic clock is guaranteed to be less than the real-time clock for the system, and within a specified delta. The default delta is 10ms; it may be overridden by applications that need more accuracy. A replay group's semi-deterministic clock is incremented deterministically based on the number and type of logged operations (which is the same during both recording and replay). When the time is queried, Arnold reads the actual real-time clock. If the semi-deterministic clock is greater than or more than delta behind the real-time clock, Arnold returns the real-time clock value, sets the semi-deterministic clock equal to the real-time clock, and records the new value in the log. Thus, the amount of time query data in the log is proportional to the number of such resets rather than the total number of time queries; if Arnold usually predicts the clock value correctly, the amount of logged time data can be quite small.

Arnold ensures that observed semi-deterministic clock values are externally consistent. It is for this reason that the semi-deterministic clock must always be less than the real-time clock. If a recorded process sends a message to a non-recorded process, the receiver will always observe that the message arrived after it was sent. Further, if a recorded process receives data from or sends data to an entity outside the replay group, the group's semi-deterministic clock is set to match the real-time clock. Thus, the observed clock values are

causally consistent both across all processes on the computer system and with respect to external entities.

Finally, Arnold compresses all log data with `gzip`. This is very effective in compressing some input, such as text. It also helps to compress applications that perform repetitive operations with similar results.

3.3.3 Copy-on-RAW file cache

Arnold records the file data read by a process so that data can be redelivered to the process during replay. Recording this data can take a substantial amount of log space, so Arnold optimizes how the read file data is stored by deduplicating it. This works particularly well when a file is read multiple times before being modified.

To deduplicate the read file data, Arnold saves a version of a file only on the first read after the file is written. Subsequent reads log only a reference to the saved version, along with the read offset and return code. We refer to this as *copy-on-RAW (read-after-write) recording*.

If another process opens the file for writing while a reading process is running, the reading process reverts back to recording the read values instead of the reference to the stored version (several optimizations are possible here, such as recording the file version on each `read` instead of `open`, or reexecuting reads and writes to files shared among processes in the same replay group.)

Arnold also uses the copy-on-RAW store for file `mmap` operations by mapping the stored file version into the process space on replay. If the mapped region is writable, Arnold creates a private temporary copy of the file version on replay; this allows the replayed process to change the file contents without affecting other replayed processes that reference the same file version.

Note that, with this design, the current version of all files is stored in the default file system (ext4 on our Ubuntu workstations). We chose this operation for efficiency; record-

ing processes (the common usage case) go through the well-optimized file system and receive the best performance. Copy-on-RAW population of the file store can proceed asynchronously and not slow down the recording process too much unless large amounts of data being read exert memory pressure. The cost of this implementation is some double-buffering of current file data, which we could reduce in the future.

We were initially surprised because the size of Arnold's file store grew more slowly than expected on our workstations. On investigation, we realized this was due to an important difference between Arnold's file store and a versioning file system: Arnold's file store does not have to store data that is written but never read. Since Arnold is an eidetic system, it can, of course, recreate this file data; however, it does so by replaying the process(es) that produced the data rather than by retrieving the data from the file system. In contrast, a versioning file system needs to store all file versions even if they are overwritten or deleted without being read.

Since Arnold can reproduce any current or past file version via replay, it is the logs of nondeterminism that are Arnold's truly persistent store [43]. We can thus treat Arnold's copy-on-RAW file store as a *cache*. The copy-on-RAW file store (and, in fact, all file system data) is simply a performance optimization that contains checkpoints of data that could be produced by replay. This reasoning led us to develop *cooperative replay*.

3.3.4 Cooperative replay

We normally think of replay groups as independent entities: we log their nondeterministic inputs during recording and reinsert these inputs during replay. Cooperative replay provides another option, which is to use one replay group to regenerate the data read by another. Cooperative replay allows us to treat the log of all interprocess communication (files, pipes, etc.) as a cache, whose records can be evicted when the cache is full and recovered when needed during replay.

Arnold uses cooperative replay to regenerate data read from files. During replay, if the requested data exists in the file system (because it is the current version of the file) or in the copy-on-RAW file cache, Arnold reads the data from one of those locations. If not, Arnold regenerates the data by replaying the replay group(s) that produced the data. Arnold stores information about the source of all file data in each read record—this includes the identifier of the replay group(s) and the system call(s) executed by the group(s) that produced the file data (Section 3.3.5). To regenerate the data, Arnold suspends the replay group requesting the data, replays the producing replay group(s), repopulates any data evicted from the file cache, and finally resumes the requesting replay group.

Cooperative replay may recurse in a depth-first manner. When replaying replay group A, Arnold may need to replay another replay group B to regenerate file data read by replay group A, and this may trigger the replay of a third replay group C, and so forth. The recursion will stop when a group can be replayed without depending on any other replay group.

As data flows forward, it creates a directed acyclic graph. While no cycles exist between nodes in the graph, Arnold may encounter a scenario where two replay groups depend on outputs of each other. In this scenario Arnold will alternate replaying each group until all dependencies are met.

3.3.5 Dependency graph

To support cooperative replay and track lineage across replay groups, Arnold maintains a logical graph of the data-flow dependencies between groups, which we refer to as the *dependency graph*. Nodes in the graph are <replay group id, system call id> tuples, where the second part of the tuple uniquely identifies a particular system call executed by a process in the replay group. Each edge in the graph is a bidirectional link between the system call that produced data and the set of one or more system calls that consumed that data. Thus, Arnold can determine the lineage of data across replay groups by tracing backward in

time through the dependency graph, and it can determine what downstream values were influenced by particular data by tracing the lineage forward.

We first describe the operation of the dependency graph for file data. When a recorded process writes to a file, Arnold records which bytes were modified, along with the <replay group id, system call id> in a per-file B-tree indexed by the file offset. The root of each per-file B-tree is in turn indexed in a B-tree of all files; we refer to this collection of B-trees as the *filemap*. Arnold allocates a separate region on disk for the filemap; it reads pages on demand into a kernel cache in physical memory and evicts pages using an LRU algorithm. Pages are flushed asynchronously using the journal mechanisms of the underlying file system (ext4 in our current implementation). Thus, the filemap contains the lineage information for all current file data in the file system.

When a recorded process reads from a file, Arnold searches through the filemap to find which system call(s) wrote the bytes being read. It copies the tuples out of the filemap into the replay log of the reading process. Thus, the log contains sufficient data to answer backward lineage queries (how was the data read by this process produced?). In order to answer forward lineage queries, Arnold generates an index over the reverse linkages and stores it in a `sqlite` database. A daemon process asynchronously generates the index by incrementally scanning recent replay logs (replay is unnecessary because the data needed to generate the index is in the logs).

Arnold uses a similar process to record the lineage of other forms of IPC. For pipes and sockets, it keeps metadata for bytes written but not yet consumed in the kernel. For most pipes and sockets, there is a single writing process and a single reading process, and bytes are read in the order they are written. In this common case, Arnold reduces log size by only logging the identifier of the writing replay group. On a query, Arnold identifies the system call(s) that generated data by scanning the log of the writing record group. If there is more than one reader or writer, Arnold tracks the reads and writes on the pipe or socket in the same manner as for file system data.

Arnold also tracks lineage of the data passed from the parent process to the child process during `exec`. This includes arguments, environment variables, and some miscellaneous data used during the `exec` system call.

Arnold does not record the lineage of data passed among processes via shared memory. Instead, Arnold tracks this lineage at query time by instrumenting the memory read and write instructions as described in the next section.

3.3.6 Intra-process queries

Arnold uses Pin [76] binary instrumentation to analyze replayed executions and analyze data within a replay group. We chose Pin because it is a flexible and well-documented tool; however, Pin can be slow, partially because it dynamically, rather than statically, inserts instrumentation into running binaries. Arnold avoids overhead during recording by only using Pin and analyzing intra-process history during replay.

While analysis tools such as Pin are typically invisible to the program they instrument, they are not transparent to the operating system: such tools insert new system calls, allocate additional memory, catch signals, etc. Without special care, these extra actions to support analysis will cause the replayed execution to diverge from the recorded execution. Arnold uses techniques from X-ray [11] to compensate for the divergences caused by analysis; for instance, it prevents Pin from allocating memory that will conflict with the replayed execution and it identifies system calls generated by Pin and executes them live rather than trying to supply nondeterministic values from the group's log.

Arnold analyzes computation within a group by restoring the group's checkpoint and replaying the processes within the group with Pin dynamic analysis enabled.

3.3.7 State queries

Arnold can recover past file versions, transient process state, inputs, and output. If a specified file version does not already exist in its cache, Arnold uses cooperative replay to

regenerate the contents of the file. Arnold reexecutes the specified replay group to regenerate both transient process state and output. Inputs (with the exception of file data) are logged, so no reexecution is needed to retrieve them.

Arnold also provides an interactive state query to allow users to inspect GUI output. During replay, X server output is displayed on the current screen, allowing the user to observe the display as it was manipulated during recording. Via a separate console, the user can fast forward to a particular output (i.e., the display is updated at replay speed without the original think time, I/O delays, etc.) or pause the replay at a particular point in the execution. By delaying a specified amount after each X output, Arnold can also display a slow-motion execution.

3.4 Privacy

One concern with eidetic systems is the potential exposure of private data. Arnold's log can be used to recreate any state on the system and thus must be protected to the same or greater degree that one would protect other private memory and file system state.

A related concern is preserving a user's ability to exclude data from recording. For example, Arnold as described here would preserve all state from a user's "private" browsing session. However, Arnold could provide the ability to remove sensitive portions of the process graph by sanitizing and replacing portions of the log. For instance, Arnold could remove sensitive information contained in a process's address space by replacing that process's execution with a simple stub program that produces the same output.

A user who wants to remove sensitive information may actually benefit from Arnold's ability to track lineage. The user could identify the sensitive information or portions of execution, then ask Arnold to identify points in the process graph that depend upon this information. The user could then replace those parts of the graph with stubs that jump over the sensitive portions. Of course, once data is removed from Arnold, all lineage information and ability to query the execution are also lost.

User	Days	Groups per day	Storage utilization (MB) per day			
			RAW file cache	Logs	Filemap	Total
A	25	995	475	267	36	779
B	24	475	1095	936	339	2064
C	21	26122	869	350	690	1910
D	16	3339	1675	82	838	2594

Table 3.1: Storage utilization during multi-week trial

3.5 Evaluation

Our evaluation answers the following questions:

- What is Arnold's performance overhead?
- What are Arnold's storage requirements?
- Does Arnold enable useful queries?

3.5.1 Storage overhead

We first consider the storage overhead of running an eidetic system. To measure this overhead, the authors of the paper continuously recorded activity on their workstations for several weeks. The recorded activity included all user-level processes started from a terminal or launched from the GUI. It did not include several system-level processes, such as the X server (which is not recorded per the design in Section 3.3.2), processes that directly manipulate the replay data (e.g., indexing tools), and the sshd server (since we sometimes needed to log in without replay enabled for testing and maintenance). With the above exceptions, the vast majority of user activity was recorded. No files were evicted from the copy-on-RAW file cache since storage utilization was reasonable.

Table 3.1 summarizes the storage cost of our eidetic system. The third column shows the average number of replay groups created per day; note that there may be many threads and processes within a single group. The number of groups created varies widely depending on workload; some users have a few long-running applications; others have workflows

(e.g., compilation) that create many short-lived groups. The next columns show average storage utilization per day, broken down to show the individual storage utilization of the copy-on-RAW file cache, the replay log storage, and the filemap. Process checkpoints are included in the total but not shown separately, since they account for less than 2MB per day of storage. Executables and shared libraries referenced by checkpoints are included in the copy-on-RAW file cache.

While we cannot make comprehensive claims about storage utilization without a wider user study, this preliminary data is very encouraging. All users require less than 2.6GB of storage per day; a 4TB drive would suffice for 4–14 years.

	Storage utilization (MB)							
	Firefox	evince	gedit	gpaint	spreadsheet	presentation	latex	make
Baseline	4517.63	194.51	764.34	95.29	4362.49	455.10	20.05	86.69
Model-based compression	308.93	7.07	12.50	36.12	41.16	31.07	7.19	81.52
Copy-on-RAW file cache	283.37	2.63	8.41	34.77	22.63	23.83	0.25	6.13
X compression	173.74	2.61	8.29	1.08	22.55	15.94	0.25	6.13
Semi-deterministic time	127.72	2.11	6.51	0.94	19.23	15.39	0.29	6.12
Gzip	24.87	0.11	0.50	0.08	3.33	12.71	0.05	1.15
Compression ratio	182:1	1752:1	1530:1	1217:1	1311:1	36:1	393:1	75:1

Table 3.2: Reduction in storage utilization via incremental application of optimizations

3.5.2 Benefits of compression

Next, we quantify the benefits of Arnold's storage optimizations. Table 3.2 shows results for several workloads. The `Firefox` workload measures a half-hour browsing session consisting of 15 minutes of active browsing across 8 complex Web sites (e.g., Facebook and stack overflow), followed by 15 minutes of idle usage with Gmail windows open (triggering periodic JavaScript execution). The `evince`, `gedit`, `gpaint`, `LibreOffice` spreadsheet, and `LibreOffice` presentation workloads use those applications intensely for 3 minutes. The `latex` workload builds a prior OSDI paper, and the `make` workload builds the libelf-0.8.9 library. In these and subsequent experiments, we ensure repeatable results by automating all GUI workloads with the Linux Desktop Testing Project library [1], which captures and emulates GUI events.

The first row in the table shows storage usage when all nondeterministic inputs are logged without any compression. The subsequent rows show the effect of cumulatively applying model-based compression, copy-on-RAW file caching, the deterministic X proxy, semi-deterministic time, and gzip compression. The final row shows the compression ratio compared to baseline due to all of Arnold's optimizations.

Arnold averages a 411:1 compression ratio compared to the baseline. For comparison, simply applying gzip to the baseline averages only a 6:1 compression ratio. At 36:1, `LibreOffice` presentation sees the lowest compression ratio; this is due to the recording of temporary files written and read by the application. In the future, Arnold could omit such data since the files are never read by other replay groups and Arnold could regenerate the file contents during replay.

3.5.3 Performance overhead

Next, we measure Arnold's performance overhead during recording (i.e., the overhead that would normally be experienced by a user). All tests were run on a computer running Ubuntu 12.04LTS with an 8 core Xeon E5620 2.4GHz processor, 6GB memory, and a 1TB 7200RPM hard drive. We measured several terminal and GUI applications, and one server workload (apache):

- **kernel copy** – `cp -a` of the 3.5.0 Linux source.
- **cvs checkout** – check out Arnold's kernel source (589MB, 52730 files) from a repository accessible via a local, 1Gb local network connection.
- **make** – compile the libelf-0.8.9 library.
- **latex** – build a prior OSDI paper with `latex/bibtex`.
- **apache** – run the apache benchmark on an apache 2.2.22 server, configured with mpm_prefork with 256 workers, and a client connected via a 1Gb local network connection (5000 requests for a 34KB page with 50 concurrent requests at a time.)

- **gedit** – open a 15,000 line C file and find/replace on a commonly occurring string.
- **facebook** – load the White House's public Facebook page in `Firefox` version 23.0 (the completion time is measured by the onLoad event.)
- **spreadsheet** – Open a 704KB csv spreadsheet in `LibreOffice` 3.5 and convert it to an xml document.

Figure 3.1 shows the performance of Arnold on a variety of desktop workloads, normalized to the performance of an uninstrumented system. The middle bar for each workload shows the performance during recording; this is the overhead the user will experience during normal operation. The third bar shows the performance during recording when Arnold uses a second hard drive for logging, which minimizes interference with normal file system writes.

Arnold's overall performance impact is small: overhead is under 12% with a single disk for all but two workloads. The cvs checkout has approximately 100% overhead with a single disk because it saves all checked-out data twice: once as nondeterministic network input and again when cvs writes the data to the file system. Adding a second logging disk reduces the overhead for cvs to 15%.

The higher overhead seen by kernel copy is caused by saving filemap entries. This workload is disk-bound and creates many small files. For each file created, Arnold must create a B-tree to record lineage data—this is effectively a worst-case for saving filemap entries. A separate logging disk reduces the overhead to 1.7%.

The Facebook tests contained some outliers due to external network and servers. We eliminated gross outliers (500% or more above the median) from our measurements (both for baseline and for Arnold); doing so did not help Arnold disproportionately.

We also evaluate the scalability of Arnold on several Splash 2 benchmarks, shown in Figure 3.2. While scalability was not a focus of our work, Arnold has low overhead for all these benchmarks up to 8 threads. We attribute this to two factors. First, Arnold requires

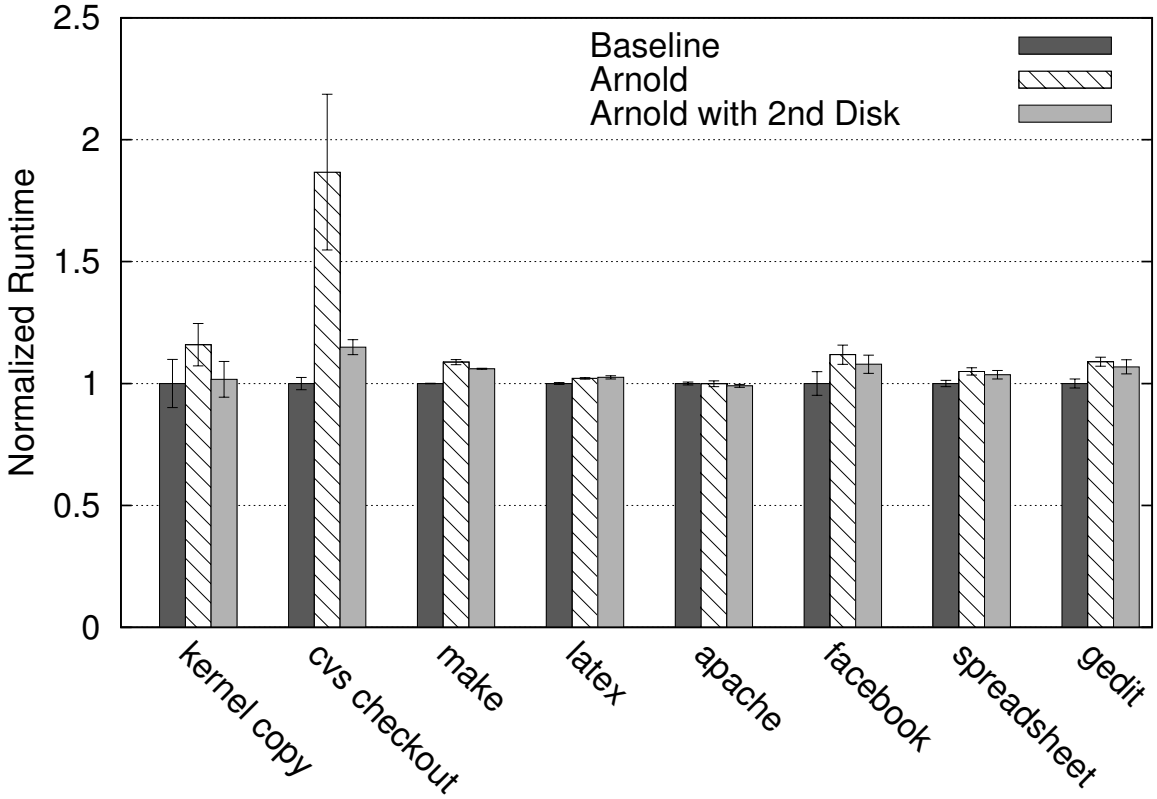


Figure 3.1: Arnold performance overhead normalized to unmodified Linux. Error bars are 95% confidence intervals.

programs to be race-free, so it only has to check and log inter-thread synchronization operations rather than all shared-memory operations. Second, Arnold's model-based compression reduces the instrumentation overhead per synchronization operation to only two atomic instructions in the common case.

In summary, Arnold adds modest overheads of less than 12% with a single disk on all but 2 workloads over a wide range of desktop and interactive applications. Adding a second hard drive reduces the overhead to under 8% on all but one workload. In practice, even on single hard drive configurations, we noticed virtually no difference between our recorded applications and non-recorded applications. In fact, we needed to add a utility to our shell interface simply to determine whether recording was currently enabled or disabled.

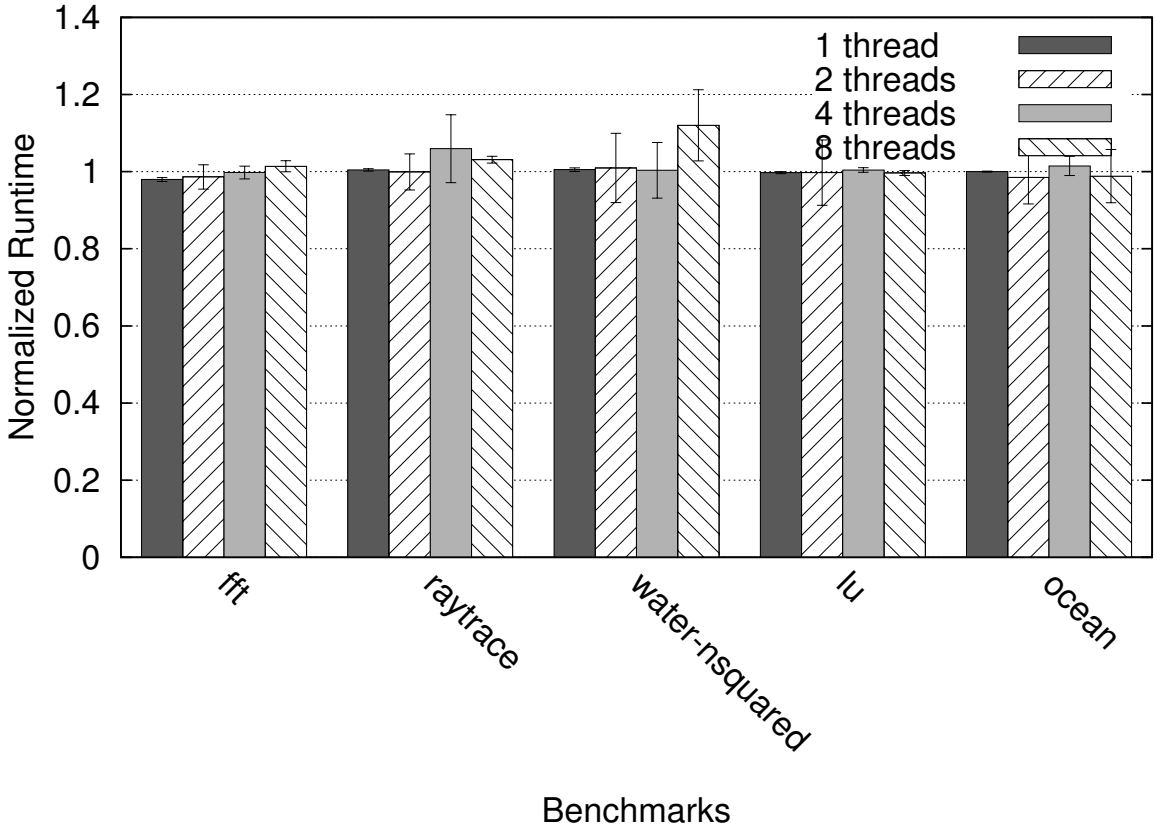


Figure 3.2: Arnold's scaling, normalized to unmodified Linux, on Splash2 benchmarks. Error bars are 95% confidence intervals.

3.5.4 Case studies

Finally, we look at a series of case studies of lineage queries enabled by Arnold.

To show the effectiveness of Arnold on real queries, we evaluate a series of practical lineage queries. These queries use Arnold's PIN support to run an intra-process taint tracking system developed by Chow [32], and the inter-process lineage facilities enabled by Arnold's filemap. Queries are run in one of two directions, bacwards (how was this data derived?) and forward (what did this state influence?).

There are many possible ways of defining lineage: one can say that an input influences an output only if the output is derived from the input via a series of copies, or one can consider other forms of data flow, or control flow, etc. Arnold evaluates several common linkage functions (and applications may define their own):

- **Copy.** An input of an instruction influences an output only if the instruction copies the value of the input to the output location (e.g., via a move instruction).
- **Data flow.** An input of an instruction influences an output if the instruction uses the input to calculate the value of the output (e.g., via an add instruction).
- **Index.** An input influences an output if the input is used to calculate the output or if the input is used as an index to load a value used to calculate the output (e.g., via an array or lookup table index).
- **Control flow.** This includes, in addition to index and data flow influence, the influence propagated via control flow as tracked using the algorithms developed by ConfAid [12].

3.5.4.1 Backward Query

Our first case study is a typical backward linage query. In this scenario, a colleague points out to the user that he has cited the wrong paper in a conference submission. The

user runs a backward query to determine how the incorrect citation was produced and what the correct citation should be.

We executed this query by opening a binary document with `xdvi`, scrolling to the bibliography, and clicking on a screen location to specify the incorrect citation as the starting state for the query. We did not specify a linkage, so Arnold ran the query with multiple linkage functions (the various linkage functions are analyzed in parallel via concurrent replay). For each step, Arnold chooses the most restrictive linkage function that produces some result (shown below in parentheses).

The query returned the following results:

- The specified output of `xdvi` came from the input file "paper.dvi" (index linkage).
- The incorrect citation in "paper.dvi" was generated by `latex` with data coming from the input file "paper.bbl" (data linkage).
- The data in "paper.bbl" was generated by `bibtex` with data from "full.bib" (copy linkage).
- The data in "full.bib" was generated by `vim` with data from the terminal device (copy linkage).
- A human linkage (comparrison between data output-to and received-from users) reveals a fuzzy substring match between data coming from the terminal and `Firefox` output.
- The output displayed by `Firefox` came from a conference Web site (data linkage).
- The query also reports four false positives: a `latex` format file, a font file, `libc.so` and `libXt.so`.

Using Arnold, the user fast-forwards a `Firefox` replay to the point indicated by the query result. On viewing the recreated GUI, he realizes that the paper that he meant to cite was the *next* paper in the session after the incorrect citation.

Case Study	Record Time	Replay Time	Replay & Pin	Query Time
Case Study 1: Backward Query	96.1s	2.2s	70.0s	209.5s
Case Study 2: Forward Query	30.3s	1.6s	80.4s	110.7s
Case Study 3: Forward Heartbleed Query	114.1s	0.1s	6.9s	19.7s
Case Study 3: Backward Heartbleed Query	230.3s	0.4s	139.5s	235.1s

Table 3.3: Summary of case studies

As shown in the first row in Table 3.3, the query takes 209 seconds to execute, whereas the cumulative execution time of the recorded processes was only 96 seconds. Replay of the processes with zero instrumentation takes only 2 seconds because all user think time and most I/O delays are eliminated or replaced with a sequential disk read from the log. Simply attaching Pin to the replayed processes and inserting a very minimal instrumentation tool (which counts the number of system calls executed) increases the replay time to 70 seconds—this is the lower bound for any Pin tool on this workload.

The time it takes Arnold to perform the queries in table 3.3 is dominated by the Pin tool instrumentation and analysis, and not the actual replay system. Consequently, Arnold's query times are dictated by the number of instructions analyzed and the amount of taint information in the address space.

This query demonstrates that Arnold can successfully follow a long chain of applications to trace the lineage of data back to external inputs. The chain contains both binary and text data, as well as several types of linkages (intra-process, file, and human). Note that simply searching over inputs and outputs cannot reveal this whole chain (e.g., incoming Web data is encrypted, text input to `vim` includes backspaces and various keyboard macros, etc.) Lineage queries, however, can uncover linkages to such inputs because they directly observe the transformation of bytes in the process address space.

3.5.4.2 Forward Query

The second case study is a typical forward query. Our user now wishes to determine what other data and output has been affected by the faulty citation.

We executed this query by specifying the starting state as the incorrect citation in "full.bib." We also specified the index linkage function.

The forward query returns a list of external outputs and current files that the incorrect citation affected. Some key points of the result are:

- All subsequent versions of "full.bib" contain the incorrect citation. This is a shared bibliography file that is used to generate citations in several other papers on the user's computer. The forward query tracks the incorrect citation through the entire paper compilation process (e.g., though `bibtex`, `latex`, `dvips`, and `ps2pdf`).

- The query flags all files produced during the paper compilation process that include the specified citation (e.g., "paper.bbl", "paper.dvi", "paper.ps", and "paper.pdf.")

- The query does not return false positives. The user also has several papers that use the bibliography file "full.bib", but those papers do not cite the incorrect citation. Even though "full.bib" is read when those papers are compiled, Arnold correctly reports that no output file is affected by the incorrect citation.

- The query shows that the user had copied and pasted the incorrect citation from "full.bib" to another file, "paper.bib", using `vim`. The query also returns subsequent compilations and output files of those compilations that reference the incorrect citation in "paper.bib".

- The query detects that the user ran a python script to produce a file with more succinct version of the citations, "small.bib", from "full.bib". It detects the incorrect citation in "small.bib" and in paper compilations that reference the incorrect citation from that file.

- The query detects that the user e-mailed a paper with the incorrect citation. This shows up as a network output of `sendmail`.

- The query returned no false positives.

The second row of Table 3.3 shows that the forward query required 111 seconds to execute, whereas simply replaying all processes with the simple Pin tool require 80 seconds. Thus, the relative overhead of the forward query instrumentation, is (as expected) much less than that for a backward query.

3.5.4.3 Heartbleed

Our third case study is motivated by the 2014 Heartbleed attack. One reason this attack caused such concern is that service providers were unable to determine what (if any) data was leaked. We show how Arnold is able to help an administrator determine whether sensitive data was leaked from a low-volume Web server, which hosts and stores a key-value database.

First, the administrator runs a forward query to see if the server's private key was leaked. This query requires a custom definition of output, so she creates a Pin tool. Heartbleed exploited a missing bounds check, so the Pin tool simply emulates the missing bounds check when the target instruction is reached and flags as output any data in excess of what the bounds check would have rejected. Her forward query specifies a starting state of the private key file, an output definition of only those bytes returned by the Pin tool, and the index linkage function.

We emulated this scenario by recording 100 GET and POST requests to Nginx 1.4.7 with OpenSSL version 1.0.1f (run times scale roughly linearly with the number of requests). This query took 20 seconds to perform and returned no outputs, showing that the private key was not leaked). We confirmed the correctness of this result manually.

Next, the administrator asks: *was any data leaked, and if so what data?* We constructed a backward query to answer that question. We used the custom Pin tool to define as starting state any data incorrectly sent due to the Heartbleed exploit and specified the index linkage. The backward query determined that:

- The Web server, `Nginx`, serviced a heartbeat request that leaked process memory.

- The leaked bytes came from a UNIX socket written by `FastCGI`, which is responsible for dynamic Web content.
- `FastCGI` received these bytes from a pipe written by a python script that it spawned.
- The script read the bytes from a database file.
- The bytes read from the database file came from POST requests that inserted those key-value pairs. This is determined by following the bytes back through a python script, `FastCGI`, and `Nginx`.

In summary, Arnold reveals both the leaked content *and* the origin of that data. Further queries could reveal the specific users who had their content leaked (e.g., by using a Pin tool to extract the userid from the connections that wrote the leaked data). The total query time was 235 seconds, roughly double the cost of replay and Pin instrumentation alone.

This case study shows the value of not limiting a priori the types of lineage that an eidetic system can track. For example, prior tools for intrusion recovery focus on inter-process lineage but cannot track intra-process lineage [66, 50, 64]. Upon learning of the vulnerability, the user can write new tools that detect data flows she had not anticipated at the time the system was being recorded, then use these tools on executions that were recorded before the tools existed.

3.6 Conclusions and future work

Arnold is a prototype of an eidetic system, targeted at personal computers and workstations. Arnold can recall any past user-level state, and it can trace the history of any byte in a current or prior state. This paper shows that the overheads of providing such functionality are reasonable: our results shows that adding a commodity hard drive can satisfy 4 or more years of storage needs with most runtime overheads under 8%. We have made the source code for Arnold available at https://github.com/endplay/omniplay.

Our case studies show the power of an eidetic system by recovering past state and tracing the lineage of data through a wide variety of applications and user interactions. The precision of combining operating system tracing of inter-process information flow with retrospective analysis of intra-process information flow yields accurate and informative query results.

CHAPTER IV

Optimistic Hybrid Analysis

4.1 Introduction

Dynamic analysis tools, such as those that detect data-races [98, 48], verify memory safety [87, 88], and identify information flow [44, 63], have become a vital part of testing and debugging complex software systems. However, their substantial runtime overhead (often an order of magnitude or more) currently limits their effectiveness. This runtime overhead requires that substantial compute resources be used to support such analysis, and it hampers testing and debugging by requiring developers to wait longer for analysis results.

These costs are amplified at scale. Many uses of dynamic analysis are most effective when analyzing large and diverse sets of executions. For instance, nightly regression tests should run always-on analyses, such as data-race detection and memory safety checks, over large test suites. Debugging tools, such as slicing, have been shown to be more informative when combining multiple executions, e.g. when contrasting failing and successful executions [52, 7]. Forensic analyses often analyze weeks, months, or even years of prior computation [63]. Any substantial reduction in dynamic analysis time would make these use cases cheaper to run and quicker to finish, so performance has been a major research focus in this area.

Hybrid analysis is a well-known method for speeding up dynamic analysis tools. This method statically analyzes the program source code to prove properties about its

execution. It uses these properties to prune some runtime checks during dynamic analysis [69, 87, 28, 29]. Conventionally, hybrid analysis requires sound [1] (no false negatives) static analysis, so as to guarantee that any removed checks do not compromise the accuracy of the subsequent dynamic analysis. However, soundness comes at a cost: a lack of precision (i.e., false positives) that substantially reduces the number of checks that can be removed and limits the performance improvement for dynamic analysis tools such as race detectors and slicers.

The key insight in this work is that **hybrid analysis can benefit from carefully adding unsoundness to the static analysis, *and* preserve the soundness of the final dynamic analysis by executing the final dynamic analysis speculatively.** Allowing the static analysis to be unsound can improve its precision and scalability (Figure 4.1), allowing it to dramatically speed up dynamic analyses such as race detection (even after accounting for the extra cost of detecting and recovering from errors introduced by unsound static analysis).

Optimistic hybrid analysis is a hybrid analysis based on this insight. It combines unsound static analysis and speculative execution to create a dynamic analysis that is as precise and sound as traditional hybrid analysis, but is much faster. Optimistic hybrid analysis consists of three phases. First, it profiles a set of executions to derive optimistic assumptions about program behavior; we call these assumptions *likely invariants*. Second, it performs a static analysis that assumes these likely invariants hold true, we call this *predicated static analysis*. The assumptions enable a much more precise analysis, but require the runtime system to compensate when they are violated. Finally, it speculatively runs the target dynamic analysis, verifying that all likely invariants hold during the execution being analyzed. If so, both predicated static analysis and the dynamic analysis are sound. In the rare case where verification fails, optimistic hybrid analysis rolls back and re-executes the program with a traditional hybrid analysis.

[1] Following convention, we classify an analysis as sound even if it is only "soundy" [72]. For example, most "sound" static analysis tools ignore some difficult-to-model language features.

We demonstrate the effectiveness of optimistic hybrid analysis by applying it to two popular analyses on two different programming languages: OptFT, an optimistic hybrid data-race detection tool built on top of a state-of-the-art dynamic race detector (FastTrack) [48] for Java, and OptSlice, a optimistic hybrid backward slicer, built on top of the Giri dynamic slicer [97] for C. Our results show that OptFT provides speedups of 2.9x compared to FastTrack, and 1.6x compared to a hybrid-analysis-optimized version of FastTrack. Further, OptSlice analyzes complex programs for which Giri cannot run without exhausting computational resources, and it provides speedups of 8.3x over a hybrid-analysis-optimized version of Giri. We then show how predicated static analysis can improve foundational static analyses, such as points-to analysis, indicating that optimistic hybrid analysis techniques will benefit many more dynamic analyses.

The primary contributions of this chapter are as follows:

- We present optimistic hybrid analysis, a method of dramatically reducing runtimes of dynamic analysis without sacrificing soundness by first optimizing with a predicated static analysis and recovering from any potential unsoundness through speculative execution.
- We identify properties fundamental to selecting effective likely invariants, and we identify several effective likely invariants: unused call contexts, callee sets, unreachable code, guarding locks, singleton threads, and no custom synchronizations.
- We demonstrate the power of optimistic hybrid analysis by applying the technique to data-race detection and slicing analyses. We show optimistic hybrid analysis dramatically accelerates these analyses, without changing the results of the analysis. To the best of our knowledge, OptFT is currently the fastest dynamic happens-before data-race detector for Java that is sound.

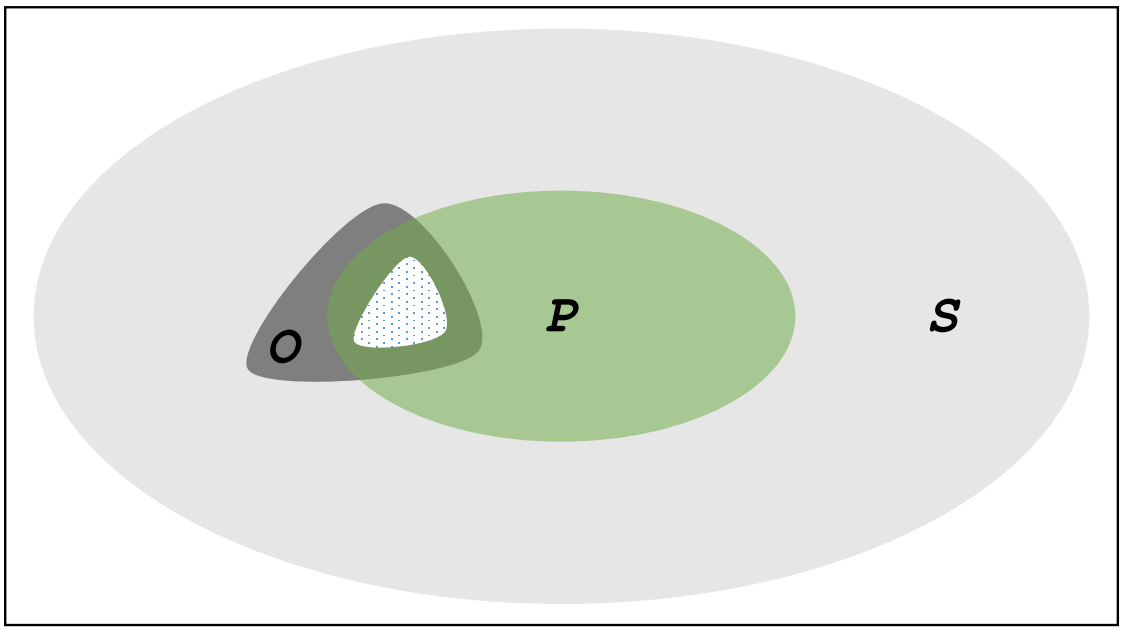


Figure 4.1: A sound static analysis not only considers all valid program states P, but due to its sound over-approximation, it also considers a much larger S. Using likely invariants, a predicated static analysis considers a much smaller set of program states O that are commonly reached (dotted space in P).

4.2 Design

Optimistic hybrid analysis reduces the overhead of dynamic analyses by combining a new form of unsound analysis, known as predicated static analysis, with speculative execution. The use of speculative execution allows optimistic hybrid analysis to provide correct results, even when entering states not considered by predicated static analysis. A predicated static analysis assumes dynamically-gathered likely invariants hold true to reduce the state space it must explore, creating a fundamentally more precise static analysis.

Figure 4.1 shows how the assumptions in a predicated static analysis can dramatically reduce the state space considered. A sound static analysis must make many overly-conservative approximations that lead it to consider not just all possible executions of a program (P), but also many impossible executions (S).

Rather than paying the cost of this over-approximation, a hybrid analysis can instead construct a static analysis based only on the set of executions likely to actually be analyzed

dynamically. Speculative assumptions make the state space (O) much smaller than not only S but also P, demonstrating that by using a predicated static analysis, optimistic hybrid analysis has the potential to optimize the common-case analysis more than even a perfect sound static analysis (whose results are bounded by P). The set of states in P not in O represent the set of states in which predicated static analysis is unsound. Optimistic hybrid analysis uses speculation and runtime support to handle when these states are encountered. As long as the set of states commonly experienced at runtime (denoted by the dotted area in P) resides in O, optimistic hybrid analysis rarely mis-speculates, resulting in an average runtime much faster than that of a traditional hybrid analysis.

We apply these principles using our three-phase analysis. First, we profile a set of executions of the target program and generate optimistic assumptions from these executions that might reduce the state space the static analysis needs to explore. As these dynamically gathered assumptions are not guaranteed to be true for all executions, we call them *likely invariants* of the executions.

Second, we use these likely invariants to perform a predicated static analysis on the program source. Leveraging the likely invariants allows this static analysis to be far more precise and scalable than traditional whole-program analysis, ultimately allowing it to better optimize dynamic analyses.

Finally, we construct and run the final dynamic analysis optimistically. Because predicated static analysis is not sound, we insert extra checks in this optimistic dynamic analysis to verify the likely invariants assumed hold true for each analyzed execution. If the checks determine that the likely invariants are in fact true for this execution, the execution will produce a sound, precise, and relatively efficient dynamic analysis. If the additional checks find that the invariants do not hold, the analysis needs to compensate for the unsoundness caused by predicated static analyses.

The rest of this section describes the three analysis steps, and important design considerations.

4.2.1 Likely Invariant Profiling

A predicated static analysis is more precise and scalable than traditional static analysis because it uses *likely invariants* to reduce the program states it considers. Likely invariants are learned though a dynamic profiling pass. We next discuss the desirable properties of a likely invariant, and how optimistic hybrid analysis learns the invariants by profiling executions.

Strong: By assuming the invariant, we should reduce the state space searched by predicated static analyses. This is the key property that enables invariants to help our static phase; if the invariant does not reduce the state space considered statically, the dynamic analyses will see no improvement.

Cheap: It should be inexpensive to check that a dynamic execution obeys the likely invariants. For soundness, the final dynamic analysis must check that each invariant holds during an analyzed execution. The cost of such checks increase the cost of the final dynamic analysis, so the net benefit of optimistic hybrid analysis is the time saved by eliding dynamic runtime checks minus the cost of checking the likely invariants. Note that the time spent in the profiling stage to gather likely invariants is done exactly once, and is therefore less important; only the checks verifying the invariants need to be inexpensive.

Stable: A likely invariant should hold true in most or all executions that will be analyzed dynamically. If not, the system will declare a mis-speculation, and recovering from such mis-speculations may be expensive for some analyses.

There is a trade-off between stability and strength of invariants. We find it sufficient to consider invariants that are true for all profiled executions. However, we could aggressively assume a property that is infrequently violated during profiling as a likely invariant. This stronger, but less stable invariant may result in significant reduction in dynamic checks, but increase the chance of invariant violations. If the reduced checks outweigh the costs of additional invariant violations this presents a beneficial trade-off.

4.2.2 Predicated Static Analysis

The second phase of optimistic hybrid analysis creates an unsound static analysis used to elide runtime checks and speed up the dynamic analysis. Traditional static analysis can elide some runtime checks. However, to ensure soundness, such static analysis conservatively analyzes not only all states that may be reached in an execution, but also many states that are not reachable in any legal execution. This conservative analysis harms both accuracy and scalability of static analysis.

A better approach would be for the static analysis to explore precisely the states that will be visited in dynamically analyzed executions. A predicated static analysis tries to achieve this goal by predicting these states through profiling and characterizing constraints on the states as likely invariants. By exploring only a constrained state space of the program (the states predicted reachable in future executions), predicated static analysis provides a fundamentally more precise analysis.

This reduction of state space also improves the scalability of static analysis, which now need perform only a fraction of the computation a traditional static analysis would. Static analysis algorithms frequently trade-off accuracy for scalability [54, 109, 114, 77]. In some instances this improved efficiency allows the use of sophisticated whole-program static analyses that are traditionally more precise but often fail to scale to large programs.

4.2.3 Dynamic Analysis

The final phase of optimistic hybrid analysis produces a sound, precise and relatively efficient dynamic analysis. Dynamic analysis is implemented by instrumenting a binary with additional checks that verify a property such as data-race freedom and then executing the instrumented binary to see if the verification succeeds.

In our work, the instrumentation differs from traditional dynamic analysis in two ways. First, we elide instrumentation for checks that static analysis has proven unnecessary; this is done by hybrid analysis also, but we elide more instrumentation due to our unsound static

analysis. Second, we add checks that verify that all likely invariants hold true during the execution and violation-handling code that is executed when a verification fails.

To elide instrumentation, this phase consumes the set of unneeded runtime checks from the predicated static analysis phase. For instance, a data-race detector will instrument all read/write memory accesses and synchronization operations. The static analysis may prove that some of these read/write or synchronization operations cannot contribute to any races, allowing the instrumentation to be elided. Since the overhead of dynamic analysis is roughly proportional to the amount of instrumentation, eliding checks leads to a commensurate improvement in dynamic analysis runtime.

The instrumentation also inserts the likely invariant checks. By design, these invariants are cheap to check, so this code is generally low-overhead and simple. For example, checking likely unused basic blocks requires adding an invariant violation call at the beginning of each assumed-unused basic block. This call initiates rollback and re-execution if the check fails.

We currently handle invariant violations with a catch-all approach: roll back the entire execution and re-analyze it with traditional (non-optimistic) hybrid analysis. We find that invariant violations are so rare that even this simple approach has minimal impact on overall analysis time. If the cost of rollback became an issue, we could reduce the frequency of rollbacks through more profiling or explore cheaper rollback mechanisms, such as partial roll-back or partial re-analysis.

4.3 Static Analysis Background

OptFT and OptSlice are built using several *data-flow* analyses, such as backward slicing, points-to, and may-happen-in-parallel. Data-flow analysis approximate how some property propagates though a program. To construct this approximation, a data-flow analysis builds a conservative model of information flow through the program, usually using a definition-use graph (DUG). The DUG is a directed graph that creates a node per def-

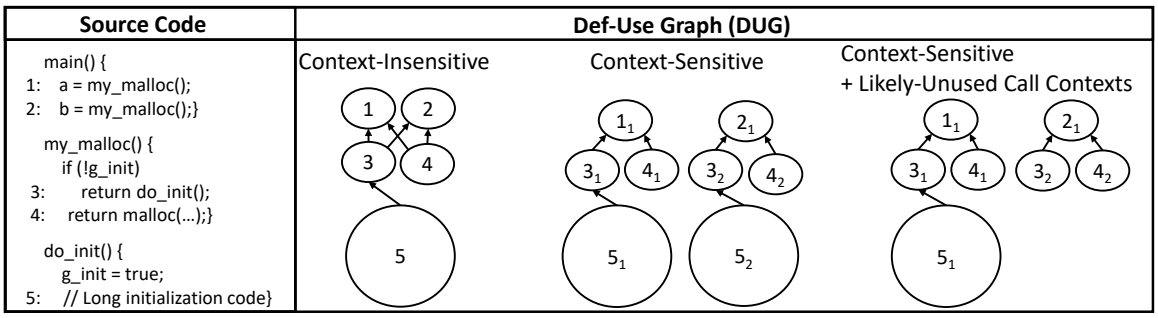


Figure 4.2: This figure shows how context-sensitive and context-insensitive analysis parse a code segment to construct a DUG, as well as the reductions from likely-unused call contexts

inition (def) analyzed. For example, a slicing DUG would have a node per instruction, while a points-to analysis would have nodes for pointer definitions. Edges represent information flow in the program between defs and the defs defined by uses. For example, an assignment operation in a points-to analysis creates an edge from the node representing the assignment's source operand to the node representing its destination. Once the DUG is constructed, the analysis propagates information through the graph until a closure is reached. To create optimistic versions of these data-flow analyses, we leverage likely invariants to reduce the number of paths through which information flows in the DUG.

There are many modeling decisions that an analysis must make when constructing the DUG. One critical choice is that of context-sensitivity. A call-site context-sensitive analysis logically distinguishes different call-stacks, allowing more precise analysis. A context-insensitive analysis tracks information flow between function calls, but does not distinguish between different invocations of a function.

Logically, a context-insensitive analysis simplifies and approximates a program by assuming a function will always behave the same way, irrespective of calling context. To create this abstraction, context-insensitive analyses construct what we call local DUGs for each function by analyzing the function independently and creating a single set of nodes in the graph per function. The analysis DUG is then constructed by connecting the nodes of the local DUGs at inter-function communication points (e.g. calls and returns).

A context-sensitive analysis differs from a context-insensitive analysis by distinguish-

ing all possible calling contexts of all functions, even those which will never likely occur in practice. To create this abstraction, the DUG of the analysis replicates the nodes defined by a function each time a new calling-context is discovered during the DUG construction. One simple method of creating such a DUG is through what is known as a bottom-up construction phase, in which the analysis begins at main, and for each call in main it creates a clone of the nodes and edges of the local DUG for the callee function. It then connects the arguments and return values to the call-site being processed. If that callee function has any call-sites, the callee is then processed in the same bottom-up manner. This recurses until all callees have been processed, resulting in a context-sensitive DUG representing the program. The context-sensitive expression of the DUG is much larger than that of a context-insensitive analysis, but it also allows for more precise analysis.

Figure 4.2 illustrates the differences between DUGs constructed by a context-sensitive and insensitive analysis. Nodes 3, 4, and 5 are replicated for each call to $my\_malloc()$, allowing the analysis to distinguish between the different call-contexts, but replicating the large $do\_init()$ function.

Context-sensitive analyses tend to be precise, but not fully scalable, while context-insensitive analyses are more scalable at the cost of accuracy. We build both context-sensitive and insensitive variants of several predicated static analyses.

4.4 OptFT

To show the effectiveness of optimistic hybrid analysis, we design and implement two sample analyses: OptFT, an optimistic variant of the FastTrack race detector for Java, and OptSlice, an optimistic dynamic slicer for C programs. This section describes OptFT and Section 4.5 describes OptSlice.

OptFT is a dynamic data-race detection tool that provides results equivalent to the FastTrack race detector [48]. FastTrack instruments load, store, and synchronization operations to keep vector clocks tracking the ordering among memory operations. These vector clocks

are used to identify unordered read and write operations, or data-races.

OptFT uses the Chord analysis framework for static analysis and profiling, building on Chord's default context-insensitive static data-race detector [85]. For dynamic analysis we use the RoadRunner [49] analysis framework, optimizing their default FastTrack implementation [48].

4.4.1 Analysis Overview

The Chord static data-race detector is a context-insensitive, lockset-based detector. The analysis depends on two fundamental data-flow analyses, a may-happen-in-parallel (MHP) analysis, which determines if memory accesses may happen in parallel, and a points-to analysis, which identifies the memory locations to which each pointer in the program may point.

The analysis first runs its static MHP analysis to determine which sets of loads and stores could dynamically happen in parallel. Once those sets are known, the analysis combines this information with a points-to analysis to construct pairs of potentially racy memory accesses which may alias and happen in parallel. Finally, the analysis uses its points-to analysis to identify the lockset guarding each memory access, and it uses these to exclude pairs of loads and stores guarded by the same lock from its set of potentially racing accesses.

To optimize the dynamic analysis, OptFT elides instrumentation around any loads or stores that predicated static analysis identifies as not racing. The analysis also elides instrumentation around some lock/unlock operations, as we discuss in Section 4.4.2.4.

4.4.2 Invariants

OptFT is optimized with four likely invariants.

4.4.2.1 Likely Unreachable Code

The first, and simplest, invariant OptFT assumes is likely-unreachable code. We define a basic block within the source code that is unlikely to be visited in a future execution as a *likely unreachable code* block. This invariant is learned by a basic block counting profiling pass. Any block not visited in the profile is likely unreachable.

This invariant easily satisfies the three criteria of good likely invariants. First, it is strong; the invariant reduces the state space our data-flow analyses considers by pruning nodes defined by likely unused code and any edges incident upon them from our analysis DUGs. This reduction in connectivity within the DUG can greatly reduce the amount information that propagates within the analysis. Second, the invariant is virtually free to check at runtime, requiring only a mis-speculation call at the beginning of the likely-unused code. Finally, we observe that unused code blocks are typically stable across executions.

4.4.2.2 Likely Guarding Locks

Chord's race detector's final phase prunes potentially racy accesses by identifying aliasing locksets. Unfortunately, this optimization is unsound. To soundly identify if two locksites guard a load or store, Chord needs to prove that the two sites *must* hold the same lock when executing. However, the alias analysis Chord uses only identifies *may* alias relations. To get sound results from Chord we must either forego this lock-based pruning or use a (typically unscalable and inaccurate) must alias analysis. In the past, hybrid analyses that use Chord have opted to remove this pruning phase to retain soundness [100].

Likely guarding locks attempt to overcome Chord's may-alias lockset issue by dynamically identifying must-alias lock pairs. The analysis profiles the objects locked at each lock site. If it can identify that two sites always lock the same object, it assumes a must-alias invariant for the lock pairs.

The invariant is strong. By assuming the invariant, the Chord race detection algorithm can add in some of the lockset-based pruning discarded due to its weaker *may* alias analysis.

Additionally, the invariant is cheap to check at runtime. The dynamic analysis need only instrument the assumed aliasing lock-sites and verify the sites are locking the same object, a check far less expensive than the lock operation itself. Finally, executions do not vary the objects locked frequently, so this invariant remains stable across executions.

4.4.2.3 Likely Singleton Thread

Likely singleton thread invariants aid Chord's MHP analysis. If a thread start location creates only a single instance of a thread, all memory accesses within that thread are ordered. If the start location spawns multiple threads (e.g. its executed within a loop), then the memory accesses in different threads associated with that start location may race. We call this single-thread start call a *singleton-thread* instance.

The knowledge of singleton-thread instances is easy to gather dynamically by monitoring thread start sites. On the other hand, statically reasoning about this information is hard, requiring understanding of complex program properties such as loop bounds, reflection, and even possible user inputs. The likely singleton thread invariant eliminates the need for this static reasoning by finding thread start locations called exactly once during profiled executions; this allows the static MHP analysis to prune many memory access pairs for singleton thread instances that it would otherwise miss.

The invariant easily meets the properties of a good likely invariant. First, the invariant can greatly aid the MHP analysis, which is foundational to our race detector. Second, the invariant is inexpensive to check, only requiring monitoring of predicted singleton thread start locations. Finally, the invariant is generally stable across runs.

4.4.2.4 No Custom Synchronizations

Ideally, static analysis would enable OptFT to elide instrumentation for lock and unlock operations. However, the possibility of custom synchronizations stops a sound analysis from enabling this optimization. Figure 4.3 shows how eliding lock/unlock instrumentation,

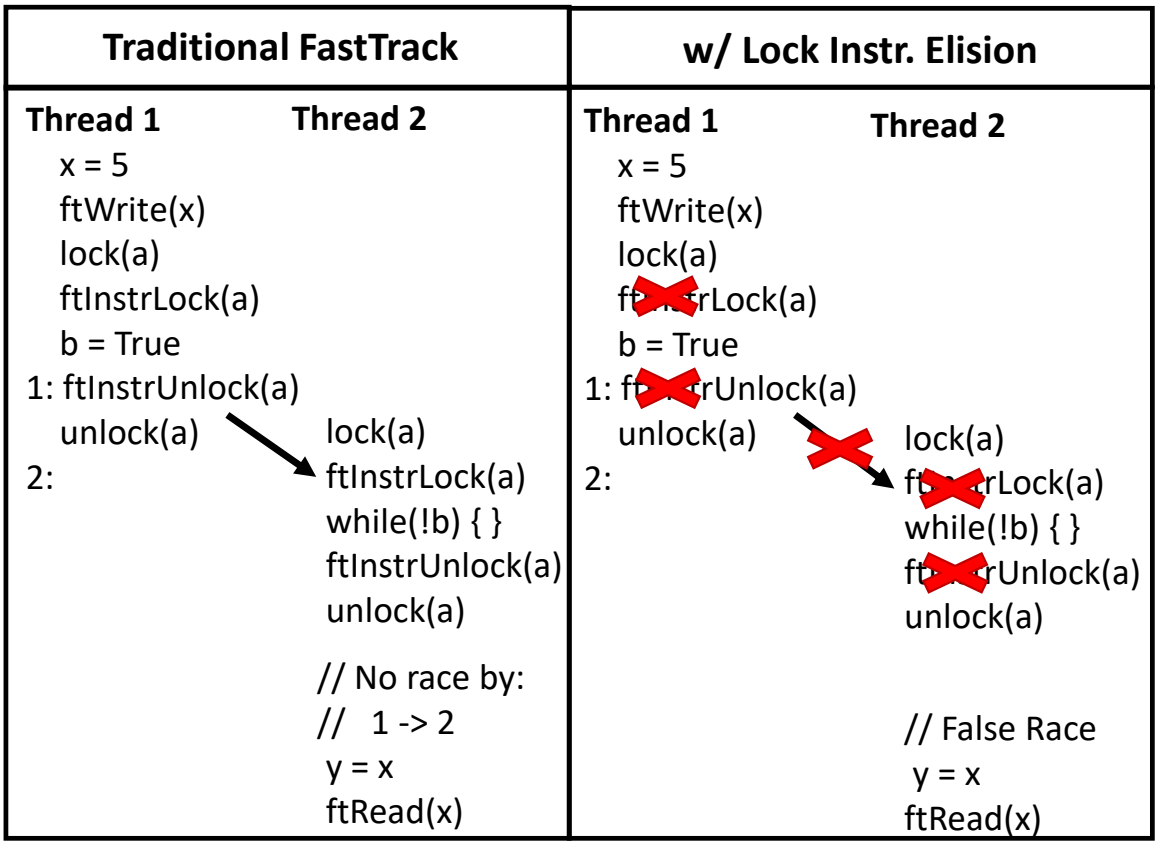


Figure 4.3: An example of how lock instrumentation elision may cause missed happens-before relations in the presence of custom synchronizations. The left hand side catches custom synchronizations, but with the elision of locking instrumentation, the necessary happens before relation (represented by an arrow) may be lost.

even when there are no racy accesses within the critical section, can cause a false race report. This problem is caused by custom synchronization (e.g. waiting on *b* in Figure 4.3).

To enable elision of lock and unlock instrumentation, we propose the no custom synchronization invariant. Using this invariant, OptFT optimistically elides lock/unlock sites whose critical sections do not contain any read or write checks. To profile this invariant, we observe if this elision causes the dynamic race detector to report false races. If so, we return the synchronization instrumentation to nearby locks until the false races are removed.

The drawback to this approach is that race reports must be considered as potential mis-speculations. This could be an undue burden if analysis frequently reports data-races; however, if a program has frequent data-races, there is little need for a highly optimized race detector.

This invariant is highly useful. First, it helps the static analysis eliminate work by reducing the instrumentation around locks. Second, it is easy to check, as our race detector already detects races. Finally, custom synchronizations rarely change between executions, so the invariant is stable.

4.5 OptSlice

OptSlice is our optimistic dynamic backward slicing tool. A backward slice is the set of program statements that may affect a target (or slice point) such as a data value at some point in the program execution. Program slices are important debugging tools, as they simplify complex programs and help developers locate the sources (i.e., root causes) of errors more easily. Backward slicing is particularly powerful when analyzing multiple executions to find differences between failing and non-failing executions [52, 7].

OptSlice optimizes the Giri dynamic slicer [97] with an optimistic variant of Weiser's classic static slicing algorithm [111]. OptSlice collects data-flow slices. Data-flow slices do not consider control dependencies and are often used when control dependencies cause a slicer to output so much information the slice is no longer useful.

4.5.1 Static Analysis

OptSlice uses a backward slicing analysis that builds on the results of a separate points-to analysis; we next describe these two analyses.

4.5.1.1 Backward Slicing

The static slicer used by OptSlice first constructs a DUG of the program. We have implemented two versions of this algorithm: a context-sensitive and a context-insensitive variant. The DUG contains a node for every instruction in the program and edges representing the reverse information flow through the program (i.e., from any defs which use instructions to the defs providing those uses). The slicing analysis resolves indirect def-use edges (e.g., loads and stores) by using a points-to analysis to determine aliases. As slicing is a flow-sensitive analysis, when resolving these indirect edges, the slicer only considers stores in basic blocks that may precede the load being analyzed according to the program's control-flow graph.

Once the DUG is constructed, our static analysis computes the conservative slice by calculating the closure of the graph, starting from any user-defined slice endpoints. The final slice is composed of any instructions whose defs are represented by the nodes within this closure.

Our optimistic backward slicer implements several optimizations. First, it lazily constructs the DUG, creating nodes only when required. Second, it uses binary decision diagrams (BDDs) [21] to keep track of the visited node set. This is similar to how BDDs are used to track points-to sets [17].

4.5.1.2 Points-To

Our Andersen's-based points-to analysis [10] constructs a DUG with a node for each statement in the program that defines a pointer. Edges represent the information flow defined by pointer uses. Unlike the slicing DUG, not all nodes and edges can be resolved at

graph creation time. These nodes and edges are dynamically added as points-to sets are discovered during the next analysis phase.

After constructing the DUG, the analysis associates an empty points-to set with each node and initializes the points-to sets of any nodes which define a pointer location (e.g. malloc calls). The algorithm then propagates points-to information according to the edges defined by the graph. Additionally, the algorithm may add edges to the graph as indirect def-use pairs are discovered; e.g., if a load-source and store-destination are found to alias, the analysis will make an edge between the two nodes. After all information has finished propagating through the DUG, each node has its conservative set of potential pointers.

Indirect function calls are handled in a special manner. For context-insensitive analyses, when the pointer used in an indirect function call is resolved, the arguments and return values are connected to the existing nodes for the resolved function(s) in the graph. Context-sensitive analyses, however, require distinct information pathways for different static call stacks. In a context-sensitive analysis, nodes may have to be added to the graph. When an indirect function call is resolved in a context-sensitive analysis, the analysis scans the call stack to check for recursive calls. If the new callee creates a recursive call, the call is connected to the prior nodes in the DUG representing that callee. If the function call is not recursive, a new set of nodes must be added for the call in the same manner as for the bottom-up DUG construction phase.

The analysis is complete once the graph reaches transitive closure. Each def's points-to set is the union of the points-to sets of all nodes in the graph representing that def.

Our algorithm also provides heap cloning and structure-field sensitivity. It also includes several well-known optimizations, including offline graph optimizations (HVN/HRU) [57], online cycle detection (LCD and HCD) [56], and using BDDs to track points-to sets [17]. While these optimizations contribute dramatically to both the scalability and accuracy of the analysis, they do not impact how we apply likely invariants, so we do not discuss them further.

4.5.2 Invariants

OptSlice uses several general invariants, aimed at increasing overall analysis accuracy. After the invariants are profiled, we use them to reduce the set of states our static analysis considers. Below, we discuss how each invariant affects DUG construction.

4.5.2.1 Likely Unreachable Code

OptSlice uses likely unreachable code identically to OptFT.

4.5.2.2 Likely Callee Sets

Our points-to analysis's indirect function call resolution process can lead to considerable slowdowns, increased memory consumption, and analysis inaccuracies. If the analysis is unable to resolve the destination pointer of an indirect call to a function, it may have to conservatively assume that the callee may be *any* function in the program, connecting the call-site arguments of this function to all functions. On its own, this is a major source of inaccuracy. It also can lead to propagating inaccuracies if a function argument is used later as an indirect call. This issue is compounded in context-sensitive analyses, where the nodes in the local DUGs for all functions are replicated, potentially dramatically increasing the analysis time. This problem occurs in programs like Perl (Perl is an interpreter that has a generic variable structure type that holds all types of variables, including ints, floats, structures, and function pointers).

Likely callee sets are the dynamically-gathered likely destinations of indirect function calls. This invariant helps resolve many of the inaccuracies and inefficiencies that unknown indirect calls can add to our points-to analysis. The invariant is easily profiled by monitoring the values of all function pointers in the program.

This invariant converts all indirect calls in the DUG to direct calls to the assumed callee functions. The invariant is relatively inexpensive to check at runtime, requiring only a relatively small (usually singleton) set inclusion check on a function pointer update (a relatively

rare operation). Most indirect function calls have very small sets of destinations, and they don't vary from execution to execution, making this invariant stable across executions.

4.5.2.3 Likely Unused Call Contexts

Context-sensitive analyses can suffer significant scalability problems due to excessive local DUG cloning, as discussed in Section 4.3. Likely callee set invariants minimize local DUG cloning by stopping the context-sensitive analysis from cloning DUGs for call contexts, or call stacks, that are unlikely to occur. This invariant is profiled by creating a caller stack for each thread. This caller stack is then used by the context sensitive analysis to limit local DUG cloning around unrealized call chains. This effect is demonstrated in Figure 4.2, removing the likely-unrealized second call to $do\_init()$.

Likely unused call contexts easily meet two of the criteria for good likely invariants. First, they are strong, as they can dramatically reduce the size of the DUG and, ultimately, the amount of space the data-flow analysis explores. Second, the invariant tends to be stable across executions.

Cheap checking of Likely unused call contexts is a slightly more complex matter. Logically, the check needs to ensure that unused call contexts are never reached, requiring a call-stack set inclusion check at many call-sites. We found that a naive implementation of this functionality was too inefficient for some programs. To accelerate it, we use a Bloom filter to elide the majority of our relatively expensive set inclusion tests. We find that this methodology makes the dynamic cost of the invariant check acceptable.

4.6 Evaluation

In this section, we show that optimistic hybrid analysis can dramatically accelerate dynamic analyses by evaluating our two sample analyses, OptFT and OptSlice, over a wide range of applications.

4.6.1 Experimental Setup

4.6.1.1 OptFT

We evaluate the effectiveness of OptFT on the Dacapo [18] and JavaGrande [102] benchmark suites. Our test suite is composed of all multi-threaded benchmarks from these suites which are compatible with our underlying Chord [85], and RoadRunner [49] frameworks.

Optimistic hybrid analysis requires considerable profiling, more than the single profile set provided by these benchmark suites. To test these applications we construct large profiling and testing sets for each benchmark. For several benchmarks we use large, readily-available input sets:

- **lusearch** – Search novels from Project Gutenberg [3].
- **pmd** – Run the pmd source code analysis tool across source files in our benchmarks.
- **montecarlo** – Analyze 10 years of S&P 500 stock data.
- **batik** – Render svg files from svgcuts.com [4].
- **xalan** – Convert xhtml versions of pydoc 2.7 Web pages to XSL-FO files.
- **luindex** – Index novels from Project Gutenberg [3].

The remainder of our benchmarks (sunflow, raytracer, sor, moldyn, lufact, crypt, series, and sparse) benchmarks do not have large, freely available input sets, so we generated large sets by sweeping parameters across the input sets (e.g. input size, number of threads, pseudo-random seed).

To profile OptFT, we generate two sets of 64 inputs for each test. One set is our candidate profiling runs; the other is our testing corpus.

We run OptFT as a programmer would use the tool on a large set of regression tests. We first profile increasing numbers of profiling executions, until the number of learned dynamic invariants stabilize. Then, we run OptFT over all tests in our testing set. We run all data race detection experiments using 8 cores of an Intel Xeon E5-2687W v3 3.1 GHz

processor.

4.6.1.2 OptSlice

OptSlice is implemented in the LLVM-3.1 compiler infrastructure. We accelerate the Giri dynamic backward slicer [97]. We evaluate the effectiveness of our analysis over a suite of common desktop and server applications.

Our test suite workloads consist of:

- **nginx** – Serve a clone of the pydoc 2.7 documentation, and load random pages.
- **redis** – Redis-benchmark application with varying data-size, client, and number of requests.
- **perl** – The SPEC2006 diffmail application with different inbox configurations.
- **vim** – Problem solutions from vimgolf.com [2].
- **sphinx** – Process a large database of voice samples.
- **go** – Randomly predict the next best move from random points in an archive of professionally played go games.
- **zlib** – Compress files with a gzip-like utility. Input files are sampled from our sphinx input files.

Much as we did for OptFT, we generate profile and testing sets (512 files each for redis, zlib, sphinx, perl, and nginx; 2048 for go and vim). We profile each program, growing the input set until the number of dynamic invariants stabilizes. We then test on our testing set of inputs. This methodology is consistent with how we imagine OptSlice may be used for debugging, such as when comparing good and failing executions of a program.

We select static slices from several random locations within our benchmarks, using the most accurate static analysis that will complete on that benchmark without exhausting available computational resources.

Once we have gathered our set of slices, we generate dynamic slicing instrumentation.

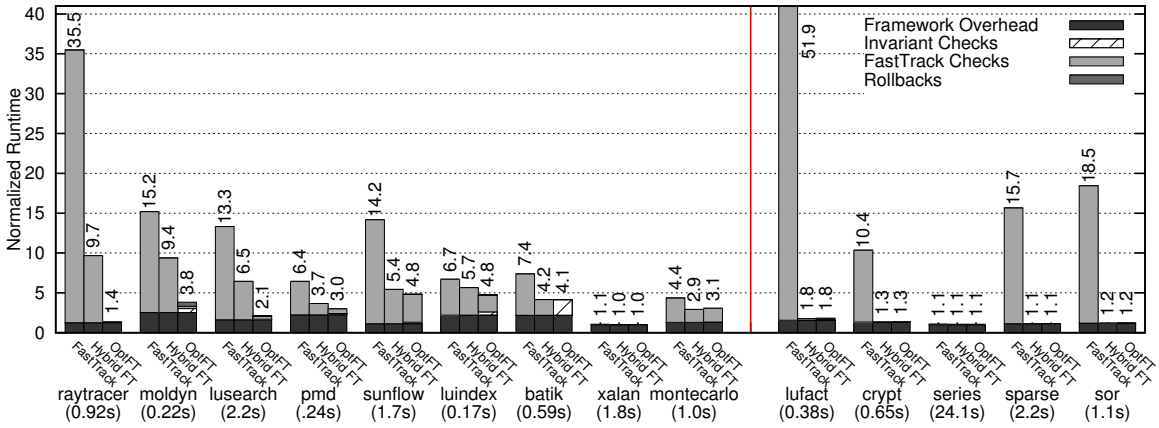


Figure 4.4: Normalized runtimes for OptFT. Baseline runtimes for each benchmark are shown in parentheses. Tests right of the red line are proven race-free by sound static race detection, but included here for completeness.

To determine statistical significance between good and bad runs of a program, a developer would start at a suspect instruction and calculate the backward slice over many executions (both failing and successful). We therefore select non-trivial endpoints for calculating such slices and calculate the slice from each endpoint for each execution in the testing set. We define a non-trivial endpoint to be an instruction with a sound static slice containing at least 500 instructions. We use non-trivial endpoints because they tend to be far more time consuming to compute slices (there is little use optimizing a trivial analysis), and they are common; on average, 55% of the endpoints from our sound static slicer are non-trivial.

We slice each endpoint with the most accurate predicated static slicer that will run on that program. Once we have our predicated static slices, we optimize our dynamic instrumentation, and dynamically slice all tests in our testing set with our dynamic slicer. We repeat this process until we have analyzed five different program endpoints; this provides a sufficient sample set of endpoints to gain confidence in OptSlice's ability to optimize slicing for a general program point.

All experiments are run using a single core of our Xeon processor, and each may use up to 16 GB of RAM. Table 4.2 gives an overview of our benchmarks, their relative sizes, static analysis times, and which static analysis we use.

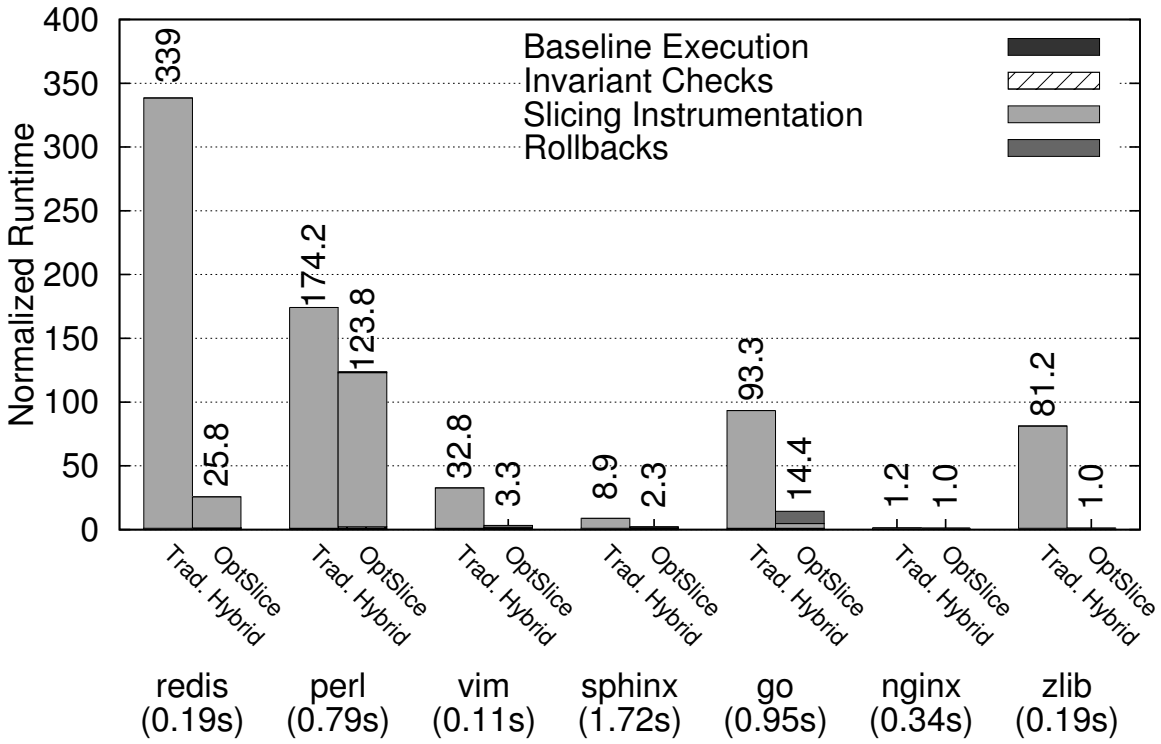


Figure 4.5: Normalized runtimes for OptSlice. Baseline runtimes for each benchmark are shown in parentheses.

4.6.2 Dynamic Overhead Reduction

Figure 4.4 shows how optimistic hybrid analysis improves performance for race detection. Although we show all benchmarks for completeness, 5 benchmarks (those to the right of the vertical line in Figure 4.4) are quite simple and can be statically proven to be race-free. Thus, there is no need for either traditional hybrid analysis or our optimistic hybrid analysis in these cases. For the remaining 9 benchmarks, OptFT shows average speedups of 2.9x versus traditional FastTrack, and 1.6x versus hybrid FastTrack. Impressively, for the first four benchmarks analyzed, the costs of OptFT approach those of the underlying RoadRunner instrumentation framework; this presents a near-optimal reduction in work due to the OptFT algorithm.

There are three remaining benchmarks for which OptFT sees limited speedups: batik, sunflow and montecarlo. Batik experiences a significant number of likely-invariant checks, which could probably be reduced with more cautious invariant selection. Montecarlo and

sunflow both make considerable use of fork-join and barrier based parallelism. Consequently, the lockset based Chord detector is algorithmically unequipped to optimize their memory operations, even with optimistic invariants. A static analysis algorithm better equipped to deal with barrier based parallelism would likely see more profound speedups from optimistic hybrid analysis.

Figure 4.4 shows that the additional invariant checking and mis-speculation overheads associated with OptFT are negligible for nearly all benchmarks, with moldyn and batik being the only benchmarks to experience notable mis-speculation or invariant checking overhead. Overall, invariant checking overheads have little effect on the runtime of our race detector, averaging 7.8% relative to a baseline execution. Additionally roll-backs are infrequent and cause little overhead, ranging from 0.0% to 20.3% and averaging 6.2%.

Figure 4.5 shows the online overheads for OptSlice versus a traditional hybrid slicer. We do not compare to purely dynamic Giri, as it exhausts system resources even on modest executions. OptSlice dramatically reduces the runtime of dynamic slicing, with speedups ranging from 1.2x to 78.5x, with an average speedup of 8.3x. Our worst absolute speedups are from perl and nginx. Perl's state is divided largely into two subsets, the interpreter state and the script state. Without knowledge of the script running in the interpreter, static analysis cannot precisely determine how information flows through the script state. Perl scripts would be better analyzed at the script level. Nginx is largely I/O bound, but OptSlice decreases its overhead from 20% to a statistically insignificant overhead. This reduction is relatively significant, even though it is not absolutely large.

We also look at the invariant-checking and mis-speculation costs of OptSlice. The overheads of ensuring executions do not violate likely-invariants are generally inconsequential, showing no measurable runtime overhead for zlib, go, nginx, and vim. Perl and sphinx have overheads of 26% and 127% respectively, largely due to likely-unrealized call-context checking. These overheads are low enough for optimistic hybrid analysis to improve slicing performance, but could be optimized further if lower overheads are needed [19]. Overall

Testname	Trad. Hybrid	Opt. Hybrid		Break-even w/ respect to		Opt. Speedup w/ respect to	
	Static Time	Profile	Static Time	Hybrid FT	Trad. FT	Hybrid FT	Trad. FT
lusearch	1m 15s	1m 12s	1m 47s	24s	16s	3.0x	6.3x
pmd	1m 25s	20s	2m 36s	2m 19s	51s	1.2x	2.1x
raytracer	31s	4m 17s	54s	33s	9s	7.2x	26.2x
moldyn	29s	10m 38s	46s	1m 58s	1m 0s	2.4x	4.0x
sunflow	3m 0s	11m 55s	3m 45s	20m 57s	1m 40s	1.1x	2.9x
montecarlo	59s	55s	52s	–	1m 23s	0.95x	1.4x
batik	3m 25s	2m 28s	7m 50s	114m 1s	3m 7s	1.0x	1.8x
xalan	55s	51s	1m 26s	361m 2s	59m 57s	1.0x	1.0x
luindex	1m 7s	1m 8s	1m 56s	2m 10s	1m 35s	1.2x	1.4x

Table 4.1: Comparing FastTrack benchmark end-to-end analysis times for pure dynamic as well as traditional and optimistic hybrid analyses. Break-even Time is the amount of baseline execution time at which optimistic analysis begins to use less computational resources (profiling + static + dynamic) than a traditional analysis. Optimistic Speedup is the ratio of runtimes for OptFT versus a traditional or hybrid FastTrack implementation.

Testname (LOC)	Traditional				Optimistic					Break-even Time	Dynamic Speedup
	Points-to		Slice		Profiling Time	Points-to		Slice			
	AT	Time	AT	Time		AT	Time	AT	Time		
nginx (119K)	CI	17s	CI	24m 33s	1m 4s	CS	8s	CS	3s	0s	1.2x
redis (80K)	CI	1m 46s	CI	170m 46s	1m 4s	CI	6s	CS	48s	0s	13.1x
perl (128K)	CI	24s	CS	55m 0s	10m 29s	CS	160m 33s	CS	9m 11s	2m 29s	1.4x
vim (306K)	CI	27s	CI	77m 55s	11m 8s	CS	1m 20s	CS	21s	0s	9.9x
sphinx (13K)	CS	7s	CS	1s	11m 24s	CS	6s	CS	0.2s	1m 44s	3.9x
go (158K)	CI	6s	CI	59s	133m 54s	CI	8s	CI	9s	1m 41s	6.5x
zlib (21K)	CS	14s	CS	33s	1m 59s	CS	5s	CS	0.4s	1s	81.2x

Table 4.2: Comparing slicing benchmark end-to-end analysis times for traditional hybrid and optimistic hybrid analyses. Shown are a breakdown of offline analysis costs for static points-to and slicing analyses and the most accurate Analysis Type (AT), either Context-Sensitive (CS) or Context-Insensitive (CI) that will run on a given benchmark. Break-even Time is the minimum amount of baseline execution time where an optimistic analysis uses less total computational resources (profiling + static + dynamic) than a traditional hybrid analysis. Dynamic Speedup is the ratio of run-times for OptSlice versus a traditional hybrid implementation.

mis-speculation rates are low for all benchmarks, with go and vim being the only benchmarks to see even modest overheads.

So far, we have looked at speedups of using optimistic hybrid analysis when the profiling and static analysis costs are inconsequential. This is typical when static analysis can be done offline (e.g., after code is released but before the first bug is reported), or for very large analysis bases, such as analyzing months or years of prior executions in forensic queries. We next look at how much execution time must be analyzed for the dynamic savings of an optimistic hybrid analysis to overcome its analysis and profiling startup costs for smaller execution bases, which might occur when running nightly regression tests or when using

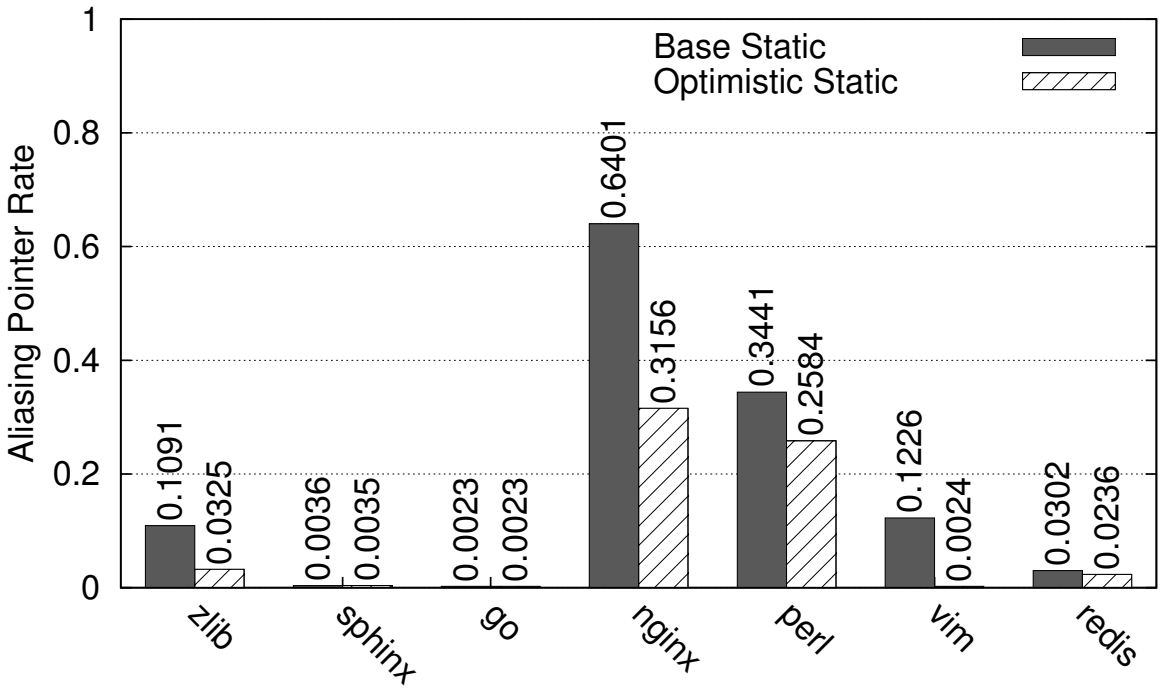


Figure 4.6: Alias rates for points-to analyses, reported as a chance that a store may alias with a load.

delta debugging for moderate sets of inputs immediately after recompilation.

Table 4.1 shows break-even times for benchmarks not statically proven race-free. OptFT begins to out-perform both traditional and hybrid FastTrack within a few minutes of test time for most benchmarks. There are exceptions, such as montecarlo, sunflow, batik, and xalan, for which OptFT does not speed up dynamics analysis and therefore should not be used.

OptSlice shows a similar breakdown in Table 4.2, which compares OptSlice to a traditional hybrid slicer. This chart shows similar static analysis and profiling times as OptFT; however, due to the both the larger dynamic speedup of OptSlice and the reduction in static analysis state provided by our likely invariants, the break-even times are generally much lower. In three cases (vim, redis, and nginx), it is on average better to run a hybrid slicer when analyzing *any* execution length. In all cases, with under 3 minutes of execution time analyzed, OptSlice will save work versus a traditional hybrid analysis.

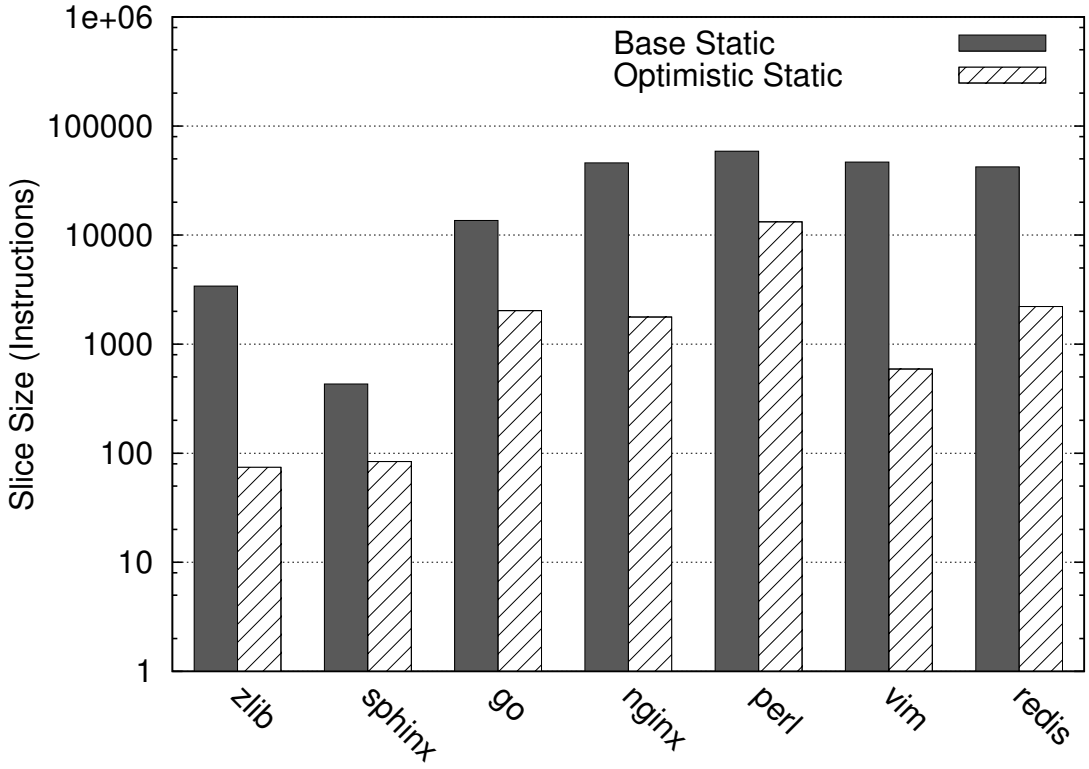


Figure 4.7: The static slice sizes, in number of instructions, as reported by a sound and a predicated static slicer. Optimistic analysis shows *order of magnitude* improvements in slice size.

4.6.3 Predicated Static Analysis

We next evaluate the effects of predicated static analysis on our C-based points-to and slicing analyses. Both are general-purpose analyses with many applications. In fact, points-to analysis is foundational to most complex static analyses; any improvement to points-to analysis will have wide ranging effect on the many analyses that depend on points-to analysis.

Figure 4.6 shows how a predicated static analysis significantly increases the accuracy of an alias analysis. Alias rates are measured as the probability that any given load can alias with any given store. For fairness, both baseline and optimistic analyses consider only the set of loads and stores present in the optimistic analysis (this is a subset of the baseline set due to state reduction caused by likely invariants). Figure 4.7 shows the reduction in overall slice sizes, with optimistic analysis providing one to two orders of magnitude in slice reduction.

We next break down how the likely invariants individually benefit static analyses. Figure 4.8 measures static slice size when running a sound static analysis and incrementally adds each likely invariant for three tests: vim, nginx, and zlib. The introduction of the likely-unrealized call-context invariant allows vim and nginx to scale to context-sensitive slicing and points-to analysis, causing a large reduction in slice sizes.

4.7 Conclusion

We argue that the traditional application of a sound static analysis to accelerate dynamic analysis is suboptimal. To this end, we introduce the concept of optimistic hybrid analysis, an analysis methodology that combines unsound static analysis and speculative execution to dramatically accelerate dynamic analysis without the loss of soundness. We show that optimistic hybrid analysis dramatically accelerates two dynamic analyses: program slicing and dynamic race detection.

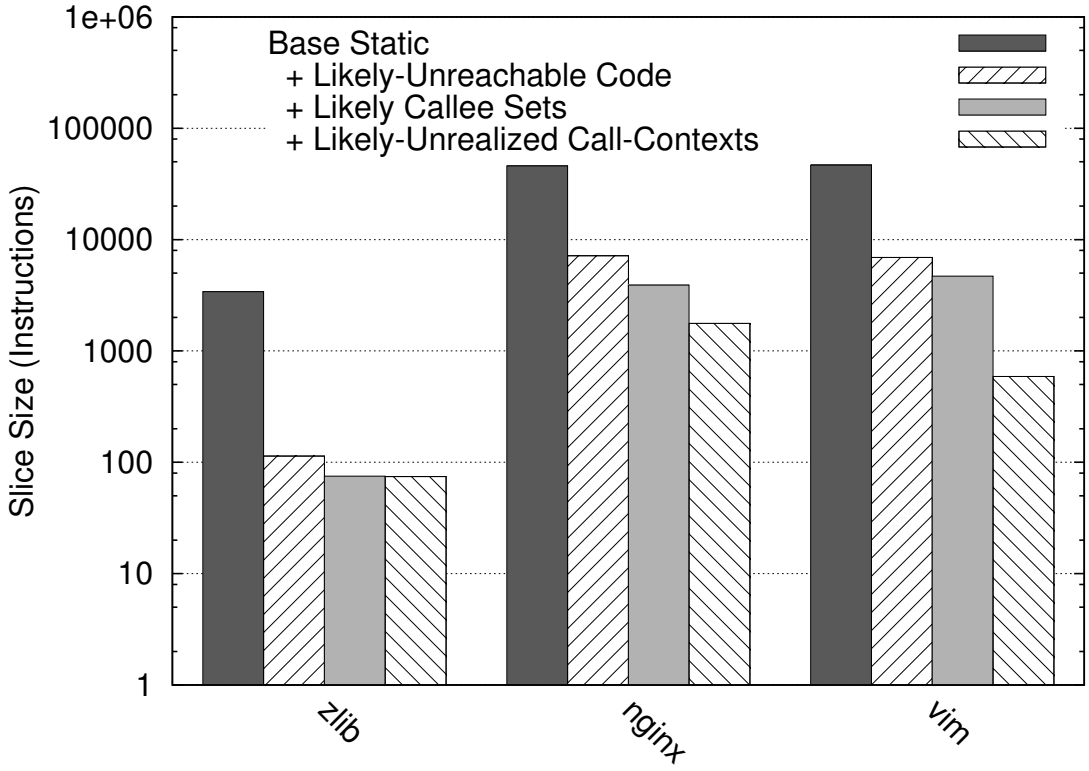


Figure 4.8: The effect different likely invariants have on slice size. Vim and nginx switch to a context-sensitive analyses when adding likely-unrealized call-context elimination.

CHAPTER V

Bounded Rollback for Optimistic Hybrid Data-Race Detection

Dynamic analysis tools, such as data race detectors [98, 48], information flow tools [44, 63], and memory safety detectors [87, 88], are essential to maintaining reliable software systems. Unfortunately, many of these tools introduce significant runtime overheads, some tools introducing one to two orders of magnitude runtime overhead. These prohibitive overheads often limit the adoption of dynamic analyses in practice, even when the dynamic analysis is otherwise highly desirable.

Optimistic Hybrid Analysis, or OHA (described in Chapter IV), is a promising technique that dramatically reduces the common case overhead of dynamic analysis. It reduces these overheads by speculating about properties of future executions and using those speculations to aid in analysis optimization. First, it dynamically profiles the application, learning common properties, or *likely invariants* about the application. Second, these likely invariants are assumed to be true by an otherwise sound static analysis known as a *predicated static analysis*. Finally, the results of the predicated static analysis are used to speculatively optimize the final dynamic analysis. If the likely invariants hold true during the final execution, the predicated static analysis will be sound, and the results of the dynamic analysis will be correct.

However, OHA is optimized based on predicated static analysis, an unsound analy-

Traditional Hybrid	Optimistic Hybrid	OHA + Recovery?

foo () {
1. r = x; $taint_r = taint_x$
2. a = y; $taint_a = taint_y$
if (c) {
 	
3. r += y;
$taint_r = taint_r \cup taint_y$
}
 	
return r; }
	foo () {
	1. r = x; $taint_r = taint_x$
	2. a = y; // taint elided
	if (c) {
	LUC_Check();
	3. r += y;
	// taint elided
	}
	
	return r; }
		foo () {
		1. r = x; $taint_r = taint_x$
		2. a = y; // taint elided
		if (c) {
		LUC_Check(); **// Failed**
		3. r += y;
		$taint_r$ = ???
		}
		
		return r; }

Figure 5.1: Demonstration of how prior state can be missing after a mis-speculation in an OHA analysis. In the "Traditional Hybrid" column, all information is gathered. In the "Optimistic Hybrid" column statement 3 is assumed to never be executed. The "OHA + Recovery" column experiences a mis-speculation when "c" is true, and must re-execute statement 2 with a conservative analysis to recover $taint_a$ for statement 3.

sis. When a likely invariant fails, the predicated static analysis is no longer sound for the execution, and the prior instrumentation elisions caused by the predicated static analysis may have removed dynamic analysis state required for future analysis checks. Figure 5.1 demonstrates how after an invariant violation an OHA can require past state. In the example, the "Optimistic Hybrid" column elides instrumentation around statement 2, as the taint is never used under the assumption that "c" is false. However, when "c" is dynamically set to be true, the taint data from statement 2 is missing when its needed by statement 3. To recover this state, once a likely invariant violation is detected OHA rolls back execution. Since the elisions caused by predicated static analysis may effect arbitrary past analysis state computations, OHA conservatively rolls back to the beginning of the execution.

In the worst case, rollbacks cause OHA to be more costly than a traditional hybrid analysis (dynamic analysis optimized with sound static analysis), and may introduce extreme and unpredictable latency spikes within live or long-running applications. Consider a web-server running an optimistic hybrid taint-tracking analysis. Web-servers frequently run for weeks, months, or years at a time without restart. In the instance of an invariant violation, the web-server would have to re-execute the entire computation, with a slow,

conservative analysis, potentially adding an order of magnitude to computation time. If the web-server were running for one month, a rollback could add 10 months of latency, an entirely unacceptable scenario. This high-overhead, high-latency worst-case scenario prohibits the adoption of OHA in many production systems or on long running applications, where these worst-case scenarios are relevant.

In this chapter we focus on minimizing the amount of recovery time an OHA must experience in the instance of rollback. We focus on why it is challenging to identify safe rollback points within an OHA, as well as techniques which help alleviate this problem. Finally, we apply these techniques, as well as observations specific to data-race detection, to create an optimistic hybrid data-race detector OptFT-BR.

To understand the challenges of identifying safe rollback points for an OHA, consider the problem of trying to find rollback points on an OHA. A conceptually simple way to identify a safe rollback point is to find a point in the execution where the dynamic analysis state during an OHA is identical (all bits are equal) to the state that would be gathered during a conservative analysis. However, there is no known way to generally monitor this state efficiently. One solution would be for an OHA to dynamically check to see when its state deviates from a conservative analysis, however this would introduce the same overhead as a traditional hybrid analysis, the exact work an OHA is attempting to avoid. An alternative solution is for OHA to guarantee its dynamic state wont diverge from a traditional analysis by design, however this isn't practical for many analyses.

Additionally, to limit rollback, OHA must solve the issue of analysis state divergence. Once the conservative analysis state diverges from the analysis state of an optimistic analysis, the two states are unlikely to re-converge. Furthermore, as the execution continues the divergences are likely to propagate through the analysis, making it unlikely the two analyses will ever reach an identical state. For these reasons, a naive approach to identifying safe and frequent rollback points within a program is insufficient for OHA.

To efficiently identify rollbacks, for an OHA, we leverage three techniques to solve the

problems of identifying analysis state equivalence, and analysis state divergence.

First, we leverage analysis specificity. By leveraging specificity we can use analysis specific properties to identify analysis states guaranteed to produce equivalent results, removing the need to run a fully conservative analysis just to determine a rollback point.

Second, we observe that the desired property of an OHA is that it produce results equivalent to that of a traditional analysis. This restriction is much softer than that of requiring identical analysis states at rollback, especially when combined with analysis specific observations. For many analyses, we can identify analysis states which are guaranteed to produce equivalent results, although they themselves are not precisely equal. For example, in data-race detection we can observe that if all threads in an application synchronize (e.g. a barrier), then any memory access after the synchronization cannot race with memory accesses before the synchronization. This means the synchronization point is an excellent checkpoint candidate for rollback as it provides logical analysis equivalence, even though restoring that checkpoint is unlikely to result in identical analysis state.

Finally, we observe that by recreating some analysis states through bounded re-execution, we can greatly increase the number of safe rollback points. Points of equivalent analysis state, while more common than identical analysis state are still not ubiquitous throughout the executions of many programs. By re-executing a bounded amount of computation on rollback, we can recreate analysis states otherwise missed by OHA, increasing the number of potential rollback locations. We call this re-execution region a *recovery region*. Recovery regions are regions of execution, that when executed with a conservative analysis guarantee the analysis state at the end of the region is equivalent to that of a traditional hybrid analysis.

Recovery regions have the advantage of allowing analysis state recovery to occur over a range of execution states. The consequence of this is that it is much more likely to find recovery regions in most applications than individual checkpoints, reducing the analysis state divergence problem. Consider data-race detection. For a checkpoint to exist without

recovery regions there would have to be a single point in the program where all memory accesses before that point could not race with any memory accesses after. Outside of a barrier operation for all threads, this is unlikely to occur. On the other hand, recovery regions for data-race detection are relatively common, as we discuss later in this chapter.

In this paper we present OptFT-BR, an optimistic variant of the FastTrack race detector which supports limiting mis-speculation rollback by identifying recovery regions. OptFT-BR takes advantage of three properties that help it solve the problem of partial execution rollback. First, it uses recovery regions to greatly increase the number of places it can safely rollback to during an execution. Second, OptFT-BR leverages state equivalence flexibility from the relaxed equivalent results constraint, and, finally, it leverages analysis specific properties to help identify recovery regions.

Throughout the remainder of this chapter we present and discuss OptFT-BR, an optimistic hybrid data-race detection that bounds rollbacks by identifying rollback recovery regions within executions. Overall OptFT-BR is highly effective at optimizing data-race detection, showing on average 50% speedups over traditional hybrid analysis, while reducing expected rollback duration of OptFT by on average 67.6%.

5.1 Recovery Region Detection

Recovery regions are the key property of programs which enable OptFT-BR to identify rollback points within executions. Throughout this section we discuss how OptFT-BR leverages analysis specificity, and analysis state equivalence to identify recovery regions for data-race detection. We first overview sufficient background on happens-before data-race detection, and then discuss the properties of recovery regions and how they are dynamically identified.

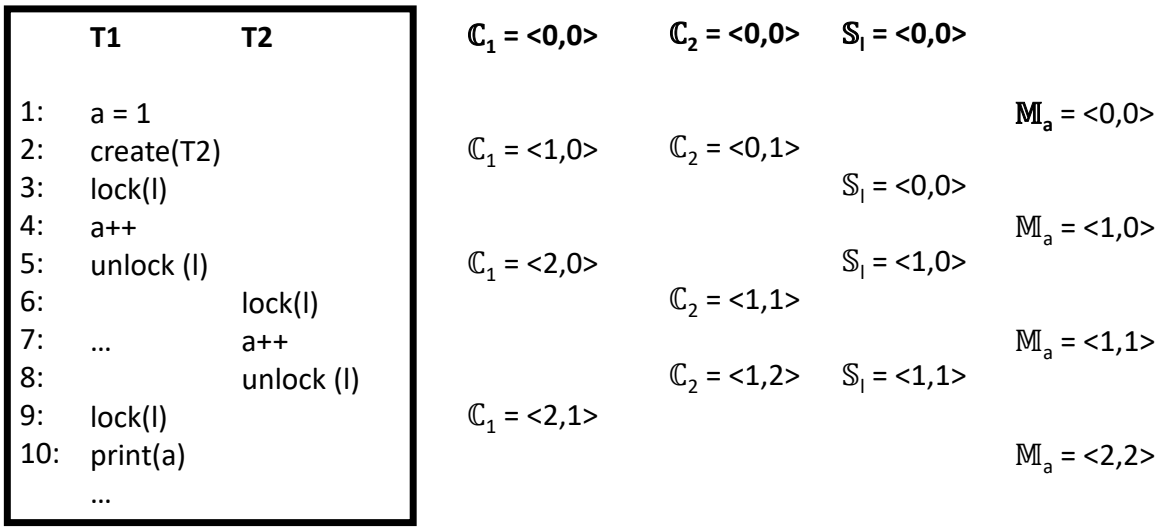


Figure 5.2: An illustration of vector clock propagation. Vector clocks for threads ($\mathbb{C}_1$ and $\mathbb{C}_2$) are updated on synchronization operations. Lock operations cause the thread's vector clock to union with the lock's vector clock ($\mathbb{C} \sqcup \mathbb{S}_l$). Memory accesses adjust the memory's vector clock ($\mathbb{M}_a$) based on the vector clock of the accessing thread.

5.1.1 Vector Clock Race Detection Background

Before discussing how we create bounded rollback for data-race detection, we first review the basics of vector-clock race detection. Vector clocks maintain a partially ordered clock, mapping each thread in the system to that thread's logical time. Vector clocks are used to keep partial ordering of events and define three operations: $\sqsubseteq$ (happens before), $\sqcup$ (join), and $inc_t(V)$ (increment).

$$
\begin{aligned}
V(t) &\in \mathbb{N} \\
V_1 \sqsubseteq V_2 &\iff \forall t.\, V_1(t) \leq V_2(t) \\
V_1 \sqcup V_2 &= \lambda t.\, max(V_1(t), V_2(t)) \\
inc_u(V) &= \lambda t.\, if\ t\ = u\ then\ V(t)+1\ else\ V(t)
\end{aligned}
\tag{5.1}
$$

In a standard vector clock based race detector, each thread t keeps a vector clock ($\mathbb{C}_t$) where for each thread u the vector clock element $\mathbb{C}_t(u)$ denotes the last operation in u that happened before the current state in t. Additionally the algorithm maintains vector clocks

for each synchronization object (e.g. lock). These clocks are updated on synchronization operations to communicate happens-before relations between threads. E.g. a lock l will have a vector clock associated with it $\mathbb{S}_l$. When l is locked, the locking thread's vector clock $\mathbb{C}_t$ is ordered by the lock ($\mathbb{S}_l \sqcup \mathbb{C}_t$) as shown in Figure 5.2 statements 6 and 9. Additionally on unlock the thread's vector clock is incremented ($inc_t(\mathbb{C}_t)$), and the lock's vector clock is updated to order future locking operations by the last thread that locked it ($\mathbb{S}_l \sqcup \mathbb{C}_t$) as shown in Figure 5.2 statements 5 and 8.

All memory locations (m) have two vector clocks, a read ($\mathbb{M}_r$) and write ($\mathbb{M}_w$) vector clock. When a memory location is read from or written to, the data-race detector first checks to ensure that the current memory location access is ordered with respect to its prior accesses ($\mathbb{M}_{r\ or\ w} \sqsubseteq \mathbb{C}_t$), and then updates the variable's read or write vector clock to represent the latest ordering information for that memory access ($\mathbb{C}_t \sqcup \mathbb{M}_{r\ or\ w}$).

5.1.2 Recovery Regions in Data-Race Detection

Finding recovery regions for data-race detection is on its surface a challenging problem, but it can be greatly aided by our analysis specificity and equivalent analysis state optimizations. Without these optimizations, the analysis must identify when all memory address' vector clocks ($\mathbb{M}$) are identical between an optimistic and traditional hybrid analysis. This would require monitoring accesses of both an optimistic and traditional hybrid analysis over all memory addresses, an operation more expensive than the original FastTrack algorithm!

To alleviate this burden, we leverage the notion of state equivalence. OptFT-BR only needs to guarantee that it provides identical analysis results as a traditional data-race detector. These results are provided by the happens before check on memory addresses ($\mathbb{M} \sqsubseteq \mathbb{C}_t$). If we can guarantee all accesses either have equivalent memory address vector clocks, or will not produce a race after recovering from rollback, then we can guarantee OptFT-BR provides equivalent results to a traditional hybrid race detector.

Fortunately, data-race detection has a key property that enables OptFT-BR to detect this efficiently. Once thread t_1 synchronizes with t_2 ($\mathbb{C}_{t_2} \sqsubseteq \mathbb{C}_{t_1}$), all memory access to t_1 cannot race with t_2. This can be further extended. If t_2 additionally synchronizes with t_1 then, any operations that happen before the first synchronization between t_1 and t_2 cannot race. Logically, if all threads synchronize with each other, then no memory accesses before the beginning of their synchronization can race with any memory afterwards. Synchronizations like this are unlikely to happen at a single point in execution, so we leverage the idea of an recovery region to allow a more general form of recovery.

If we define our recovery region to be the region of execution between a set of synchronization operations that synchronizes all threads, this leaves us with a logically equivalent analysis state at the end of the recovery region. Any accesses before the region begins are guaranteed race free, and the re-executing the region during rollback will reconstruct vector clock states for memory accesses during the recovery region.

This definition of recovery regions is more formally specified by the vector clocks of the threads in the program. At any given time during the program execution (r_b), there exists a vector clock $\mathbb{R}$ such that if all thread's vector clocks are ordered after $\mathbb{R}$ ($\forall t.\ \mathbb{C}_t \sqsubseteq \mathbb{R}$) then all memory accesses before r_b are ordered with respect to $\mathbb{R}$ ($\mathbb{M} \sqsubseteq \mathbb{R}$). The intuition behind this is that every memory access's vector clock ($\mathbb{M}$) is by definition ordered with respect to the last thread to access that memory ($\mathbb{M} \sqsubseteq \mathbb{C}_t$). $\mathbb{C}_t$ by definition happens before $\mathbb{R}$, so transitively $\mathbb{M} \sqsubseteq \mathbb{R}$. Finding $\mathbb{R}$ at any point in program time is trivial, by taking the union all vector clocks for all threads. For example, in Figure 5.2 at statement 2, $\mathbb{R}$ is $< 1,1 >$, and at statement 8 $\mathbb{R}$ is $< 2,2 >$.

More concisely:

$$\begin{aligned} \mathbb{R} &= \bigsqcup_t \mathbb{C}_t \\ \forall m.\ \mathbb{M} &\sqsubseteq \mathbb{C}_t \\ \forall t.\ \mathbb{C}_t &\sqsubseteq \mathbb{R} \\ \forall m.\ \mathbb{M} &\sqsubseteq \mathbb{R} \end{aligned} \tag{5.2}$$

Once all thread's vector clocks have advanced so they are ordered after $\mathbb{R}$, then any memory accesses before r_b can no longer contribute to program data races. We say r_b is the recovery region's beginning, and we refer to the point in time when all thread's are ordered after $\mathbb{R}$ ($\mathbb{R} \sqsubseteq \mathbb{C}_t$) as the recovery region's end.

Additionally, when the recovery region is re-rexecuted on rollback, any memory accesses during the recovery region will update the vector clocks for that memory region ($\mathbb{M} \sqcup \mathbb{C}_t$). The result is that after rolling back and replaying the recovery region, OptFT-BR guarantees that any accesses before the recovery region do not effect the data-race detector output after the recovery region, and any accesses during the recovery region will have identical state to that of a conservative analysis. With these guarantees OptFT-BR provides equivalent data-race detector results to a fully conservative analysis.

5.2 OptFT-BR

OptFT-BR implements our optimistic hybrid FastTrack algorithm, with partial-rollback support. It builds on-top of OptFT, described in Chapter IV, and uses the data-race specific recovery regions discussed in Section 5.1 to enable partial rollback. This section discusses some of the challenges and design decisions of building OptFT-BR.

OptFT-BR is built entirely at the Java bytecode layer, using the ASM5 Java bytecode library [24] for instrumentation, and the Chord analysis framework for static analysis [85]. OptFT-BR uses slightly modified versions of the instrumentation passes from

FastTrack [48] to provide race detection instrumentation.

To support partial rollback, OptFT-BR runs a slightly modified variant of OptFT, altering some behaviors which make rollbacks challenging. OptFT-BR also detects rollback recovery regions, creates rollback points, and on mis-speculation properly handles rollback behaviors. We now explain each of these components in more detail.

5.2.1 Detecting Recovery Regions and Creating Checkpoints

OptFT-BR monitors for completed recovery regions by first identifying $\mathbb{R}$ at the recovery region's beginning. $\mathbb{R}$ is constructed by unioning all thread's vector clocks, as shown in equation 5.2. On each synchronization operation, OptFT-BR checks if the thread is ordered by $\mathbb{R}$ ($\mathbb{R} \sqsubseteq \mathbb{C}_t$). Once $\mathbb{R}$ is found to happen before all threads, OptFT-BR knows that the recovery region's beginning (r_b) is a safe rollback point.

As OptFT-BR monitors every synchronization operation to detect recovery region ends, it must disable the no custom synchronization likely invariant used by OptFT. No custom synchronizations assumes that all synchronization is done through specified synchronization functions. This assumption allows it to skip monitoring locks around operations which do not protect any potentially racy accesses. As OptFT-BR must maintain accurate thread vector clocks for recovery region detection, we disable this optimization. Overall this optimization has a relatively small impact on final runtime performance.

As described in Section 5.1, potential recovery regions exist at all points in the program's execution. However, recovery regions are only candidates for rollback once the recovery region has completed ($\forall t.\ \mathbb{R} \sqsubseteq \mathbb{C}_t$), and monitoring a recovery region requires a vector-clock comparison on each synchronization operation, a relatively expensive operation. Further, to enable rollback, a checkpoint must be taken at the beginning of any recovery region OptFT-BR may rollback to, potentially resulting in high overheads for the many available recovery regions. These costs make it impractical for OptFT-BR to monitor all possible recovery regions in the execution.

Since considering all recovery regions is impractical, OptFT-BR only considers one recovery region at a time. At the beginning of the program, OptFT-BR checkpoints the program state, and constructs a vector clock representing the recovery region ($\mathbb{R}$), and monitors every synchronization call, waiting for the recovery region to end. Once the recovery region end is detected, OptFT-BR creates a new recovery region vector clock ($\mathbb{R}$) by unioning all thread's vector clocks, and repeats the checking process. When a rollback is needed, OptFT-BR reloads the checkpoint from the last completed recovery region, switches the analysis to run a conservative dynamic race detector, and replays any inputs until the execution reaches the point of invariant violation, at which point it begins to run on live inputs. In some cases (e.g. single threaded execution), recovery regions are created very rapidly, so OptFT-BR also rate-limits recovery region creation to once every 2 seconds to remove excessive check-pointing overheads. As rollbacks are very rare, the benefits in performance from monitoring fewer recovery regions likely greatly outweigh costs in rollback overhead.

5.2.2 Handling Mis-Speculations

OptFT-BR handles mis-speculations by rolling back to the beginning of the last completed recovery region, and resuming the program after that point. It uses a deterministic record and replay system to guarantee the rolled-back execution reaches the same state of the original execution after rollback.

Upon rollback OptFT-BR must transition from its fast-path optimistic hybrid analysis checks to slow-path conservative analysis checks. Ideally, this would be done by re-instrumenting the code at mis-speculation time to provide slow-path checks. As mis-speculations are rare, the one-time re-instrumentation cost could introduce a near minimum runtime overhead. Unfortunately, the Java Virtual Machine (JVM) does not support live function bytecode re-instrumentation. To work around this limitation, OptFT-BR statically creates two versions of each function, a slow-path version, and fast-path version. If a rollback occurs, it switches the execution to the exact same point in the slow-path codebase.

Due to the limitations of the JVM, this slow-path switch is more complex and costly than ideal. Ideally, the JVM would allow seamless transitions between slow-path functions and the fast-path code; however, such a transition would require the mis-speculation bytecode ensure all returns went to slow-path functions, requiring the bytecode to modify the return address from each function call on the callstack. As the JVM disallows this behavior, OptFT-BR instead creates an additional copy of the slowpath code at the end of each fast-path function, and on return from each function call the fast-path checks to see if it must branch to slow-path code. This adds the cost of a global variable check and (highly predictable) conditional jump at the end of each function call on the fast-path. This overhead is generally acceptable, however there are some function call patterns which can cause OptFT-BR's slow-path switching code to cause considerable overheads.

After these modifications, OptFT-BR constructs an optimistic hybrid race detector with partial rollback support, and results equivalent to those of the FastTrack race detector.

5.3 Evaluation

In this section we show that OptFT-BR successfully bounds rollbacks for optimistic hybrid data-race detection for many applications, while still providing performance improvements comparable to OptFT in rollback-free executions.

5.3.1 Experimental Setup

To evaluate the overhead of OptFT-BR we run the same benchmark suite as we did in Chapter IV for OptFT. This benchmark suite consists of all of the benchmarks from the JavaGrande and Dacapo benchmark suites which are multi-threaded, and work with our underlying frameworks (chord and RoadRunner). We also use the same training and testing sets used by the original OptFT work.

We begin by learning likely invariants for use in OptFT-BR. We incrementally run tests in our training set until additional training runs provide no new invariants. We then apply

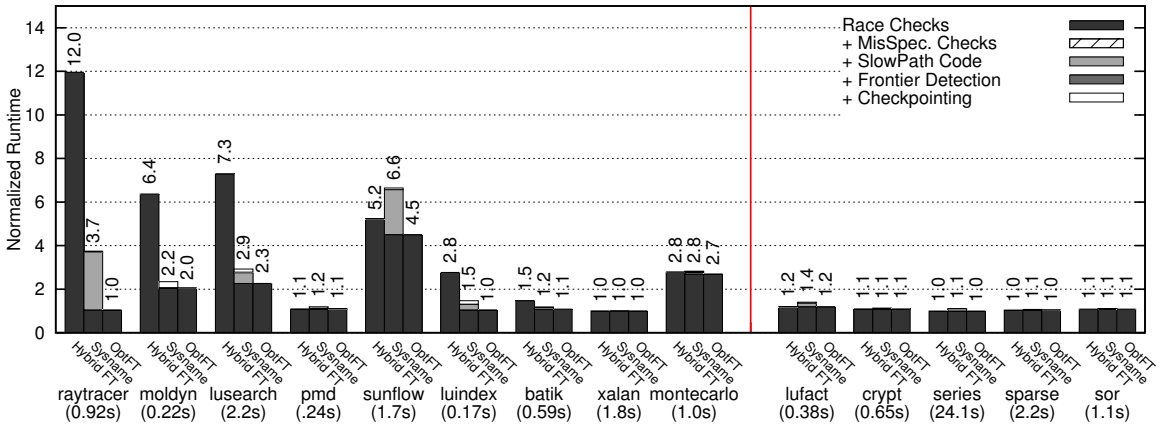


Figure 5.3: Normalized runtimes for OptFT. Baseline runtimes for each benchmark are shown in parentheses. Tests right of the red line are proven race-free by sound static race detection, but included here for completeness.

these invariants towards optimistic hybrid analysis.

Once the optimistic hybrid analysis is constructed, we run all of the testing inputs over the analysis bytecode. All experiments are conducted using 8 cores of an Intel Xeon E5-2687W v3 3.1 GHz processor.

The major difference between the setup of OptFT's evaluation and OptFT-BR's is that OptFT-BR uses a static bytecode instrumentation library, where the vanilla RoadRunner framework used by OptFT uses dynamic instrumentation. This switch to static tooling eliminates the start-up and framework overheads reported by OptFT.

5.3.2 Dynamic Overhead

Figure 5.3 shows the overall runtime overhead of OptFT-BR in comparison with a traditional dynamic analysis, traditional hybrid analysis, and OptFT. Overall OptFT-BR shows considerable speedups versus a conservative hybrid analysis with average speedups of 50%. While, OptFT-BR adds a non-trivial overhead (40.7%) to OptFT, it provides bounded roll-back recovery, making it much more suitable for analyzing long-running executions, or production scenarios. We also note that most of this overhead is caused by inefficiencies in the implementation of the slow-path transition code-paths (explained in Section 5.2.2), and is consequently not fundamental to OptFT-BR. If these inefficiencies were resolved,

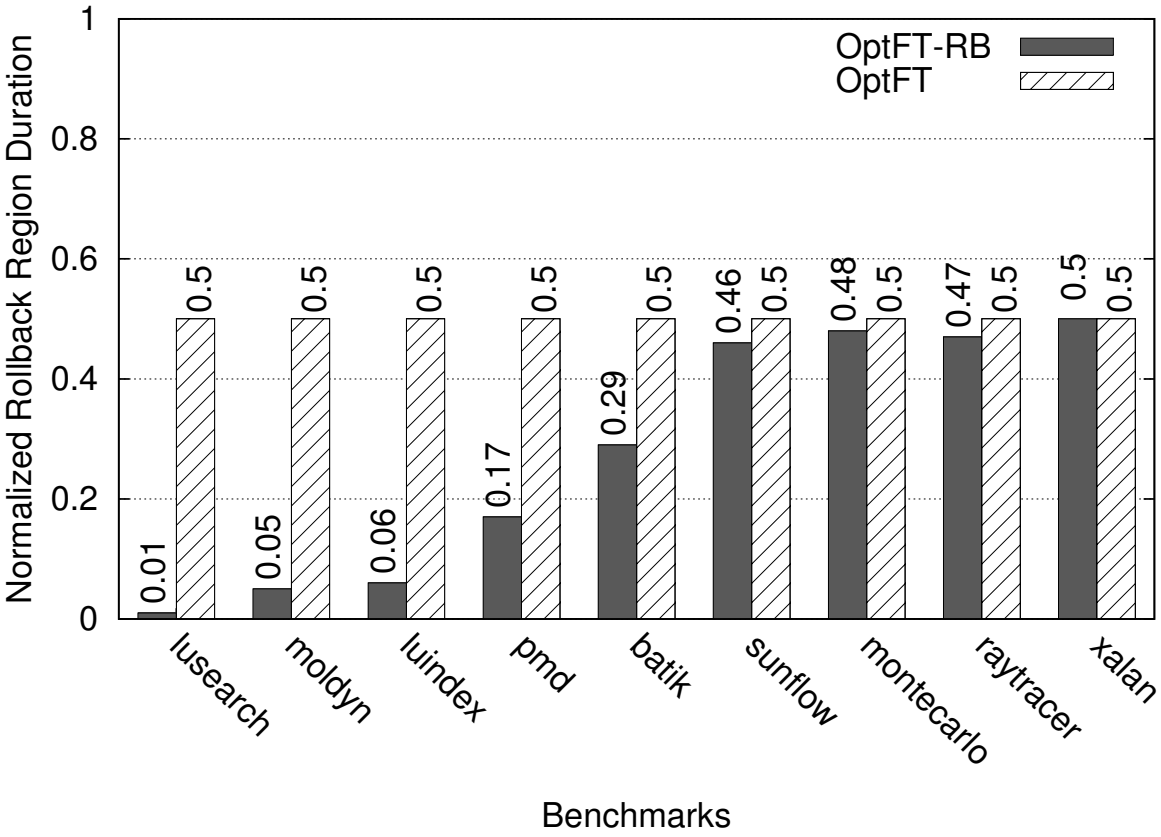
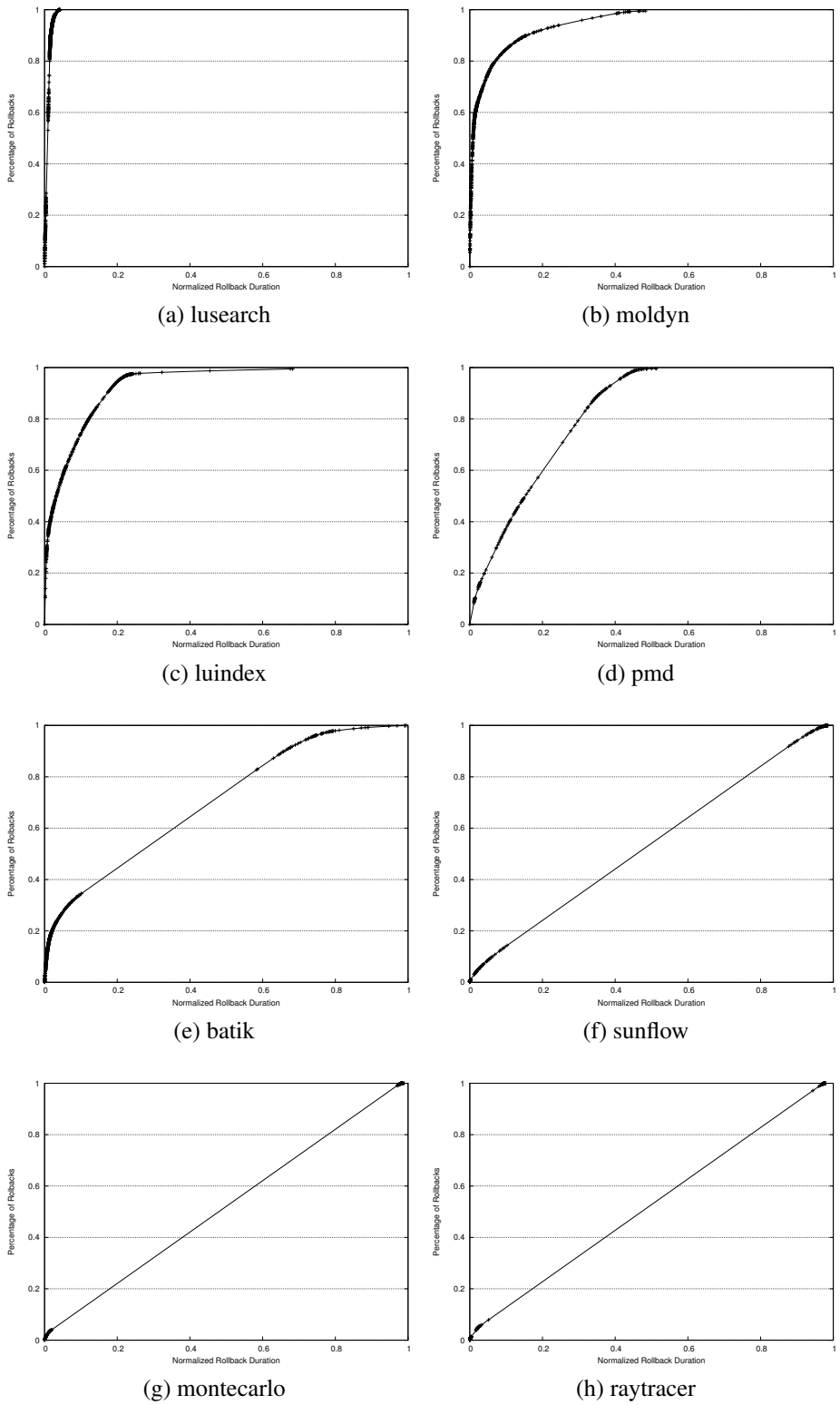


Figure 5.4: Expected rollback time for OptFT-BR versus OptFT. OptFT-BR dramatically decreases rollback recovery times for many benchmarks.

for example with JVM level instrumentation, OptFT-BR should expect overheads closer to 10%.

The second component of OptFT-BR we evaluate is the frequency, duration, and expected effect of rollback bounds. We begin by looking at the expected amount of execution an analysis will rollback on mis-speculation. Figure 5.4 shows the expected rollback duration of a rollback with OptFT-BR versus a rollback with OptFT. OptFT-BR shows an average reduction in rollback time of 67.6%. While this reduction is significant, the results are largely bimodal. Several benchmarks, such as lusearch, moldyn, and luindex are significantly aided with OptFT-BR, while several benchmarks, such as raytracer and xalan do not show frequent enough recovery regions for OptFT-BR to significantly reduce rollback duration.

Figure 5.5 gives a visual breakdown of the CDF of expected average rollback durations



(a) lusearch (b) moldyn

(c) luindex (d) pmd

(e) batik (f) sunflow

(g) montecarlo (h) raytracer

Figure 5.5: CDF of expected rollback duration normalized to total execution time. Rollbacks assumed to be evenly spaced over program duration. Xalan is excluded, as no recovery regions are found in the benchmark, making rollback results equivalent to OptFT.

for each benchmark measured with OptFT-BR. These graphs show the maximum potential rollback duration, represented by the point with the highest "x" axis point for each benchmark. These graphs additionally demonstrate a breakdown of recovery region frequency and duration, outlining how different benchmarks discover recovery regions, and how effective OptFT-BR is at reducing rollbacks.

5.4 Conclusion

This chapter has identified discussed and identified the challenges constructing an OHA which bounds rollback duration on mis-speculation. Although this is a challenging problem to solve for generic analyses, recovery regions, along with analysis-specific observations make rollback practical and effective for data-race detection. Using these optimizations I have presented OptFT-BR, demonstrating that bounded rollbacks are possible, at least for data-race detection. I believe some of these techniques can be applied to create OHAs with bounded rollback for other analyses.

CHAPTER VI

Conclusion and Future Works

In this thesis I have presented eidetic systems, and optimistic hybrid analyses. These two systems together make it practical to run retrospective analysis over entire computer systems. Eidetic systems, such as Arnold, make information recall and query over the entire history of a single machine practical. Optimistic hybrid analyses, like OptFT and OptSlice, dramatically reduce the burden of these retroactive analyses through a novel method of analysis optimization. Together these techniques allow entire system's worth of data to be efficiently gathered and queried.

While these systems are both useful and powerful as described, they are also both potential platforms to enable further research. Optimistic hybrid analysis in particular has many open problems, and possible applications. My current implementations of optimistic hybrid analysis are limited by their inconsistent execution times. Rollbacks, and slow-path code analyses cause OHA to provide only average-case speedups, and in the worst case can add significant overhead.

Future work in the area of optimistic hybrid analysis includes addressing these worst-case performance overheads. Worst-case overheads in OHA stem from its rollback mechanism. In chapter V I discussed how bounding rollbacks was challenging, and OptFT-BS, one system that opportunistically bounds rollbacks, however OptFT-BS is opportunistic in its rollback bounding, and is still unable to provide a guaranteed bounds for rollbacks. I

would like to explore which analyses can have bounded rollbacks, or how an application can be modified to guarantee bounded rollback points exist within an OHA. In the same vein, with some restrictions to static analysis, it may be possible to construct OHA's which do not require any rollback on mis-speculation.

Even with the rollback problem solved, OHA suffers from unpredictable performance. Currently, OHAs are either in fast-path, or a conservative slow-path execution. I would like to explore methods of creating a more gradual degradation of performance for likely invariant mis-speculations within an OHA, for instance by creating multiple rollback paths depending on which invariants fail.

I have also only begun exploring the uses of optimistic hybrid analysis. While it has shown to have great benefit on slicing and data-race detection analyses, I believe OHA could dramatically benefit many other useful analyses. Consider taint-tracking. Taint-tracking is used heavily today for security and forensic queries [44, 11], but often creates unacceptable overheads. An optimistic hybrid taint-tracking algorithm could greatly benefit these overheads. Furthermore, OHA could apply very well to enforcing safety properties of programs, such as region serializability, or RS. RS is considered the gold-standard of memory models [79], but too expensive to enforce generally. The expense in enforcing RS is that of avoiding memory conflicts, caused by data-race detection. OHA has already been shown to significantly reduce data-race detection. However, as RS only requires conflict freedom, and not general data-race freedom, an optimistic hybrid RS enforcement algorithm could likely mitigate nearly all mis-speculation recovery costs, making it highly practical.

This thesis has shown how eidetic systems and optimistic hybrid analysis make it practical to retroactively analyze entire systems of computations. These two systems additionally demonstrate a mutualistic relationship. I now argue retroactive dynamic program analyses, also work to further enable eidetic systems to record and replay systems efficiently. Eidetic systems, such as Arnold, require programs to be free of data-race bugs in order to efficiently record programs. Retroactive analyses, such as OptFT present an efficient way to ensure all

executions of a program are data-race free. By recording executions for later analysis eidetic systems enable dynamic retroactive analyses to identify and help programmers reduce the bugs which make eidetic systems less efficient.

By applying these techniques to today's problems, and building upon them in the future we can greatly reduce the burdens of users, programmers, and system administrators in constructing, maintaining, and understanding complex software systems.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Linux Desktop Testing Project. `http://ldtp.freedesktop.org`.

[2] VimGolf. `http://vimgolf.com`, 2016. Accessed: 2016-07-31.

[3] Project Gutenbveg. (n.d.). `http://www.gutenberg.org`, 2017. Accessed: 2017-04-12.

[4] SvgCuts. `http://svgcuts.com`, 2017. Accessed: 2017-07-28.

[5] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 213–227, New York, NY, USA, 1989. ACM.

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

[7] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 143–151. IEEE, 1995.

[8] Bowen Alpern, Ton Ngo, Jong-Deok Choi, and Manu Sridharan. Dejavu: deterministic java replay debugger for jalapeno java virtual machine. In *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 165–166, New York, NY, USA, 2000.

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

[10] Lars Ole Andersen. Program analysis and specialization for the c programming language. In *PhD thesis, DIKU, University of Copenhagen*, 1994.

[11] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[12] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

[13] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with confaid. *USENIX ;login*, 36(1), February 2011.

[14] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. *SIGPLAN Not.*, 29(6):290–301, June 1994.

[15] D. F. Bacon and S. C. Goldstein. Hardware assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 194–206. ACM Press, 1991.

[16] R. M. Balzer. Exdams: Extendable debugging and monitoring system. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, AFIPS '69 (Spring), pages 567–580, New York, NY, USA, 1969. ACM.

[17] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *ACM SIGPLAN Notices*, volume 38, pages 103–114. ACM, 2003.

[18] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[19] Michael D Bond and Kathryn S McKinley. Probabilistic calling context. In *ACM SIGPLAN Notices*, volume 42, pages 97–112. ACM, 2007.

[20] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th International Conference on Mobile Systems, Applications and Services*, Breckenridge, CO, June 2008.

[21] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 40–45, New York, NY, USA, 1990. ACM.

[22] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 1–11, Copper Mountain, CO, December 1995.

[23] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[24] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.

[25] Michael G Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J Serrano, Vugranam C Sreedhar, Harini Srinivasan, and John Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141. ACM, 1999.

[26] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 259–269, Washington, DC, USA, 1997. IEEE Computer Society.

[27] Robert Cartwright and Mike Fagan. Soft typing. *SIGPLAN Not.*, 26(6):278–292, May 1991.

[28] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 39–50, New York, NY, USA, 2008. ACM.

[29] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[30] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

[31] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 1–14, June 2008.

[32] Michael Chow. *Scaling Causality Analysis for Production Systems.* PhD thesis, 2016.

[33] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *In Proceedings of the International Symposium on Software Testing and Analysis*, pages 196–206, July 2007.

[34] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Softw. Eng.*, 35(5):684–702, September 2009.

[35] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings*

of the 7th Conference on USENIX Security Symposium - Volume 7, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[36] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):8:1–8:37, May 2008.

[37] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 118–128, New York, NY, USA, 2007. ACM.

[38] Bruno Dufour, Barbara G. Ryder, and Gary 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*, SIGSOFT '08/FSE-16, pages 59–70, New York, NY, USA, 2008. ACM.

[39] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, MA, December 2002.

[40] George W. Dunlap, Dominic G. Lucchetti, Michael Fetterman, and Peter M. Chen. Execution replay on multiprocessor virtual machines. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 121–130, March 2008.

[41] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, March 2008.

[42] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 245–255, 2007.

[43] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[44] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

[45] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In

Proceedings of the 21st International Conference on Software Engineering, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[46] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007.

[47] Stuart I. Feldman and Channing B. Brown. IGOR: A system for program debugging via reversible execution. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123, 1988.

[48] Cormac Flanagan and Stephen Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, pages 121–133, Dublin, Ireland, June 2009.

[49] Cormac Flanagan and Stephen N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 1–8, New York, NY, USA, 2010. ACM.

[50] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. In *Proceedings of the 2005 Symposium on Operating Systems Principles*, October 2005.

[51] Jim Gray. Why do computer stop and what can be done about it? Technical Report 85.7, Tandem Corp., June 1985.

[52] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 263–272. ACM, 2005.

[53] Rajiv Gupta, Mary Lou Soffa, and John Howard. Hybrid slicing: Integrating dynamic information with static analysis. *ACM Trans. Softw. Eng. Methodol.*, 6(4):370–397, October 1997.

[54] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 214–236, Berlin, Heidelberg, 2003. Springer-Verlag.

[55] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, May 2002.

[56] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *ACM SIGPLAN Notices*, volume 42, pages 290–299. ACM, 2007.

[57] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *International Static Analysis Symposium*, pages 265–280. Springer, 2007.

[58] Fritz Henglein. Global tagging optimization by type inference. *SIGPLAN Lisp Pointers*, V(1):205–215, January 1992.

[59] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 265–276, June 2008.

[60] Richard W M Jones, Paul H J Kelly, Most C, and Uncaught Errors. Backwards-compatible bounds checking for arrays and pointers in c programs. In *in Distributed Enterprise Applications. HP Labs Tech Report*, pages 255–283, 1997.

[61] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. Privacy oracle: A system for finding application leaks with black box differential testing. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 279–288, New York, NY, USA, 2008. ACM.

[62] Edward Kaiser, Wu-chang Feng, and Travis Schluessler. Fides: Remote anomaly-based cheat detection using client emulation. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 269–279, New York, NY, USA, 2009. ACM.

[63] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Efficient patch-based auditing for Web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[64] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

[65] Johannes Kinder and Dmitry Kravchenko. Alternating control flow reconstruction. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 267–282, Berlin, Heidelberg, 2012. Springer-Verlag.

[66] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 223–236, Bolton Landing, NY, October 2003.

[67] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. Technical Report CSE-TR-495-04, University of Michigan, August 2004.

[68] O. Laadan, R. Baratto, D. Phung, S. Potter, and J. Nieh. DejaView: A personal virtual computer recorder. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 279–292, Stevenson, WA, Oct 2007.

[69] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: Hybrid program analysis for determinism. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation*, Beijing, China, June 2012.

[70] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 25–33, New York, NY, USA, 2006. ACM.

[71] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, pages 101–110, New York, NY, USA, 2009. ACM.

[72] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[73] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.

[74] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[75] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 277–288, Beijing, China, 2008.

[76] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[77] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations*

of Software Engineering, ESEC/FSE 2015, pages 462–473, New York, NY, USA, 2015. ACM.

[78] JP Mangalindan. Facebook users dial 911 over social network outage. `http://fortune.com/2014/08/01/facebook-users-dial-911-over-social-network-outage/`, August 2014.

[79] Daniel Marino, Todd Millstein, Madanlal Musuvathi, Satish Narayanasamy, and Abhayendra Singh. The Silently Shifting Semicolon. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 177–189, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[80] Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 71–80, New York, NY, USA, 2002. ACM.

[81] Markus Mock, Manuvir Das, Craig Chambers, and Susan 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*, pages 66–72. ACM, 2001.

[82] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 289–300, June 2008.

[83] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 228–241, San Antonio, TX, January 1999.

[84] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. *SIGPLAN Not.*, 45(8):31–40, June 2010.

[85] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[86] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using Strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, October 2006.

[87] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. *SIGPLAN Not.*, 37(1):128–139, January 2002.

[88] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007.

[89] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[90] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking:. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 11–20, New York, NY, USA, 2002. ACM.

[91] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution for parallel programs. *SIGPLAN Not.*, 24(1):124–129, November 1988.

[92] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 177–191, October 2009.

[93] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPOPP03*, pages 179–190, San Diego, CA, June 2003.

[94] Dustin Rhodes, Cormac Flanagan, and Stephen 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, pages 141–156, New York, NY, USA, 2017. ACM.

[95] Michiel Ronsse and Koen De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[96] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 258–266, 1996.

[97] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. *ACM SIGPLAN Notices*, 48(4):139–152, 2013.

[98] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[99] Donna Scott. Assessing the costs of application downtime. Technical report, 1998.

[100] Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D Bond, and Milind Kulkarni. Hybrid static–dynamic analysis for statically bounded region serializability. In *ACM SIGPLAN Notices*, volume 50, pages 561–575. ACM, 2015.

[101] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, December 2009.

[102] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 8–8, New York, NY, USA, 2001. ACM.

[103] Sudarshan Srinivasan, Christopher Andrews, Srikanth Kandula, and Yuanyuan Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 29–44, Boston, MA, June 2004.

[104] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 2–13, February 1998.

[105] Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[106] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, October 2011.

[107] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Long Beach, CA, March 2011.

[108] Evangelos Vlachos, Michelle L. Goodstein, Michael A. Kozuch, Shimin Chen, Babak Falsafi, Phillip B. Gibbons, and Todd C. Mowry. ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–284, Pittsburgh, PA, March 2010.

[109] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214, Dubrovnik, Croatia, 2007.

[110] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 336–346, New York, NY, USA, 2013. ACM.

[111] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[112] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 122–135, June 2003.

[113] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. TightLip: Keeping applications from spilling the beans. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, pages 159–172, Cambridge, MA, April 2007.

[114] Jianwen Zhu. Towards scalable flow and context sensitive pointer analysis. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 831–836, June 2005.