Runtime Verification of Kernel-Level Concurrency Using Compiler-Based Instrumentation

A Dissertation Presented

by

Justin Seyster

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2012

Abstract of the Dissertation

Runtime Verification of Kernel-Level Concurrency Using Compiler-Based Instrumentation

by

Justin Seyster

Doctor of Philosophy

in

Computer Science

Stony Brook University

2012

To approach the challenge of exploiting the performance potential of multi-core architectures, researchers and developers need systems that provide a reliable multi-threaded environment: every component of the underlying systems software must be designed for concurrent execution. But concurrency errors are difficult to diagnose with traditional debugging tools and, as not all schedules trigger them, can slip past even thorough testing.

Runtime verification is a powerful technique for finding concurrency errors. Existing runtime verification tools can check potent concurrency properties, like atomicity, but have not been applied at the operating system level. This work explores runtime verification in the systems space, addressing the need for efficient instrumentation and overhead control in the kernel, where performance is paramount.

Runtime verification can speculate on alternate schedules to discover potential property violations that do not occur in a test execution. Non-speculative approaches detect only violations that actually occur, but they are less prone to false positives and are computationally faster, making them well suited to online analysis.

Offline monitoring is suited to more types of analysis, because speed is less of a concern, but is limited by the space needs of large execution logs, whereas online monitors, which do not store logs, can monitor longer runs and thus more code. All approaches benefit from the ability to target specific system components, so that developers can focus debugging efforts on their own code. We consider several concurrency properties: data-race freedom, atomicity, and memory model correctness.

Our Redflag logging system uses GCC plug-ins we designed to efficiently log memory accesses and synchronization operations in targeted subsystems. We have developed data race and atomicity checkers for analyzing the resulting logs, and we have tuned them to recognize synchronization patterns found in systems code.

Additionally, our Adaptive Runtime Verification framework (ARV) provides monitoring for concurrency properties when overhead constraints make it impractical to monitor every event in the system. Even with this incomplete knowledge, ARV can estimate the state of the monitored system and dynamically reassign monitoring resources to objects that are most likely to encounter errors.

To Mom and Dad.

Contents

List of Figures

List of Tables

Acknowledgments

If there is one person who is to blame for the extension of my stay here at Stony Brook to over a decade, that person is Mike Gorbovitski. It was Mike who, in 2005, convinced a young undergraduate preparing for his next steps that a master's degree would be essentially a half measure and that he should actually be applying directly to Ph. D. programs, as Mike had done himself. Not long after, George Hart, who was then a professor at Stony Brook, found this same undergraduate wandering the department looking for advice on how to do just that. He immediately found the right people to talk to, inquiring on how an "exceptional undergraduate student" might apply for the department's doctorate program. With help like that, getting accepted to the program was the easiest part of this whole process.

Sean Callanan told me that he was drawn to the FSL because of the people here. He and two other good friends, Dave Quiqley and Rick Spillane, similarly motivated me. They and all the other people who work and have worked at the FSL make it one of the most engaging places to be at this university. Among their ranks are the master's students who have worked on projects with me: Abhinav Duggal, Prabakar Radhakrishnan, Ketan Dixit (though he is from another lab), Samriti Katoch, Siddhi Tadpatrikar, Mandar Joshi, and Atul Karmarkar. Their sweat is in here too.

As with most research, this work is a collaboration with many people. In addition to the fellow students listed above, Ezio Bartocci, Klaus Havelund, Scott Smolka, Scott Stoller, Radu Grosu, and my advisor, Erez Zadok are behind the research presented here. Being part of a close-knit research group from the very beginning of my career here made the process of learning about scientific research much smoother.

I think that all of us recognize what an uncommon advisor we have in Professor Zadok. I learned from his example what *constructive criticism* is: rarely does he have anything to say on what is bad about your work, but he has plenty of advice about what would make it *better*.

My roommate, Tal Eidelberg, has made my grad career go by much faster than it otherwise would have. Weekends might have gotten boring if not for late night hot wings, ski trips, probably more video games than I should admit in this document, and even a trip to Vegas. On top of that, he taught me to drive stick and hired me for my first internship.

With all the support that my parents have given me, I should be writing this from the Oval Office. More than anybody, I really have them to thank for everything. My mom made me promise that if I ever won an Oscar that I would name her in my acceptance speech. Mom, I think this is about as close as you are going to get.

This work was made possible in part thanks to a U.S. Air-Force research grant number FA0550-09-1-0481.

Chapter 1

Introduction

For the software industry, the promise of runtime verification is the power to find programming defects in testing before they become faults in production. Verification tools can screen for these errors by checking for out-of-bounds accesses, leaked memory, unsanitized user input, and other property violations. With techniques like these already taking an important place in developers' toolboxes, runtime verification holds great potential for tackling the subtle issues of concurrent software development.

Concurrency errors are an important target for verification because they are so difficult to find with testing alone. Even tests that exercise all the code paths involved in an error will not expose the error unless they run with a triggering schedule. Bugs that do appear in testing can appear randomly during each run, frustrating debugging efforts. But a reported property violation from a runtime verification tool can point to exactly where the problem is.

The Linux community has already adopted runtime verification for checking the correctness of its concurrency. Lockdep is a powerful tool for checking lock ordering to ensure that test runs are free from potential deadlocks [47].

Concurrency Errors

Concurrency errors occur when parallel threads of execution access shared data structures simultaneously. Without careful synchronization, these threads will step on each other's toes, tripping into inconsistent states and eventually crashing or, worse, producing corrupted results.

Data races A *data race* is the simplest example of this kind of error because it involves simultaneous accesses to a single variable. In a data race, one thread tries to write a variable while another thread is also accessing it. On some architectures, just this pair of accesses is dangerous per se. For example, on 32-bit x86, when two threads write to the same 64-bit variable, it may get half of its value from each thread, a state that is inconsistent for both threads.

More commonly, a data race is part of a bad interleaving involving several accesses in each thread. A simple increment operation involves two accesses: the first to read value i, the second to write value $i + 1$. When two threads increment a variable at the same time, the write operation in the first thread can form a data race with the second thread's read. Depending on who "wins" the race, the second thread will observe either i or $i + 1$ as the value to increment. In the former case, the second thread will also write $i + 1$, and the two threads together will only succeed in incrementing the variable once, an error known as a *lost update*.

The *Lockset* algorithm [20,57] checks for data races by verifying each variable's *lock discipline*. A lapse in lock discipline, meaning a variable that is not consistently protected by some lock, means a potential data race. We have implemented Lockset for the Linux kernel, as discussed in Section 2.1.2

Data races do not correspond precisely with concurrency errors, however. Not all races lead to an error, and in systems especially, developers design code that can tolerate data races rather than accepting the cost of locking every access. More importantly, even programs that are free of data races may have concurrency errors.

Atomicity Checking for atomicity is a more direct way to observe unintended effects from parallelization. Two regions of code are *atomic* with respect to each other if, when executed concurrently, they produce the same result as if they executed one after the other. Clearly, the racy increment discussed above does not satisfy this property: a pair of atomic increments will add two to a variable whether executed in parallel or sequentially.

The block-based algorithms [67, 68] check a program execution for potential schedules that would violate the atomicity property, leading to possible concurrency errors. Section 2.1.3 discusses these algorithms, which we also implemented for the Linux kernel.

Memory Model Errors Developers often expect sequentially consistent behavior from multicore systems, meaning that all memory accesses in the system follow a canonical linear order, but modern processors do not always provide that guarantee. Processors can reorder memory operations and delay the affects of memory writes, and these changes in order are sometimes visible to other cores accessing the same memory. Modern systems do guarantee sequential consistency for programs free of data races, but that is not sufficient for most systems code.

When a memory reordering could negatively affect program execution, a *memory fence* is necessary to tell the compiler and processor to disallow the dangerous reordering. Finding these buggy reorderings among all the accesses in a large system is a difficult task, however. Interleaving code needs to execute within a very small window to be affected by a reordering, so any errors they cause are difficult to expose.

Offline Verification

We have implemented offline verification for two of the properties discussed above, data race freedom and atomicity, that checks kernel code. Our system, called *Redflag*, can target specific data structures for comprehensive logging and then analyze those logs for concurrency errors.

Redflag uses *compiler-based instrumentation* to log relevant events. We have developed compiler plug-ins that instrument field accesses and lock operations that operate on targeted data structures. Instrumented operations pass details of the operation, such as which object was accessed or locked, directly to our logging system.

Targeting data structures is an important part of our verification strategy because it allows users to choose specific system components to verify. In production systems, developers are responsible for individual subsystems. Monitoring an entire kernel, for example, would produce reports from systems that the user has no interest in and would incur huge overheads.

An offline analysis tool checks the log for property violations. The tool produces a report for each potential violation that includes complete stack traces for each operation involved. For example, when our Lockset implementation observes a possible data race, it outputs the stack trace for each of the two racing memory accesses.

The greatest challenge to using these analysis techniques on systems-level code is avoiding a proliferation of false positives. We found several conventions in kernel code that resulted in false positives in the Lockset and block-based algorithms. We adapted these algorithms to recognize these conventions. In particular our Lexical Object Availability (LOA) analysis determines when a schedule is impossible because of *multi-stage escape*, described in Section 2.1.4.

Online Analysis

Checking program execution at runtime has the advantage that there is no need to store logs, which grow indefinitely for as long as the program continues to run. Though our offline analysis is thorough, it can only verify relatively short runs before logs grow too large. In Section 2.2, we present an atomicity verification algorithm that can run online in kernel context.

Our algorithm maintains a *shadow memory* for each atomic region running in the system. The shadow memory keeps a thread-local picture of how memory looks to the atomic region. At each access, the algorithm checks for a discrepancy between shadow memory and global memory, which would indicate that a remote thread interfered with the accessed variable in a way that violates its atomicity.

Because this approach only recognizes atomicity violations as they occur, it represents another trade-off with the offline block-based algorithms. The block-based algorithms speculate on all possible schedules, allowing them to expose errors that do not actually occur in the test run. However, determining which schedules are possible is a difficult problem: considering schedules that are actually impossible leads to false positives, and techniques to filter out these impossible schedules may filter out some legitimate schedules, instead causing false negatives. The online algorithm cannot detect errors unless they actually occur in an execution, but it will never report an impossible schedule as an error. This trade-off makes sense for online verification because it can verify longer runs, allowing it to observe many more schedules than our log-based approach.

Aspect-Oriented Instrumentation

We have found compiler-assisted instrumentation to be a simple and efficient way to monitor events that are relevant to our verification techniques. During compilation, the compiler constructs detailed type information that we use to target specific data structures for monitoring.

We use GCC for this purpose because its plug-in system gives access to its internal representation, GIMPLE, which includes the type information we need. On finding data structure accesses that are targeted for monitoring, the plug-in can modify the GIMPLE code for the access, inserting efficient instrumentation directly into the program.

Our INTERASPECT framework is an easy-to-use interface for targeting and adding instrumentation based on the ideas of Aspect-Oriented Programming (AOP). In developing instrumentation plug-ins for Redlfag, we found that, although GIMPLE plug-ins are powerful, a lot of work is necessary to correctly transform GIMPLE statements. INTERASPECT makes concrete many of the lessons we learned about implementing these kinds of transformations.

AOP provides a natural way to express code transformations that consist of attaching additional functionality to specific events that occur in the code. The user specifies a *pointcut*, which defines the set of events, as well as *advice*, which defines the additional functionality. The advice is added at each instrumentation site, or *join point*, in the pointcut.

INTERASPECT's API allows for customized instrumentation. An INTERASPECT plug-in can visit each join point, choosing custom parameters to pass to advice based on properties of the join point. The plug-in can also choose a different advice function or elect to leave a join point uninstrumented. We developed an example plug-in, described in Section 3.3.2, for performing integer range analysis that uses customized instrumentation to efficiently link each join point with the range estimate the join point is associated with. Chapter 3 describes the complete INTERASPECT API.

Adaptive Runtime Verification and State Estimation

Finally, we focus on verification for environments in which overhead is the primary consideration. Our tool for verifying lock discipline uses state estimation [61] to design a monitor that can operate effectively even when it cannot observe every event because of overhead control.

State estimation is a technique that uses a system model to estimate the probability that an incompletely monitored execution experienced a property violation. Using a model of the system, state estimation infers what events might have occurred during "gaps," while the monitor was not able to observe events. Overhead control techniques like SMCO [34] cause gaps when they temporarily disable monitoring to meet an overhead target.

Adaptive Runtime Verification (ARV) uses state estimation to allocate monitoring resources to system objects. ARV allows overhead control mechanisms to spend more of their overhead monitoring *critical* objects, those objects which are most likely to cause a property violation. In our implementation, described in Chapter 4, we use hardware resources to fully monitor the most critical objects, even when monitoring is disabled for the rest of the system.

Chapter 2

Analysis of Kernel Concurrency

As the kernel underlies all of a system's concurrency, it is the most important front for eliminating concurrency errors. To design a highly reliable operating system, developers need tools to find concurrency errors before they cause real problems in production systems. Understanding concurrency in the kernel is difficult. Unlike many user-level applications, almost the entire kernel runs in a multi-threaded context, and much of it is written by experts who rely on intricate synchronization techniques.

Static analysis tools like RacerX [24] can check even large systems code bases for potential data races, but they produce moderate to large numbers of false positives. Heuristic rankings of warnings mitigates but does not eliminate this problem. Static analysis tools that check more general concurrency properties, such as atomicity [56] are less scalable and would also produce many false positives for the kernel. In principle, model checkers can verify any property of any system by exhaustive state-space exploration, but in practice, model checkers do not scale to verification of complex properties, such as concurrency properties, for programs as large and complex as typical kernel components.

Runtime analysis is a powerful and flexible approach to detection of concurrency errors. We designed the *Redflag* framework and system with the goal of airlifting this approach to the kernel front lines. Redflag takes its name from stock car and formula racing, where officials signal with a red flag to end a race. Analysis begins with observing a kernel execution via targeted instrumentation. Instrumentation is provided by compiler plug-ins, which target data structures in specific kernel subsystems for observation. The Redflag framework comprises our targeted instrumentation plug-ins and three main verification components:

1. *Fast Kernel Logging* produces a trace of all observed operations on targeted data structures. It reserves an in-memory buffer to log these events, along with corresponding stack traces, with minimal performance overhead.

2. The *Redflag offline analysis* tool performs post-mortem analyses on logs. Offline analysis reduces runtime overhead and allows any number of analysis algorithms to be applied to the logs.

3. The *Redflag online analysis* tool analyzes events during execution, detecting violations as they occur. Online analysis has higher runtime overhead than offline analysis, but can handle long executions that execute too many events to log practically.

Currently, Redflag implements two kinds of offline concurrency analysis: *Lockset* [57] analysis for potential data races and *block-based* [67, 68] analysis for potential atomicity violations. These two types of analyses cover a wide range of errors, from missing locks to complex interleavings involving multiple variables. These analyses detect *potential errors* (in addition to actual errors) in the sense that they report a warning of a potential error in the observed trace if the same kind of error manifests itself in a permutation of the observed trace that is consistent with the synchronization in the observed trace. For example, if one of those permutations contains an unserializable interleaving of two atomic regions, then atomicity analysis reports a potential atomicity violation in the observed trace. For efficiency, the algorithms are designed to check this without explicitly constructing permutations of the observed trace. We developed several enhancements to improve the accuracy of these algorithms, including *Lexical Object Availability* (LOA) analysis, which eliminates *false alarms* (also called *false positives*) caused by sophisticated synchronization during initialization. We also augmented Lockset to support Read-Copy-Update (RCU) [44] synchronization, a synchronization pattern used in the Linux kernel.

Redflag currently implements one online analysis, for atomicity checking, based on the algorithm used in AVIO [42] and the block-based algorithms [67, 68]. Our online atomicity analysis detects only actual atomicity violations (not potential atomicity violations) and reports all accesses involved in the violation, along with their stack traces. As our analysis does not look for potential errors, it is less susceptible to false alarms. However, it is also less likely to catch rare bugs; nevertheless, with online analysis, the chance of catching rare bugs can be increased by analyzing longer runs and therefore more schedules.

2.1 Offline Analysis

2.1.1 Instrumentation and Logging

Redflag inserts targeted instrumentation using a suite of GCC compiler plug-ins that we developed specifically for Redflag. Plug-ins are a recent GCC feature that we contributed to its development. Plug-in support is a recent GCC feature, formally introduced in the 2010 release of GCC 4.5 [30], which we contributed to the development of. Compiler plug-ins execute during compilation and have direct access to GCC's intermediate representation of the code [13]. Redflag's GCC plug-ins search for relevant operations and instrument them with function calls that serve as hooks into Redflag's logging system.

Redflag currently logs four types of operations: (1) Field access: read from or write to a field in a `struct`; (2) Synchronization: acquire/release operation on a lock or wait/signal operation on a condition variable; (3) Memory allocation: creation of a kernel object, necessary for tracking memory reuse (Redflag can also track deallocations, if desired); (4) System call (syscall) boundary: syscall entrance/exit (used for atomicity checking).

When compiling the kernel with the Redflag plug-ins, the developer provides a list of `struct`s to target for instrumentation. Field accesses and lock acquire/release operations are instrumented only if they operate on a targeted `struct`. A lock acquire/release operation is considered to operate on a `struct` if the lock it accesses is a field within that `struct`. Some locks in the kernel are not members of any `struct`: these global locks can be directly targeted by name.

If a field of a targeted struct is itself a struct, assignments that modify a field of the nested struct are treated as updates to the field of the targeted struct. For example, if `inode` is a targeted struct, `inode.list_head.next = NULL` is treated as an update to `inode.list_head`.

To minimize runtime overhead, and to allow logging in contexts where potentially blocking I/O operations are not permitted (e.g., in interrupt handlers or while holding a spinlock), Redflag stores logged information in a lock-free in-memory buffer. I/O is deferred until logging is complete.

When an event occurs in interrupt context, the logging function also stores an interrupt ID that uniquely identifies the interrupt handler. Redflag assigns a new ID to each hardware interrupt that executes, keeping a per-processor stack to track IDs when interrupt handlers nest. Redflag also assigns interrupt IDs to Soft IRQs, a Linux mechanism for deferred interrupt processing. Offline analysis treats each interrupt handler execution as a separate thread.

When logging is finished, a backend thread empties the buffer and writes the records to disk. With 1GB of memory allocated for the buffer, it is possible to log 7 million events, which was enough to provide useful results for all our analyses.

2.1.2 Lockset Algorithm

Lockset is a well known algorithm for detecting *data races* that result from variable accesses that are not correctly protected by locks. Our Lockset implementation is based on Eraser [57].

A *data race* occurs when two accesses to the same variable, at least one of them a write, can execute together without intervening synchronization. Not all data races are bugs. A data race is *benign* when it does not affect the program's correctness.

Lockset maintains a *candidate set* of locks for each monitored variable. The candidate lockset represents the locks that have consistently protected the variable. A variable with an empty candidate lockset is potentially involved in a race. Before the first access to a variable, its candidate lockset is the set of all possible locks.

The algorithm tracks the current lockset for each thread. Each lock-acquire event adds a lock to its thread's lockset, and the corresponding release removes the it.

When an access to a variable is processed, the variable's candidate lockset is refined by intersecting it with the thread's current lockset. Thus, a variable's candidate lockset to be the set of locks that were held for *every* access to the variable. When a variable's candidate lockset becomes empty, the algorithm revisits every previous access to the same variable; if no common locks protected both the current access and that previous one, we report the pair as a potential data race.

Redflag produces at most one report for each pair of lines in the source code, so the developer does not need to examine multiple reports for the same race. Each report contains every stack trace that led to the race for both lines of code and the list of locks that were held at each access.

Beyond the basic algorithm described above, there are several common refinements that eliminate false alarms due to pairs of accesses that do not share locks but cannot occur concurrently for other reasons.

Variable initialization When a thread allocates a new object, no other thread has access to that object until the thread stores the new object's address in globally accessible memory. Most initialization routines in the kernel exploit this to avoid the cost of locking during initialization. As a result, most accesses during initialization appear to be data races to the basic Lockset algorithm.

The Eraser algorithm solves this problem by tracking which threads access variables to determine when each variable become shared by multiple threads [57]. Note that this makes the algorithm more sensitive to thread schedule in the monitored execution. We implement a variant of this idea: when a variable is accessed by more than one thread or accessed while holding a lock, it is considered shared. Accesses to a variable before its first shared access are marked as thread local, and Lockset ignores them.

Memory reuse When a region of memory is freed, allocation of new data structures in the same memory can cause false alarms in Lockset, because variables are identified by their location in memory. Eraser solves this problem by reinitializing the candidate lockset for every memory location in a newly allocated region [57]. Redflag also logs calls to allocation functions, so that it can similarly account for reuse.

Redflag does not track variables that are allocated on the stack. Instead, it ignores all accesses to stack variables, which we assume are never shared. Shared stack variables are considered bad coding style in C and are very rare in quality systems code.

Happened-before analysis Besides using locks for mutual exclusion, threads also synchronize using order-enforcing synchronization, such as condition variables. Redflag uses Lamport's happened-before relation [39] to track orderings due to condition variables: there is a happened-before ordering from a signaling event to the corresponding wake-up event in the thread that receives the signal. As usual, there are also happened-before orderings between consecutive events on the same thread, and the overall happened-before relation is the transitive closure of these basic happened-before orderings. Thread fork and join operations are usually also considered in the happened-before relation, but we did not find any accesses in the Linux kernel that depend on fork or join for synchronization.

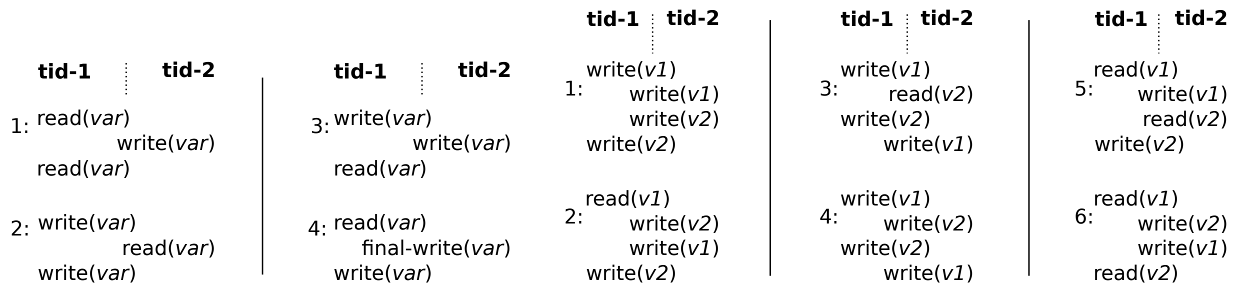
If a happened-before relation exists between two accesses, we assume that they cannot occur concurrently even if they share no common locks. This assumption is common in concurrency analysis, because it reduces false alarms, even though it can rule out feasible interleavings and hence cause actual errors to be missed.

2.1.3 Block-Based Algorithms

Redflag includes two variants of Wang and Stoller's block-based algorithm [67, 68]. These algorithms check for *atomicity*, which is similar to serializability of database transactions and provides a stronger guarantee than freedom from data races. Two atomic functions executing concurrently always produce the same result as if they executed in sequence, one after the other.

When checking atomicity for the kernel, system calls provide a natural unit of atomicity. By default, we check atomicity for each syscall execution. Not all syscalls need to be atomic, so Redflag provides a simple mechanism, discussed in Section 2.1.5, to specify smaller atomic regions.

We implemented two variants of the block-based algorithm: a single-variable variant that detects violations involving just one variable and a two-variable variant that detects violations involving multiple variables.



(a) Single variable (b) Double variable

Figure 2.1: Illegal interleavings in the single- and double-variable block-based algorithms

Note that a final write is the last write to a variable during the execution of an atomic region [68].

Single-variable variant The single-variable block-based algorithm decomposes each syscall execution into a set of *blocks*, which represent sequential accesses to a variable. Each block includes two accesses to the same variable in the same thread, as well as the list of locks that were held for the duration of the block (i.e., all locks that were acquired before the first access and not released until after the second access). The algorithm then checks each block, searching all other threads for any access to the block's variable that might interleave with the block in an unserializable way. An access can interleave with a block if it is made without holding any of the block's locks, and an interleaving is unserializable if it matches any of the patterns in Figure 2.1(a).

Two-variable variant The two-variable block-based algorithm also begins by decomposing each syscall execution into blocks. A two-variable block comprises two accesses to *different variables* in the same thread and syscall execution. The algorithm searches for pairs of blocks in different threads that can interleave illegally. Each block includes enough information about which locks were held, acquired, or released during its execution to determine which interleavings are possible. Figure 2.1(b) shows the six illegal interleavings for the two-variable block-based algorithm; Wang and Stoller give details of the locking information saved for each block [68].

Together, these two variants are sufficient to determine whether any two syscalls in a trace can violate each other's atomicity [68]. In other words, these algorithms can detect atomicity violations involving any number of variables.

Analogues of the Lockset refinements in Section 2.1.2 are used in the block-based algorithm to take variable initialization, memory re-use, and condition variables into account.

Deadlock For the block-based algorithms to produce correct results, the program trace must be free of potential deadlocks [68]. We assume traces to be free from deadlocks because all of the systems we analyzed with Redflag have been thoroughly tested with Lockdep, the Linux kernel's tool for catching potential deadlocks [47]. It would be straightforward to implement potential deadlock detection as one of Redflag's analysis tools, but we have not done this, because Lockdep is already widely available to kernel developers.

2.1.4 Algorithm Enhancements

The kernel is a highly concurrent environment and uses several different styles of synchronization. Among these, we found some that are not addressed by previous work on detecting concurrency violations. This section describes two of them—multi-stage escape and RCU—and discusses how Redflag handles them. Although we observed these synchronization methods in the Linux kernel, they may be used in other concurrent systems as well.

Multi-stage escape As explained in Section 2.1.2, objects within their initialization phases are effectively protected from concurrent access, because other threads do not have access to them. However, an object's accessibility to other threads is not necessarily binary. An object may be available to a limited set of functions during a secondary initialization phase and then become available to a wider set of functions when that phase completes. During the secondary initialization, some concurrent accesses are possible, but the initialization code is still protected against interleaving with many functions. We call this phenomenon *multi-stage escape*. As an example, inodes go through two stages of escape. First, after a short first-stage initialization, the inode is placed on a master inode list in the file system's superblock. File-system–specific code performs a second initialization and then assigns the inode to a dentry.

The block-based algorithm reported illegal interleavings between accesses in the second-stage initialization and syscalls that operate on files, like `read()` and `write()`. These interleavings are not possible, however, because file syscalls *always* access inodes through a dentry. Before an object is assigned to a dentry—its second escape—the second-stage initialization code is protected against concurrent accesses from file syscalls. Interleavings are possible with functions that traverse the superblock's inode list, such as functions called by the writeback thread, but they do not result in atomicity violations, because they were designed to interleave correctly with second-stage initialization.

To avoid reporting these kinds of false interleavings, we introduce *Lexical Object Availability* (LOA) analysis, which produces a relation on field access statements for each targeted `struct`. Intuitively, the LOA relation encodes observed orderings among lines of code. We use these orderings to infer when an object becomes unavailable to a region of code, marking the end of an initialization phase.

In the inode example, an access from the writeback thread is evidence that first-stage initialization is finished; and an access from a file syscall is evidence that first- and second-stage initialization are finished. These mean that accesses from those initialization routines are no longer possible.

The LOA algorithm first divides the log file into sub-traces. Each sub-trace contains all accesses to one particular instance o of a targeted type of `struct` S. For each sub-trace, which is for some instance of some `struct` S, the algorithm adds an entry for a pair of statements in the LOA relation for S, denoted LOA_S, when it observes that one of the statements occurred after the other in a different thread in that sub-trace. Specifically, for a `struct` S and read/write statements a and b, (a, b) is included in LOA_S iff there exists a sub-trace for an instance of `struct` S containing events e_a and e_b such that:

1. e_a is performed by statement a, and e_b is performed by statement b, and

2. e_a occurs before e_b in the sub-trace, and

3. e_a and e_b occur in different threads.

We modified the block-based algorithm to report an atomicity violation only if the interleaving statements that caused the violation are allowed to interleave by their LOA relation. For an event produced by statement b to interleave a block produced by statements a and c, the LOA relation must contain the pairs (a, b) and (b, c). Otherwise, the algorithm considers the interleaving impossible.

Returning to the inode example, consider a and c to be statements from the secondary initialization stage and b to be a statement in a function called by the `read` syscall. Because statement b cannot access the inode until after secondary initialization is finished, (b, c) cannot be in LOA_{inode}, the LOA relation for inode objects.

We also added LOA analysis to the Lockset algorithm: it reports that two statements a and b can race only if both (a, b) and (b, a) are in the LOA relation for the `struct` that a and b access.

For example, the kernel sometimes uses condition variables to protect against (i.e., postpone) certain operations by other threads to inodes that are in a startup phase, which lasts longer than initialization. While happened-before analysis (described in Section 2.1.2) is the traditional approach to take order-enforcing synchronization into account, we found that LOA analysis is equally effective for this; for example, in our experiments, LOA analysis eliminated all false alarms corresponding to interleavings precluded by condition variables. After finding this, we did not use happened-before analysis in subsequent experiments, including the experiments reported in Section 2.4. LOA analysis can also infer *destruction* phases, when objects typically return to being exclusive to one thread, even when the start of the destruction phase is not indicated by an explicit synchronization operation.

Because LOA filters interleavings based on the observed order of events, it can eliminate warnings corresponding to actual errors. The common technique of filtering based on when variables become shared (see Section 2.1.2) has the same problem: if a variable becomes globally accessible but is not promptly accessed by another thread, neither technique recognizes that such an access is possible. Dynamic escape analysis addresses this problem by determining precisely when an object becomes globally accessible [68], but it accounts for only one level of escape and hence would not eliminate many of the false alarms successfully filtered by LOA analysis.

Syscall interleavings Engler and Ashcraft observed that dependencies on data prevent some kinds of syscalls from interleaving [24]. For example, a `write` operation on a file never executes in parallel with an `open` operation on the same file, because user-level programs have no way to call `write` before `open` finishes.

These dependencies are actually a kind of multi-stage escape. The return from `open` is an escape for the file object, which then becomes available to other syscalls, such as `write`. For functions that are called from only one syscall, our LOA analysis already rules out impossible interleavings between statements in syscalls with this kind of dependency.

However, when a function is reused in several syscalls, the LOA relation, as described above, cannot distinguish executions of the same statement that were executed in different syscalls. As a result, if LOA analysis sees that an interleaving in a shared function is possible between one pair of syscalls, it will believe that the interleaving is possible between any pair of syscalls.

To overcome this problem, we augment the LOA relation with some context sensitivity; the modified LOA relation contains entries of the form $((syscall, statement), (syscall, statement))$.

```
/* [Thread 1] */
spin_lock(inode->lock);
inode->i_state |= I_SYNC;
spin_unlock(inode->lock);
                              /* [Thread 2] */
                              spin_lock(inode->lock);
                              if (inode->i_state & I_CLEAR) {
                                /* ... */
                              }
                              spin_unlock(inode->lock);
/* [Thread 1] */
spin_lock(inode->lock);
inode->i_state &= ~I_SYNC;
spin_unlock(inode->lock);
```

Figure 2.2: False-alarm atmocity violation for a bitfield variable
This interleaving appears to violate the atomicity of the `i_state` field, but the two threads actually access independent bits within the bitfield.

This causes LOA analysis to distinguish the possible orderings of each statement in the context of each syscall; it achieves the same effect as making multiple copies of each function, one for each syscall. Statements that do not execute in a syscall are instead paired with the name of the kernel thread they execute in.

RCU Read-Copy Update (RCU) synchronization is a recent addition to the Linux kernel that allows very efficient read access to shared variables [44]. A typical RCU-write copies the protected data structure, modifies the local copy, and then replaces the pointer to the original copy with a pointer to the updated copy. RCU synchronization by itself does not protect against lost updates, so writers must also use locking if they want to prevent lost updates. A reader needs only to surround read-side critical sections with `rcu_read_lock()` and `rcu_read_unlock()`, which ensure that the copy of the shared data structure that they are reading does not get freed during the critical section, even if a writer concurrently replaces that copy with an updated copy.

We extended Lockset to test for correct use of RCU synchronization. When a thread enters a read-side critical section by calling `rcu_read_lock()`, our implementation adds a virtual RCU lock to the thread's lockset. We do not report a data race between a read and a write if the read access has the virtual RCU lock in its lockset. However, conflicting writes to an RCU-protected variable will still produce a data race report.

2.1.5 Filtering False Positives and Benign Warnings

Bit-level granularity We found that many false alarms produced by the block-based algorithms were caused by *flag variables*, like the `i_state` field in Figure 2.2, which group several boolean values into one integer variable. Because several flags are stored in the same variable, an access to one flag appears to access all flags in the variable. Erickson et al. observed this same pattern in the

Windows 7 kernel and account for it in their DataCollider race detector [25].

Figure 2.2 shows an example of an interleaving that the single-variable block-based algorithm would report as a violation. The two bitwise assignments in thread 1 both write to the `i_state` field. These two writes form a block between which the read of the `i_state` field in thread 2 can interleave; this is one of the illegal patterns shown in Figure 2.1(a). However, there is no atomicity problem, because thread 1 writes only the `I_SYNC` bit, and thread 2 reads only the `I_CLEAR` bit.

We eliminate such false alarms by modifying the block-based algorithms to treat any variable that is sometimes accessed using bitwise operators as 64 individual variables (on 64-bit systems). Redflag's plug-ins detect bitwise operations and pass their bitmasks to the logger so that the block-based analysis can identify which operations read or write individual bits. Our analysis still detects interleavings between bitwise accesses to individual flags and accesses that involve the whole variable.

Idempotent operations An operation is *idempotent* if, when it is executed multiple times on the same variable, only the first execution changes the variable's value. For example, setting a bit to 1 in a flag variable is an idempotent operation. When two threads execute an idempotent operation, the order of these operations does not matter, so atomicity violation reports involving them are false positives. The user can annotate lines that perform idempotent operations. Our algorithms filter out warnings that involve only these lines.

Choosing atomic regions We found that many atomicity violations initially reported by the block-based algorithms are benign: the syscalls involved are not atomic, but are not required to be atomic. For example, the Btrfs file system function `btrfs_file_write()` loops through each page that it needs to write. The body of the loop, which writes one page, should be atomic, but the entire function is not required to be atomic.

Redflag lets the user break up atomic regions by marking lines of code as *fenceposts*. A fencepost ends the current atomic region and starts a new one. For example, placing a fencepost at the beginning of the page-write loop in `btrfs_file_write()` prevents Redflag from reporting atomicity violations spanning two iterations of the loop.

Fenceposts provide a simple way for developers to express expectations about atomicity. Even for the largest systems we checked, about an hour of work placing fenceposts led to substantially better reports.

To facilitate fencepost placement, Redflag determines which lines of code, if marked as fenceposts, would filter the most atomicity violations. Any line of code that executes in the same thread as a block and between the first and last operations of the block (see Section 2.1.3 for a description of blocks) can serve as a fencepost that filters all violations involving that block. After block-based analysis produces a list of atomicity violations with corresponding blocks, fencepost inference proceeds by greedily choosing the fencepost that will filter the most violations, removing these violations from its list, and repeating until no violations remain. The result is a list of potential fenceposts sorted by the number of violations they filter. The user can examine these candidate fenceposts to see whether they lie on the boundaries of sensible atomic regions in the code. The resulting set of fenceposts is not necessarily the smallest set of fenceposts that filters all violations, but computing such a set is equivalent to the NP-complete minimum hitting set problem.

2.2 Online Atomicity Checking

The block-based algorithms used in our offline atomicity checking increase the likelihood of finding bugs by implicitly considering alternative schedules for (i.e., permutations of) the observed trace, relying on sophisticated reasoning about schedule feasibility, such as that provided by our LOA analysis technique. In contrast, our online atomicity checking algorithm reports only atomicity violations that actually occur in the observed run.

The illegal interleavings used in the single-variable block-based algorithm, which we list in Figure 2.1(a), involve a pair of accesses within an atomic region interleaved by an access from a remote thread. We categorize these atomicity-violating patterns into *interleaved write violations* and *interleaved read violations*, based on the type of interleaved operation, and use different techniques to detect them.

2.2.1 Interleaved write violations

In an interleaved write violation, a remote write interferes with a variable's value while it is in use by another thread. When the write interleaves a pair of reads, the two reads observe different values, violating atomicity. Similarly, an atomic region that writes a field should observe the value it originally wrote when it later reads that same field. A remote write that changes that value between those two operations violates atomicity.

To detect this category of atomicity violations, our online algorithm maintains *shadow memory* for each atomic region. The shadow memory for an atomic region contains a private copy of every targeted field that the atomic region accesses. Some software transactional memories use a similar kind of private memory to defer updates until an atomic region commits [33]. Shadow memory does not defer writes to system memory, however, because our online analysis aims only to detect conflicts, not prevent them.

The first time an atomic region accesses a targeted field, an entry is created for that field in the atomic region's shadow memory. All writes to that field by that atomic region update both the atomic region's shadow memory and system memory. All accesses to the field trigger a check that the field's actual value, in system memory, matches the shadow copy. Any discrepancy means that an interleaved operation in another thread modified the field and indicates an atomicity violation. In contrast to interleaving 4 in Figure 2.1(a), our online algorithm does not check whether the interleaved write is a final write (note that this condition cannot be checked until the atomic region containing the interleaved write terminates, so checking this condition efficiently in the online algorithm is more complicated than checking this condition in the offline algorithm). The effect of omitting this check is that the algorithm checks conflict serializability rather than view serializability. In future work, we plan to modify the algorithm to check this condition, to see how many atomicity violation reports are eliminated by using the more permissive concept of view serializability.

2.2.2 Interleaved read violations

An *interleaved read violation* is possible when an atomic region writes the same field more than once. To preserve atomicity, all writes except the last one should be private to the atomic region.

A read by another thread interleaved between two writes in an atomic region will observe a private value, violating atomicity.

To detect interleaved read violations, our algorithm checks when overwriting a field's value in an atomic region's shadow memory, whether the value being overwritten was read by another thread. We do this by comparing the time of the most recent read by another thread with the time that the value being overwritten was written, and reporting an atomicity violation if the former is after the latter. To implement this, we augment each atomic region's shadow memory so that, when a write is saved to an atomic region's shadow memory, the time of the write is stored along with it. We introduce a global data structure, which we call *global shadow memory*, that stores the time of the most recent read to each field by any thread. Each atomic region's shadow memory and global shadow memory are implemented as red-black trees.

As mentioned above, the algorithm needs the time of the most recent read by another thread, i.e., a thread other than the current thread. In order to provide this information, the global shadow memory stores information about two reads: the *most recent* read by any thread, and the *runner-up* read, which is the most recent read from any thread other than the thread that made the most recent read. When a read occurs, its timestamp gets saved as the most recent. The previously most recent read gets saved as the runner-up if and only if it was from a different thread than the new read. It is never necessary to store more than two reads because the two reads stored are never from the same thread.

For efficiency, the times of reads and writes are based on a logical clock, rather than a real-time clock. The logical clock is a system-wide counter incremented on every write to an instrumented field. Using a real-time clock would be slower (because reading a real-time clock takes longer than reading a counter) and would not be accurate enough to reliably determine the ordering of memory accesses amongst multiple cores.

As another optimization, Redflag updates the timestamp of the last read in global shadow memory only when an atomic region reads a field it has not accessed before. Any later read from the same atomic region either observes the same value or exposes an interleaved write violation. In the latter case, the later read is also involved in an interleaved read violation, which is not reported due to this optimization. We consider this acceptable, because the interleaved write violation that is reported already shows that the later read causes atomicity-violating interference between the two atomic regions involved.

2.2.3 Instrumentation

Instrumentation for online analysis is built on the same plug-in as Redflag Logging. Just as with Redflag Logging, instrumentation for online analysis can be configured to target specific data structures for analysis. In addition to all the information used for logging, the shadow memory also needs to know the address of each accessed field and, for writes, the value that was written. We modified the plug-in to provide this information.

The plug-in must directly capture the values of field writes to avoid concurrency errors in the implementation of Redflag itself. The instrumentation could copy the written value from system memory after the write takes place, but there is no efficient mechanism to lock the field between the write and the copy. Instead, the plug-in modifies the write operation so that it assigns the value to an unshared temporary variable, which is passed to the shadow memory and copied to the original field.

In addition to instrumenting explicit assignments to targeted fields (an explicit field assignment has the form *expr*`.`*field* `=` *expr* or *expr*`->`*field* `=` *expr*), our plug-ins add instrumenation to observe two other kinds of updates, described in the following paragraphs.

First, assignments that modify a field of a targeted struct through a pointer to the field are not explicit field assignments and hence require special treatment by the plug-in. In the kernel code we examined, such assignments occur only in contexts where the address of a field is passed as an argument to a function, e.g., `atomic_inc(&(foo->counter))`, where `atomic_inc` atomically increments the value stored in the memory location to which its argument points. Therefore, the plug-in syntactically identifies function calls that take a pointer to a targeted field as a parameter and, if the called function is one that we manually identified as modifying the value stored where that argument points, the plug-in instruments the function call as a write to that field. We identified about thirty such functions in the kernel, mostly functions, such as `atomic_inc`, that abstract hardware-supported interlocked arithmetic operations.

Second, functions like `memcpy` can modify all the fields within a block of memory without any direct assignment statements. These functions are used extensively in Btrfs, for example, to copy entire structs that are stored within B-tree nodes. We identified three such functions used in the Btrfs code to which we applied our online atomicity tool, and we manually modified them to call a shadow `memcpy` function after completing the system memory copy. Btrfs uses these internal functions instead of standard `memcpy` because `memcpy` is not designed for data structures that may reside in high memory.

The shadow `memcpy` function mirrors the effects of the actual `memcpy` operation in the atomic region's shadow memory. First, the shadow `memcpy` finds all the fields in the `memcpy` source memory region that have entries in the atomic region's shadow memory. This search is performed as a range query on the red-black tree used to store the shadow memory. Each of these fields' entries is copied into the atomic region's shadow memory for the destination memory region.

Redflag treats a `memcpy` operation from a source struct to a destination struct differently from individually copying each field with an assignment (as in `dest.a = source.a`) in three ways. First, any source region fields that the atomic region never accessed will appear in the destination region as if they were never accessed. The shadow `memcpy` function does not create new entries in the destination region for such entries, and it erases any entries that already exist. Second, the destination field does not appear in shadow memory to have been written by the `memcpy`; it appears as if it were accessed in the same way as the corresponding source field, because its entry in the shadow memory is copied verbatim. These shadow `memcpy` semantics can suppress violations that would be triggered by a write and not a read. They also change which operation Redflag implicates in its error reports. We designed these semantics to treat copied objects as if they existed in only one place throughout their lifetimes, because we believe this produces more useful error reports. Finally, for performance reasons, the shadow `memcpy` operation does not alter the global shadow memory: no read timestamps are recorded, and stack traces used for debugging are not copied (these stack traces are discussed in Section 2.2.4).

2.2.4 Debugging information

Providing comprehensive debugging information for each reported violation, as Redflag's offline analysis does, requires storing additional information for each access.

Each time Redflag saves a value in an atomic region's shadow memory, it stores a stack trace

`address`	Address of the field.
`value`	The atomic region's view of the field's value: set on the first access within the atomic region and on each subsequent write by the atomic region.
`was_written`	True iff this field has been written within this atomic region.
`stacktrace`	Stack trace saved when the `value` field of this entry was last updated.
`write_timestamp`	Timestamp of the first write to this field in the atomic region.

Table 2.1: Data stored in an atomic region's shadow memory
Each atomic region has its own shadow memory, which has an entry for each targeted field it has accessed.

`address`	Address of the field.
`remote_reads`	Struct containing information about the most recent read and the runner-up read from this field. This struct stores `thread_id`, `stacktrace`, and `timestamp` fields for each read.
`remote_writes`	Struct containing information about the most recent write and the runner-up write to this field. This struct stores `thread_id` and `stacktrace` fields for each write.

Table 2.2: Data stored in global shadow memory
There is one global shadow memory shared by all atomic regions, which contains an entries for every targeted field.

with it. On detecting a violation, Redflag reports that saved stack trace and the current stack trace, providing debugging information for the two accesses occurring in the violating atomic region.

Equally important for debugging is information about the interleaved access that caused the violation. This information is not available in the atomic region's shadow memory, because that access is performed by a different thread. To provide that information, Redflag stores, in the global shadow memory, a stack trace for the most recent write to every field. When an interleaved write violation is detected, this stack trace is reported as the stack trace associated with the interleaved write. Similarly, when Redflag updates a field's last-read timestamp in global shadow memory, it also saves a stack trace along with the timestamp, in order to report stack traces for interleaved reads.

Under rare circumstances, the interleaved write implicated in a violation can appear to be from the same thread as the two accesses in the violated atomic region. This happens because we do not update the global shadow memory and actual system memory in one atomic operation, making it possible for the stack trace saved in global memory to not correspond to the actual most recent write to a field. When the global memory shows an interleaving write from the same thread, we instead report the stack trace associated with a runner-up write, which is the most recent write from any thread other than the thread that made the most recent write. This is analogous to the way that we sometimes report a runner-up read as part of an interleaved-read violation, as described in Section 2.2.2. Even when the reported write is not actually the most recent, it will have occurred in between the two accesses in the violated atomic region, making it a legitimately violating access.

2.2.5 Pseudo-code

For reference, Tables 2.1 and 2.2 summarize the information stored in each atomic region's shadow memory and global shadow memory, respectively, including the debugging information discussed in Section 2.2.4. The fields described in these figures are referenced in the pseudo-code for the online atomicity algorithm, which is presented in Figures 2.3 and 2.4.

2.3 Weak memory model errors

Weak memory models are an oft overlooked source of concurrency errors in systems code. Under weak memory models, the compiler and processor can reorder memory accesses for performance reasons. Reorderings are invisible to single-threaded code, but developers of multi-threaded programs must account for them.

In the *sequentially consistent* memory model, loads and stores across all processors are totally ordered, such that a load always observes the value written by the most recent store to the same address [40]. Furthermore, the total order is consistent with each processor's order of execution. Figure 2.5(a) shows a sequentially consistent trace of a simple parallel program.

Weak memory models are those models that do not provide the sequential consistency guarantee. Either the global ordering of memory events may be inconsistent with some processor's execution order, or there might not be a global ordering. Figure 2.5(b) shows the same program executing on a weak memory model.

The paradoxical result shown would not be possible with sequential consistency. Under any sequentially consistent ordering, the last `load` must execute after both `store` operations, meaning that at least one of `r1` and `r2` would have a final value of 1.

The result in Figure 2.5(b) is possible, though, on architectures that implement *store buffering* because store buffers do not enforce sequential consistency. In store-buffered memory models, writes are held temporarily in the buffer so that the processor does not have to wait for the write to complete before retiring subsequent instructions, similar to how file systems use write caches to optimize write system calls. In our example, the `load` from *v1* can execute after the concurrent `store` retires but before CPU 1 propagates the new value of *v1* from its store buffer to memory, allowing CPU 2 to observe the now stale initial value. The Total Store Order (TSO) memory model formalizes the memory reorderings allowed by store buffers [12, 51].

In systems code that avoids locks for performance reasons, programmers often rely on the ordering of memory accesses for synchronization. Atig et al. note that the pattern in Figure 2.5 appears in Dekker's mutual exclusion protocol [4]. We also found this pattern in the Linux kernel, embedded in the synchronization between consumer threads polling a network socket and a producer thread receiving a packet on that socket, as described in a Linux Kernel Mailing List (LKML) bug report [50]. Burckhardt and Musuvathi found a similar producer/consumer bug in a Microsoft concurrency library [12].

For this kind of lockless synchronization to work, programmers need to manually enforce sequential consistency using *memory fences*[1]. On encountering a full memory fence, the processor

[1] Among programmers, the term *memory barrier* is often used to describe a fence, though there is no relation the

```
OnAccess(newAccess):
  address := FieldAddress(newAccess)
  prevAccess := ReadShadowEntry(address)
  glEntry := ReadGlobalEntry(address)
  isFirstAccess := !Exists(prevAccess)

  if !Exists(glEntry) then:
    glEntry := new GlobalEntry(address)
  fi

  if !isFirstAccess then:
    CheckAtomicity(newAccess, prevAccess, glEntry)

  ; update the atomic region's shadow memory
  if isFirstAccess then:
    prevAccess := new LocalEntry(address)
  fi

  if IsWrite(newAccess) or isFirstAccess then:
    prevAccess.value := GetValue(newAccess)
    prevAccess.was_written := prevAccess.was_written or IsWrite(newAccess)
    prevAccess.stacktrace := GetCurStack()
    prevAccess.write_timestamp := GetLogicalTime()
    WriteShadowEntry(address, prevAccess)
  fi

  ; update global shadow memory
  if IsWrite(newAccess) then:
    lastWrite := glEntry.remote_writes.most_recent
    newWrite.thread_id := GetCurThread()
    newWrite.stacktrace := GetCurStack()

   if (last_write.thread_id != GetCurThread()) then:
     glEntry.remote_writes.runner_up := last_write
   fi
   glEntry.remote_writes.most_recent := newWrite

  else if isFirstAccess then:
    lastRead := glEntry.remote_reads.most_recent
    newRead.thread_id := GetCurThread()
    newRead.timestamp := GetLogicalTime()
    newRead.stacktrace := GetCurStack()

    if last_read.thread_id != GetCurThread() then:
      glEntry.remote_reads.runner_up := lastRead
    fi
    glEntry.remote_reads.most_recent := newRead
  fi

  WriteGlobalEntry(address, glEntry)

  if IsWrite(newAccess) then:
    IncrementLogicalClock()
  fi
```

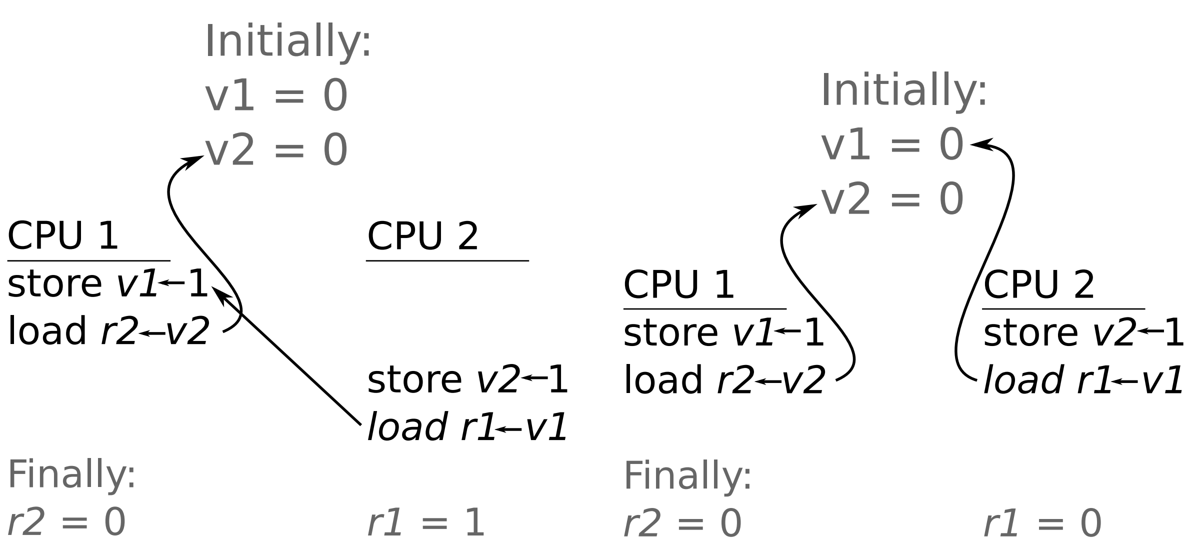
Figure 2.3: Pseudo-code for the Redflag online atomicity checking algorithm.

```
CheckAtomicity(newAccess, prevAccess, glEntry):
  if prevAccess.value != GetValue(newAccess) and
     !(prevAccess.was_written and IsWrite(newAccess)) then:
    ; interleaved write violation
    remoteWrite := ChooseRemoteAccess(glEntry.remote_writes)
    ReportViolation(prevAccess.stacktrace,
                    remoteWrite.stacktrace,
                    GetCurStack())
  fi

  if IsWrite(newAccess) and prevAccess.was_written then:
    remoteRead := ChooseRemoteAccess(glEntry.remote_reads)
    if Exists(remoteRead) and remoteRead.timestamp > prevAccess.write_timestamp then:
      ; interleaved read violation
      ReportViolation(prevAccess.stacktrace,
                      remoteRead.stacktrace,
                      GetCurStack())
    fi
  fi

; Among a pair of accesses, choose the most recent
; from another thread, unless no such access exists.
ChooseRemoteAccess(accesses):
  if Exists(accesses.most_recent) and
     accesses.most_recent.thread_id != GetCurThread() then:
    return accesses.most_recent
  else:
    return accesses.runner_up
  fi
```

Figure 2.4: Pseudo-code for the Redflag online atomicity checking algorithm, continued.



(a) Sequentially consistent (b) Violating sequential consistency

Figure 2.5: Two possible executions of a star-crossed data race
One is possible on sequentially consistent architectures and one is only possible on a weak memory model, like Total Store Order (TSO) [12].

is not allowed to execute any memory operations that follow the fence until the effects of all prior memory operations are committed. Compilers also honor memory fences and will not reorder variable reads and writes across fences.

A pair of correctly placed fences can prevent the inconsistent result in Figure 2.5(b): one in each thread between the `store` and the `load`. Linux developers chose this approach to fix the previously mentioned network socket bug [50]. Note that a single fence is not enough to enforce sequentially consistent behavior here. The LKML discussion of the network socket bug informally acknowledges this requirement, noting that a fence needs to "pair" with another fence in a racing thread to be useful.

The socket error essentially results in a deadlock when a thread attempts to poll a socket at the exact moment that another processor is reading an incoming packet bound for the same socket. As with Dekker's protocol, each processor proceeds in two steps, first with a write to indicate it has initiated the protocol and then with a read to check if a racing thread is simultaneously executing the protocol. For the polling thread, this means 1) placing itself on the list of threads waiting for packets from the socket then 2) checking whether packets are already available. Meanwhile, the receiving processor 1) updates a variable to indicate an available packet then 2) checks if there are any waiting threads that need to be notified of the new packet.

The sequentially inconsistent outcome in Figure 2.5(b) deadlocks because the polling threads believes there are no incoming packets, and the incoming packet believes there are no polling threads. As a result, the polling thread waits on a condition variable, but the receiving processor does not wake it, leaving it to sleep indefinitely unless another packet eventually arrives.

To see how the socket polling process embeds the pattern in Figure 2.5, it helps to know exactly what pair of variables are involved. In this case, *v1* is the TCP/IP sequence number of the socket's next unread byte, `tp->rcv_nxt`, and *v2* is the head pointer for the socket's waiter list.

We focus our bug-finding efforts on this specific pattern of memory accesses, which we refer to as a *star-crossed* data race. A star-crossed data race exists between a pair of variables *v1* and *v2* when 1) there are racing read and write accesses to *v1*, 2) the write is followed by a read from *v2* and the read is preceded by a write to *v2*, and 3) at least one of the two racing threads lacks a protecting memory fence. Here, a protecting memory fence is one that separates the *v1* access from the corresponding *v2* access. A star-crossed data race is still possible if *v2* is protected by a lock, so long as that lock does not also protect *v1*. On many architectures, releasing the *v2* lock would serve as a fence, preventing the sequentially inconsistent behavior, but this guarantee does not hold for all Linux-supported systems.

2.3.1 Star-crossed block-based algorithm

We propose a star-crossed data race dectector that operates like the two-variable block-based algorithm. Like the block-based algorithms, our new detector begins by collecting a set of blocks for each thread.

Each block has a write operation that is followed by a read operation in the same thread but from a different field. For each read operation in the thread from any variable *v2*, we create one

---

barriers synchronization primitive.

block for each previously written variable *v1* s.t. $v1 \neq v2$. The block comprises a pair of the latest write to *v1* preceding the read and the read itself. Each block is also annotated with any fence operations that executed between the write and read operations and the set of locks protecting each of the two accesses.

Two blocks are potentially conflicting if operate on the the same pair of variables but in opposite order and they execute in different threads. For example, a blocking consisting of a write to *v1* then read from *v2* potentially conflicts with a block that writes to *v2* then reads from *v1*.

Unlike the two-variable block-based algorithm, our detector does not need to check for any kind of interleaving between two potentially conflicting blocks. It only checks that the intersection of the read lockset from one block and the write lockset from the other block is empty, meaning the two accesses can race, and that one of the blocks has no memory fences. Any potentially conflicting blocks that meet these criteria indicate a star-crossed data race. As with the our Lockset implementation we will also need to use Lexical Object Availability (LOA) analysis to filter out any pair of blocks where initialization prevents the race from actually occuring.

Our current implementation of the star-crossed block-based algorithm generates too many false alarms to be of use in most situations. Improving the algorithm's accuracy is a subject of future work.

2.4 Evaluation

To evaluate Redflag's accuracy and performance, we exercised it on three kernel components: Btrfs, Wrapfs, and Noveau. Btrfs is a complex in-development on-disk file system. Wrapfs is a pass-through stackable file system that serves as a stackable file system template [70]. Because of the interdependencies between stackable file systems and the underlying virtual file system (VFS), we instrumented all VFS data structures along with Wrapfs's data structures. We exercised Btrfs and Wrapfs with Racer [62], a workload designed to test a variety of file-system system calls concurrently. Nouveau is a video driver for Nvidia video cards. We exercised Nouveau by playing a video and running several instances of `glxgears`, a simple 3D OpenGL example.

Lockset results Lockset revealed two confirmed locking bugs in Wrapfs. The first bug results from an unprotected access to a field in the `file struct`, which is a VFS data structure instrumented in our Wrapfs tests. A Lockset report shows that parallel calls to the `write` syscall can access the `pos` field simultaneously. Investigating this race, we found an article describing a bug resulting from it: parallel writes to a file may write their data to the same location in a file, in violation of POSIX requirements [18]. Proposed fixes carry an undesirable performance cost, so this bug remains.

The second bug is in Wrapfs itself. The `wrapfs_setattr` function copies a data structure from the wrapped file system (the *lower inode*) to a Wrapfs data structure (the *upper inode*) but does not lock either inode, resulting in several Lockset reports. We discovered that file truncate operations call the `wrapfs_setattr` function after modifying the lower inode. If a truncate operation's call to `wrapfs_setattr` races with another call to `wrapfs_setattr`, the updates to the lower inode from the truncate can sometimes be lost in the upper inode. We confirmed this bug with Wrapfs developers.

	setattr		stat		atime		useless read		counting		struct granularity		untraced lock		other	
Btrfs					5				61	6	2				40	
Wrapfs	34	6		14	43										2	
Nouveau							1				21	2		1		

Table 2.3: Summary of results of block-based algorithms

From left to right, the columns show: reports caused by `wrapfs_setattr`, reports caused by `touch_atime`, reports caused by reads with no effect, reports involving counting variables, reports caused by coarse-grained reporting of `struct` accesses, and reports that do not fall into the preceding categories. Each column has two sub-columns, with results for the single-variable and two-variable algorithms, respectively. Empty cells represent zero.

Lockset detected numerous benign races: 8 in Btrfs, and 45 in Wrapfs. In addition, it detected benign races involving the `stat` syscall in Wrapfs, which copies file metadata from an inode to a user process without locking the inode. The unprotected copy can race with operations that update the inode, causing `stat` to return inconsistent (partially updated) results. This behavior is known to Linux developers, who consider it preferable to the cost of locking [7], so we filter out the 29 reports involving `stat`.

Lockset produced some false alarms due to untraced locks: 2 for Wrapfs, and 11 for Noveau. These false alarms are due to variable accesses protected by locks external to the traced `struct`s. These reports can be eliminated by telling Redflag to trace those locks.

Block-based algorithms results Table 2.3 summarizes the results of the block-based algorithms. We omitted four `struct`s in Btrfs from the analysis, because they are modified frequently and are not expected to update atomically for an entire syscall. The two-variable block-based algorithm is compute- and memory-intensive, so we applied it to only part of the Btrfs and Wrapfs logs.

For Wrapfs, the `wrapfs_setattr` bug described above causes atomicity violations as well as races; these are counted in the "setattr" column. The results for Wrapfs do not count 86 reports for the file system that Wrapfs was stacked on top of (Btrfs in our experiment). These reports were produced because we told Redflag to instrument all accesses to targeted VFS structures. A Wrapfs developer uninterested in these reports could easily eliminate them by telling Redflag to instrument accesses to those structures only in Wrapfs code.

For Wrapfs, the unprotected reads by `stat` described above cause two-variable atomicity violations, which are counted in the "stat" column. These reads do not cause single-variable atomicity violations, because inconsistent results from `stat` involve multiple inode fields, some read before an update by a concurrent operation on the file, and some read afterwards.

For Noveau, the report in the "untraced lock" column involves variables protected by the Big Kernel Lock (BKL), which we did not tell Redflag to track.

The "counting" column counts reports whose write accesses are increments or decrements (e.g., accesses to reference count variables). Typically, these reports can be ignored, because the order in which increments and decrements execute does not matter—the result is the same. Our plug-ins mark counting operations in the log, so Redflag can automatically classify reports of this type.

	Fenceposts	Bit-level granularity	LOA	Unfiltered
Btrfs	44	0	159	108
Wrapfs	81	6	215	79
Nouveau	-	2	70	22

Table 2.4: Number of false positives filtered out by various techniques

The "struct granularity" column counts reports involving `struct`s whose fields are grouped together by Redflag's logging. Accesses to a `struct` that is *not* targeted get logged when the non-targeted `struct` is a field of some `struct` that is targeted and the access is made through the targeted `struct`. However, all the fields in the non-targeted `struct` are labeled as accesses to the field in the targeted `struct`, so they are treated as accesses to a single variable. Revisiting the example in Section 2.1.1, if `inode` is targeted and `list_head` is not, then accesses to all fields in `inode.list_head` (such as `inode.list_head.next`) are treated as accesses to the field `inode.list_head`. This can cause false alarms, in the same way that bit-level operations can (see Section 2.1.5). These false alarms can be eliminated by adding the non-targeted struct to the list of targeted structs.

Filtering Table 2.4 shows how many reports were filtered from the results of the single-variable block-based algorithm (which produced the most reports) by manually chosen fenceposts, bit-level granularity, and LOA analysis. We used fewer than ten manually chosen fenceposts each for Btrfs and Wrapfs. Choosing these fenceposts took only a few hours of work. We did not use fenceposts for our analysis of Nouveau because we found that entire Nouveau syscalls are atomic.

LOA analysis is the most effective among these filters. Only a few `struct`s in each of the modules we tested go through a multi-stage escape, but those `struct`s are widely accessed. It is clear from the number of false alarms removed that a technique like LOA analysis is necessary to cope with the complicated initialization procedures in systems code.

Some reports filtered by LOA analysis may be actual atomicity violations, as discussed in Section 2.1.4. This happened with a bug in Btrfs' inode initialization that we discovered during our experiments. The Btrfs file creation function initializes the new inode's *file operations vector* field just after the inode is linked to a dentry. This linking is the inode's second stage of escape, as discussed in Section 2.1.4. When the dentry link makes the new inode globally available, there is a very narrow window during which another thread can open the inode while the inode's file operations vector is still empty. This bug is detected by the single-variable block-based algorithm, but the report is filtered out by LOA analysis. LOA analysis will determine that the empty operations vector is available to the `open` syscall only if an open occurs during this window in the logged execution, which is unlikely. Dynamic escape analysis correctly recognizes the possible interleaving in any execution, but has other drawbacks, because it accounts for only one level of escape. In particular, the bug can be fixed by moving the initialization of the file operations vector earlier in the function: before the inode is linked to a dentry, but still after the inode's first escape. Dynamic escape analysis would still consider the interleaving possible, resulting in a false alarm.

We tested the fencepost inference algorithm in Section 2.1.5 on Btrfs. We limited it to placing fenceposts in Btrfs functions (not, e.g., library functions called from Btrfs functions). In our first tests, the fenceposts that filtered the most violations were in common functions, like linked-list operations, that occurred frequently in the log. We improved these results by limiting the algorithm to placing fenceposts in Btrfs functions. After this, The algorithm produced a useful list of candidate fenceposts. For example, the first fencepost on the list is just before the function that serializes an inode, which is reasonable because operations that flush multiple inodes to disk are not generally designed to provide an atomicity guarantee across all their inode operations.

Online atomicity results We evaluated the online atomicity algorithm on Btrfs and Wrapfs with two kinds of application workloads: the Racer [62] workload that we used for evaluating offline atomicity analysis, and more targeted workloads that exercise two specified system calls. The following paragraphs discuss the results for these two kinds of workloads in turn. We found, with both of these kinds of workloads, that the online algorithm is effective at detecting atomicity bugs, even though it detects only actual violations, not potential violations.

After inserting fenceposts, results from Racer on Btrfs were similar to the results obtained with the single-variable block-based algorithm: 60 out of 83 violations were caused by counting variables. The remaining reports were for atomicity violations (recall that the online algorithm produces no false alarms) that we concluded are permissible based on the semantics of the system calls, but do not span distinct atomic operations that should be separated by a fencepost.

For example, one such violation involves the `btrfs_add_link` function, which adds a link to a file. This function performs two separate B-tree updates. The first update is done by the function `btrfs_insert_dir_item`, which adds a file to its parent directory by inserting a `dir_item` into the B-tree. The second update is done by `btrfs_update_inode`, which searches the B-tree for the parent directory's inode and updates it to reflect its increased size. A B-tree update by another system call can be interleaved between these two updates, causing the two B-tree traversals in `btrfs_add_link` to each observe different keys as they navigate down the B-tree's internal nodes, resulting in atomicity violation reports. These violations are permissible, because `btrfs_update_inode` is still able to locate the correct inode in the B-tree. A fencepost added between these two updates in `btrfs_add_link` would also break up all of the system calls that call this function, potentially in places where atomicity is required for other data structures.

Our experiments with Racer also confirmed that online analysis allows for much longer test runs than logging. After running the Racer workload for just two minutes, the online algorithm had processed 29 million targeted reads and writes. In offline analysis using the Racer workload, Redflag logging was able to capture only 5.9 million reads and writes before filling its 1GB buffer.

We developed targeted workloads designed to check whether two specified system calls can interfere with each other. Each targeted workload contains two concurrent threads, each making repeated calls to a specified system call, all on the same file. On Wrapfs, we tested simultaneous `write` and simultaneous `truncate` calls to reproduce the `file pos` and `wrapfs_setattr` races discussed in our Lockset results. On Btrfs, we tested an `open` call executing simultaneously with file creation to reproduce the Btrfs inode initialization error described above. In all cases, the online atomicity algorithm reported the violating interleavings that we expected, successfully identifying the threads involved in the violation and pinpointing the three responsible memory operations. Although we manually wrote the programs that implement these targeted workloads, it would be practical to automatically generate similar programs to comprehensively test every pair of system calls on each type of supported file.

Performance To evaluate the performance of our instrumentation and logging, we measured overhead with a micro-benchmark that stresses the logging system by constantly writing to a targeted file system. For this experiment, we stored the file system on a RAM disk to ensure that I/O costs did not hide overhead. This experiment was run on a computer with two 2.8GHz single-core Intel Xeon processors. The instrumentation targeted Btrfs running as part of the 2.6.36-rc3 Linux kernel.

We measured an overhead (i.e., the ratio of the running time of the modified system to the running time of the unmodified system) of 2.44× for an instrumented kernel with logging turned off, and 2.65× with logging turned on. The additional overhead from logging includes storing event data, copying the call stack, and reserving buffer space using atomic memory operations.

We evaluated the performance of our online atomicity checking algorithm on a dual-core 2.4GHz Core 2 processor, targeting the same Btrfs data structures as in our logging test. The overhead in our test was 16.2×. As expected, this is significantly higher than the runtime overhead of offline analysis, but it compares quite favorably with the overhead of other online atomicity checking tools, such as AVIO, which reports 25× overhead [42], and SVD (the Serializability Violation Detector), which reports 65× overhead [69].

Schedule sensitivity of LOA Although LOA is very effective at removing false alarms, it is sensitive to the observed ordering of events, potentially resulting in missed errors, as discussed in Section 2.1.4. We evaluated LOA's sensitivity to event orderings by repeating the Racer workload on Btrfs under different configurations: single-core, dual-core, quad-core, and single-core with kernel preemption disabled. We then analyzed the logs with the single-variable block-based algorithm.

The analysis results were quite stable across these different configurations, even though they generate different schedules. The biggest difference is that the non-preemptible log misses 13 of the 201 violations found in the quad-core log. There were only three violations unique to just one log.

2.5 Related Work

A number of techniques, both runtime and static, exist for tracking down difficult concurrency errors. This section discusses tools from several categories: runtime race detectors, static analyzers, model checkers, and runtime atomicity checkers.

Static data race detection Static analysis tools, typically based on the Lockset approach of identifying variables that lack a consistent locking discipline, have uncovered races even in some large systems. For example, RacerX [24] and RELAY [65] found data races in the Linux kernel. Static race detection tools generally produce more false alarms than runtime tools (such as Redflag), due to the well-known difficulty of accurately and statically analyzing aliasing, function pointers, calling context, etc.

Static atomicity checking Static analysis of atomicity has also been studied [28, 56, 64] but not applied to large systems software. Generally, these analyses check whether the code follows certain safe synchronization patterns and are not completely automatic, relying on user-supplied annotations in the code. Static analysis of atomicity, like static analysis of data races, generally produces many false alarms when applied to large, complex systems.

Runtime data race detection Our Lockset algorithm is based on the Eraser algorithm [57]. Several other variants of Lockset exist, implemented for a variety of languages. To the best of our knowledge, none of these other Lockset-based race detectors have been applied to components of an OS kernel. LOA analysis is the main distinguishing feature of our version. As shown in Section 2.4, LOA analysis is effective at eliminating false alarms not eliminated by other techniques. Some features of other race detectors could be integrated into Redflag: for example, the use of sampling to reduce overhead, at the cost of possibly missing some errors, as in LiteRace [43]. We explore this possibility with the lock discipline monitor discussed in Chapter 4.

Microsoft Research's DataCollider [25] is the only other runtime data race detector that has been applied to components of an OS kernel, to the best of our knowledge. It was applied to several modules in the Windows kernel and detected numerous races. At runtime, DataCollider pauses a thread about to perform a memory access and then uses data breakpoint registers in the CPU to intercept conflicting accesses that occur within the pause interval. Thus, it detects actual data races when they occur, in contrast to Lockset-based tools that analyze synchronization to detect potential races. DataCollider's approach produces no false alarms but may take longer to find races and may miss races that occur only rarely. DataCollider uses sampling to reduce overhead.

Runtime atomicity checking To the best of our knowledge, Redflag is the first runtime atomicity checker that has been applied to components of an OS kernel. Redflag implements two kinds of atomicity checking algorithms: the block-based algorithms to detect potential atomicity violations, and the Redflag algorithm to detect (actual) atomicity violations. In principle, either algorithm could be used online or offline, although the block-based algorithms would incur too much overhead for practical online use. LOA analysis is the main distinguishing feature of our version of the block-based algorithms, compared to the original version [67, 68].

Runtime atomicity checking algorithms based on Lipton's reduction theorem [27, 41, 68] also detect potential atomicity violations. They are computationally much cheaper than the block-based algorithms, but they check a simpler condition that is sufficient but not necessary for ensuring atomicity; hence, they usually produce more false alarms than the block-based algorithms [68]. Algorithms based on Lipton's reduction theorem rely on runtime race detection and hence produce false alarms due to multi-stage escape and other complex order-enforcing synchronization used in the Linux kernel—unless the underlying race-detection algorithm are extended with a technique such as LOA analysis, which takes such synchronization into account.

Redflag's algorithm for detecting atomicity violations is similar to the runtime atomicity checking algorithms in AVIO [42] and CTrigger [52]. All of these algorithms detect only actual—not potential—atomicity violations, by detecting problematic interleavings of a few memory accesses from two threads. AVIO and CTrigger use heuristics to infer programmers' expectations about atomicity, and then check for violations thereof (i.e., atomicity violations). AVIO and CTrigger, unlike Redflag, do not allow instrumentation to be targeted to specific data structures. Their implementations use binary instrumentation and are not integrated with the compiler, so it would be difficult to modify them to provide this feature. In addition, AVIO and CTrigger's algorithms are harder to localize to part of a software system, because their algorithms detect interleaved write violations and interleaved read violations only if the code executed by both of the involved threads is instrumented. In contrast, Redflag's algorithm for detecting interleaved write violations works even if only the code executed by the "interfered-with" thread (i.e., the thread corresponding to thread-1 in Figure 2.1(a)) is instrumented. The only disadvantage of executing the other thread without instrumentation is that Redflag will not provide debugging information about the interleaved write. Redflag's algorithm for detecting interleaved reads does rely on instrumentation of the code executed by each involved thread. This requirement could be eliminated by modifying the algorithm to use data-breakpoint registers in the CPU to intercept interleaved reads; the disadvantage of this approach is that there is a limited number of such registers (e.g., four in the Intel x86 architecture), so sampling would need to be used.

Velodrome [29] detects atomicity violations in executions of Java programs. Velodrome constructs a happens-before relation on events, uses it to construct a happens-before relation on transactions (i.e., executions of atomic regions), and reports an atomicity violation if the happens-before relation on transactions contains a cycle. Velodrome's algorithm is more complicated and expensive than Redflag's online atomicity checking algorithm, but Velodrome's algorithm has the advantage of detecting violations involving multiple variables. Recall that Redflag's online algorithm detects only the single-variable unserializable interleavings shown in Figure 2.1(a).

Logging Feather-Trace uses a lock-free buffer similar to ours to log kernel events [11]. A reader thread simultaneously empties the buffer, storing the log entries to disk. For logging memory accesses, we found that the rate of events was so high that any size buffer would fill too fast for a reader thread to keep up, so Redflag limits logging to a fixed sized buffer and defers all output until after logging is turned off.

2.6 Conclusions

We have described the design of Redflag and shown that it can successfully detect data races and atomicity violations in components of the Linux kernel. To the best of our knowledge, Redflag is the first runtime race detector applied to the Linux kernel and the first runtime atomicity detector applied to components of any OS kernel.

Redflag's offline race detector and offline atomicity checker detect potential concurrency problems even if actual errors occur only in rare schedules not seen during testing. The analyses are based on well-known algorithms but contain a number of extensions to significantly improve the accuracy of our analysis. Redflag automatically identifies variables accessed using bit-wise operations and analyzes them with bit-level granularity, and it filters harmless interleavings involving idempotent operations. Redflag logs RCU synchronization and checks for correct usage of it. Finally, we developed Lexical Object Availability (LOA) analysis, which takes into account order-enforcing synchronization (in contrast to mutual exclusion), including synchronization in complicated initialization code that uses multi-stage escape. LOA significantly reduced the number of false positives in our experiments.

Although the cost of thorough system logging can be high, we have shown that Redflag's performance is sufficient to capture traces that exercise many system calls and execution paths.

We also believe that Redflag is a good demonstration of the usefulness of GCC plug-ins for runtime monitoring. We further explore this potential in the following chapter, which discusses the framework we designed to simplify the process of writing instrumentation plug-ins.

Chapter 3

Compiler-Assisted Instrumentation

GCC is a widely used compiler infrastructure that supports a variety of input languages, e.g., C, C++, Fortran, Java, and Ada, and over 30 different target machine architectures. GCC translates each of its front-end languages into a language-independent intermediate representation called GIMPLE, which then gets translated to machine code for one of GCC's many target architectures. GCC is a large software system with more than 100 developers contributing over the years and a steering committee consisting of 13 experts who strive to maintain its architectural integrity.

In earlier work [13], we extended GCC to support *plug-ins*, allowing users to add their own custom passes to GCC in a modular way without patching and recompiling the GCC source code. Released in April 2010, GCC 4.5 [30] includes plug-in support that is largely based on our design.

GCC's support for plug-ins presents an exciting opportunity for the development of practical, widely-applicable program transformation tools, including program-instrumentation tools for run-time verification. Because plug-ins operate at the level of GIMPLE, a plug-in is applicable to all of GCC's front-end languages. Transformation systems that manipulate machine code may also work for multiple programming languages, but low-level machine code is harder to analyze and lacks the detailed type information that is available in GIMPLE.

Implementing instrumentation tools as GCC plug-ins provides significant benefits but also presents a significant challenge: despite the fact that it is an intermediate representation, GIMPLE is in fact a low-level language, requiring the writing of low-level GIMPLE Abstract Syntax Tree (AST) traversal functions in order to transform one GIMPLE expression into another. Therefore, as GCC is currently configured, the writing of plug-ins is not trivial but for those intimately familiar with GIMPLE's peculiarities.

To address this challenge, we developed the INTERASPECT program-instrumentation framework, which allows instrumentation plug-ins to be developed using the familiar vocabulary of Aspect-Oriented Programming (AOP). INTERASPECT is itself implemented using the GCC plug-in API for manipulating GIMPLE, but it hides the complexity of this API from its users, presenting instead an aspect-oriented API in which instrumentation is accomplished by defining *pointcuts*. A pointcut denotes a set of program points, called *join points*, where calls to *advice functions* can be inserted by a process called *weaving*.

INTERASPECT's API allows users to customize the weaving process by defining *callback functions* that get invoked for each join point. Callback functions have access to specific information about each join point; the callbacks can use this to customize the inserted instrumentation, and to leverage static-analysis results for their customization.

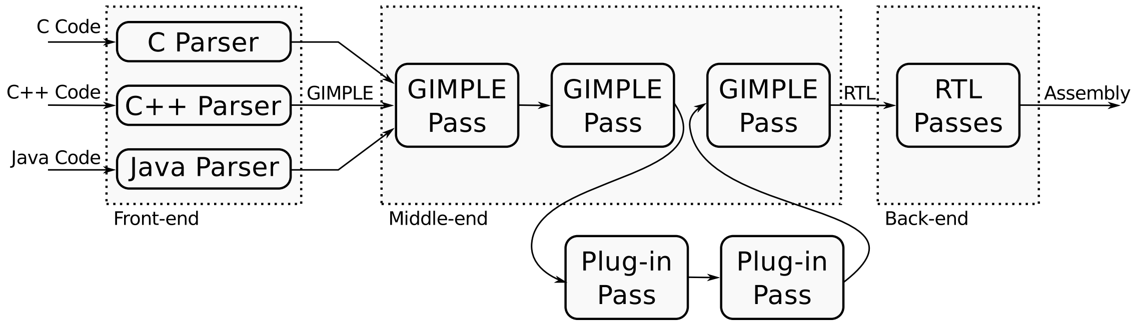


Figure 3.1: A simplified view of the GCC compilation process

We also present the InterAspect *Tracecut extension* to generate program monitors directly from formally specified tracecuts. A tracecut [66] matches *sequences of pointcuts* specified as a regular expression. Given a tracecut specification T, InterAspect Tracecut instruments a target program so that it communicates program events and event parameters directly to a monitoring engine for T. The tracecut extension adds the necessary monitoring instrumentation exclusively with the InterAspect API presented here.

In summary, InterAspect offers the following novel combination of features:

- InterAspect builds on top of GCC, a widely used compiler infrastructure.
- InterAspect exposes an API that encourages and simplifies open-source collaboration.
- InterAspect is versatile enough to provide instrumentation for many purposes, including monitoring a tracecut specification.
- InterAspect has access to GCC internals, which allows one to exploit static analysis and meta-programming during the weaving process.

The full source of the InterAspect framework is available from the InterAspect website under the GPLv3 license [36].

To illustrate InterAspect's practical utility, we have developed a number of example program-instrumentation plug-ins that use InterAspect for custom instrumentation. These include a *heap visualization* plug-in designed for the analysis of JPL Mars Science Laboratory software; an *integer range analysis* plug-in that finds bugs by tracking the range of values for each integer variable; and a *code coverage* plug-in that, given a pointcut and test suite, measures the percentage of join points in the pointcut that are executed by the test suite.

3.1 Overview of GCC and the InterAspect Architecture

Overview of GCC As Figure 3.1 illustrates, GCC translates all of its front-end languages into the GIMPLE intermediate representation for analysis and optimization. Each transformation on GIMPLE code is split into its own *pass*. These passes, some of which may be implemented as *plug-ins*, make up GCC's *middle-end*. Moreover, a plug-in pass may be InterAspect-based, enabling the plug-in to add instrumentation directly into the GIMPLE code. The final middle-end

```
int main() {
    int a, b, c;
    a = 5;
    b = a + 10;
    c = b + foo(a, b);
    if (a > b + c)
        c = b++ / a + (b * a);
    bar(a, b, c);
}
```

=>

```
int main {
    int a, b, c;
    int T1, T2, T3, T4;
    a = 5;
    b = a + 10;
    T1 = foo(a, b);
    c = b + T1;
    T2 = b + c;
    if (a <= T2) goto fi;
        T3 = b / a;
        T4 = b * a;
        c = T3 + T4;
        b = b + 1;
fi:
    bar (a, b, c);
}
```

Figure 3.2: Sample C program (left) and corresponding GIMPLE representation (right)

passes convert the optimized and instrumented GIMPLE to the Register Transfer Language (RTL), which the *back-end* translates to assembly.

GIMPLE is a C-like three-address (`3A`) code. Complex expressions (possibly with side effects) are broken into simple `3A` statements by introducing new, temporary variables. Similarly, complex control statements are broken into simple `3A` (conditional) `goto`s by introducing new labels. Type information is preserved for every operand in each GIMPLE statement.

Figure 3.2 shows a C program and its corresponding GIMPLE code, which preserves source-level information such as data types and procedure calls. Although not shown in the example, GIMPLE types also include pointers and structures.

A disadvantage of working purely at the GIMPLE level is that some language-specific constructs are not visible in GIMPLE code. For example, targeting a specific kind of loop as a pointcut is not currently possible because all loops look the same in GIMPLE. InterAspect can be extended with language-specific pointcuts, whose implementation could hook into one of the language-specific front-end modules instead of the middle-end.

InterAspect architecture InterAspect works by inserting a pass that first traverses the GIMPLE code to identify program points that are join points in a specified pointcut. For each such join point, it then calls a user-provided weaving callback function, which can insert calls to advice functions. Advice functions can be written in any language that will link with the target program, and they can access or modify the target program's state, including its global variables. Advice that needs to maintain additional state can declare static variables and global variables.

Unlike traditional AOP systems which implement a special AOP language to define pointcuts, InterAspect provides a C API for this purpose. We believe that this approach is well suited to open collaboration. Extending InterAspect with new features, such as new kinds of pointcuts, does not require agreement on new language syntax or modification to parser code. Most of the time, collaborators will only need to add new API functions.

The InterAspect Tracecut extension API uses InterAspect to generate program monitors from formally specified tracecuts. Tracecuts match sequences of pointcuts, specified as regular expressions. The instrumentation component of the extension, which is implemented in C, benefits

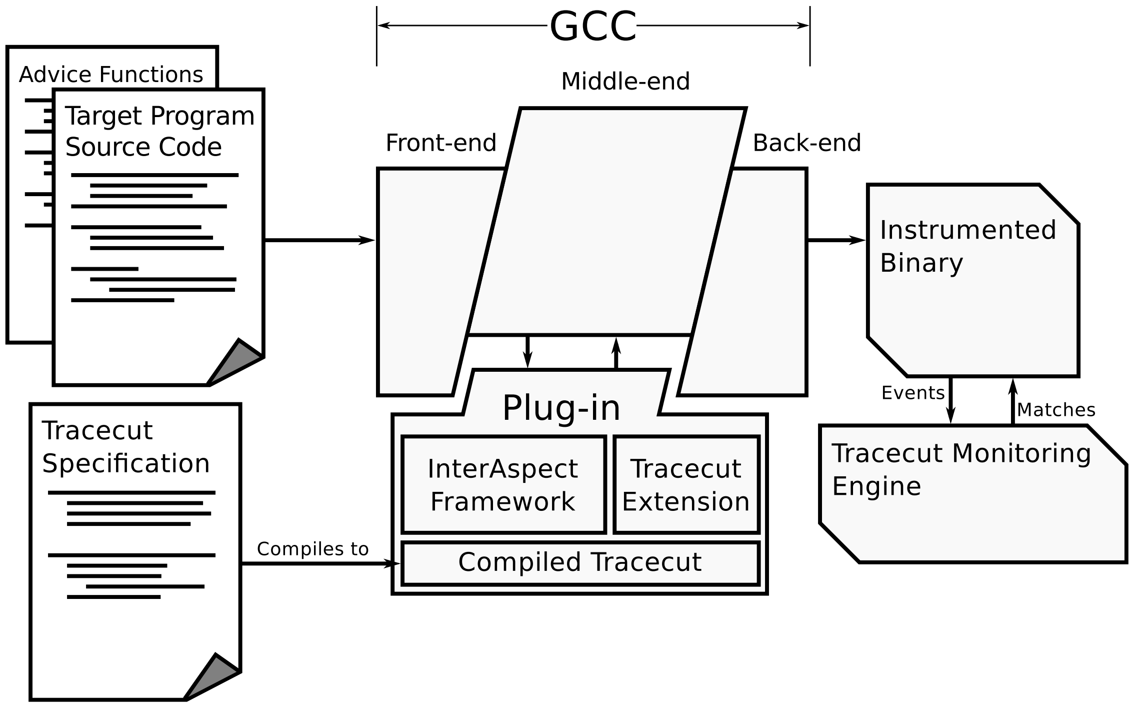


Figure 3.3: Architecture of the INTERASPECT framework with its tracecut extension
The tracecut specification is a simple C program. The tracecut extension translates events in the specification to pointcuts, and the INTERASPECT framework directly instruments the pointcuts using GCC's GIMPLE API. The instrumented binary sends events to the tracecut monitoring engine, and monitors signal matches by calling advice functions, which are compiled alongside the target program. It is also possible to specify just pointcuts, in which case the tracecut extension and monitoring engine are not necessary.

```
struct aop_pointcut *aop_match_function_entry(void);
```

Creates pointcut denoting every function entry point.

```
struct aop_pointcut *aop_match_function_exit(void);
```

Creates pointcut denoting every function return point.

```
struct aop_pointcut *aop_match_function_call(void);
```

Creates pointcut denoting every function call.

```
struct aop_pointcut *aop_match_assignment_by_type(struct aop_type *type);
```

Creates pointcut denoting every assignment to a variable or memory location that matches a type.

Figure 3.4: *Match functions* for creating pointcuts

from INTERASPECT's design as an API: it need only call API functions to define and instrument the pointcuts that are necessary to monitor the tracecut.

Figure 3.3 shows the architecture of a monitor implemented with INTERASPECT Tracecut. The tracecut itself is defined in a short C program that calls the INTERASPECT Tracecut API to specify tracecut properties. Linking the compiled *tracecut program* with INTERASPECT and the tracecut extension produces a plug-in that instruments events relevant to the tracecut. A target program compiled with this plug-in will send events and event parameters to the tracecut monitoring engine, which then determines if any sequence of events matches the tracecut rule. The target program can include tracecut-handling functions so that the monitoring engine can report matches directly back to the program.

3.2 The INTERASPECT API

This section describes the functions in the INTERASPECT API, most of which fall naturally into one of two categories: (1) functions for creating and filtering pointcuts, and (2) functions for examining and instrumenting join points. Note that users of our framework can write plug-ins solely with calls to these API functions; it is not necessary to include any GCC header files or manipulate any GCC data structures directly.

Creating and filtering pointcuts The first step for adding instrumentation in INTERASPECT is to create a pointcut using a *match* function. Our current implementation supports the four match functions given in Figure 3.4, allowing one to create four kinds of pointcuts.

Using a function entry or exit pointcut makes it possible to add instrumentation that runs with every execution of a function. These pointcuts provide a natural way to insert instrumentation at the beginning and end of a function the way one would with before-execution and an after-returning advices in a traditional AOP language. A call pointcut can instead target calls to a function. Call pointcuts can instrument calls to library functions without recompiling them. For example, in Section 3.3.1, a call pointcut is used to intercept all calls to `malloc`.

The assignment pointcut is useful for monitoring changes to program values. For example, we use it in Section 3.3.1 to track pointer values so that we can construct the heap graph. We plan to add several new pointcut types, including pointcuts for conditionals and loops. These new pointcuts will make it possible to trace the complete path of execution as a program runs, which is potentially useful for coverage analysis, profiling, and symbolic execution.

```
void aop_filter_call_pc_by_name(struct aop_pointcut *pc, const char *name);
```

Filter function calls with a given name.

```
void aop_filter_call_pc_by_param_type(struct aop_pointcut *pc, int n,
                                      struct aop_type *type);
```

Filter function calls that have an n^{th} parameter that matches a type.

```
void aop_filter_call_pc_by_return_type(struct aop_pointcut *pc,
                                       struct aop_type *type);
```

Filter function calls with a matching return type.

Figure 3.5: *Filter functions* for refining function-call pointcuts

```
void aop_join_on(struct aop_pointcut *pc, join_callback callback,
                 void *callback_param);
```

Call `callback` on each join point in the pointcut `pc`, passing `callback_param` each time.

Figure 3.6: *Join function* for iterating over a pointcut

After creating a match function, a plug-in can refine it using *filter* functions. Filter functions add additional constraints to a pointcut, removing join points that do not satisfy those constraints. For example, it is possible to filter a call pointcut to include only calls that return a specific type or only calls to a certain function. Figure 3.5 summarizes filter functions for call pointcuts.

Instrumenting join points INTERASPECT plug-ins iterate over the join points of a pointcut by providing an iterator callback to the *join* function, shown in Figure 3.6. For an INTERASPECT plug-in to instrument some or all of the join points in a pointcut, it should join on the pointcut, providing an iterator callback that inserts a call to an *advice* function. INTERASPECT then invokes that callback for each join point.

Callback functions use *capture* functions to examine values associated with a join point. For example, given an assignment join point, a callback can examine the name of the variable being assigned. This type of information is available statically, during the weaving process, so the callback can read it directly with a capture function like `aop_capture_lhs_name`. Callbacks can also capture dynamic values, such as the value on the right-hand side of the assignment, but dynamic values are not available at weave time. Instead, when the callback calls `aop_capture_assigned_value`, it gets an `aop_dynval`, which serves as a weave-time placeholder for the runtime value. The callback cannot read a value from the placeholder, but it can specify it as a parameter to an inserted advice function. When the join point executes (at runtime), the value assigned also gets passed to the advice function. Sections 3.3.1 and 3.3.2 give more examples of capturing values from assignment join points.

Capture functions are specific to the kinds of join points they operate on. Figures 3.7 and 3.8 summarize the capture functions for function-call join points and assignment join points, respectively.

AOP systems like AspectJ [37] provide Boolean operators such as *and* and *or* to refine pointcuts. The INTERASPECT API could be extended with corresponding operators. Even in their absence, a similar result can be achieved in INTERASPECT by including the appropriate logic in the callback. For example, a plug-in can instrument calls to `malloc` *and* calls to `free` by joining on a pointcut with all function calls and using the `aop_capture_function_name` facility to add

```
const char *aop_capture_function_name(aop_joinpoint *jp);
```

Captures the name of the function called in the given join point.

```
struct aop_dynval *aop_capture_param(aop_joinpoint *jp, int n);
```

Captures the value of the n^{th} parameter passed in the given function join point.

```
struct aop_dynval *aop_capture_return_value(aop_joinpoint *jp);
```

Captures the value returned by the function in a given call join point.

Figure 3.7: *Capture functions* for function-call join points

```
const char *aop_capture_lhs_name(aop_joinpoint *jp);
```

Captures the name of a variable assigned to in a given assignment join point, or returns NULL if the join point does not assign to a named variable.

```
enum aop_scope aop_capture_lhs_var_scope(aop_joinpoint *jp);
```

Captures the scope of a variable assigned to in a given assignment join point. Variables can have global, file-local, and function-local scope. If the join point does not assign to a variable, this function returns `AOP_MEMORY_SCOPE`.

```
struct aop_dynval *aop_capture_lhs_addr(aop_joinpoint *jp);
```

Captures the memory address assigned to in a given assignment join point.

```
struct aop_dynval *aop_capture_assigned_value(aop_joinpoint *jp);
```

Captures the assigned value in a given assignment join point.

Figure 3.8: *Capture functions* for assignment join points

advice calls only to `malloc` and `free`. Simple cases like this can furthermore be handled by using regular expressions to match function names, which would be a straightforward addition to the framework.

After capturing, a callback can add an advice-function call before or after the join point using the *insert* function of Figure 3.9. The `aop_insert_advice` function takes any number of parameters to be passed to the advice function at runtime, including values captured from the join point and values computed during instrumentation by the plug-in itself.

Using a callback to iterate over individual join points makes it possible to customize instrumentation at each instrumentation site. A plug-in can capture values about the join point to decide which advice function to call, which parameters to pass to it, or even whether to add advice at all. In Section 3.3.2, this feature is exploited to uniquely index named variables during compilation. Custom instrumentation code in Section 3.3.3 separately records each instrumented join point in order to track coverage information.

Function duplication INTERASPECT provides a *function duplication* facility that makes it possible to toggle instrumentation at the function level. Although inserting advice at the GIMPLE

```
void aop_insert_advice(struct aop_joinpoint *jp, const char *advice_func_name,
                       enum aop_insert_location location, ...);
```

Insert an advice call, before or after a join point (depending on the value of `location`), passing any number of parameters. A plug-in obtains a join point by iterating over a pointcut with `aop_join_on`.

Figure 3.9: *Insert function* for instrumenting a join point with a call to an advice function

level creates very efficient instrumentation, users may still wish to switch between instrumented and uninstrumented code for high-performance applications. Duplication creates two or more copies of a function body (which can later be instrumented differently) and redefines the function to call a special advice function that runs at function entry and decides which copy of the function body to execute.

When joining on a pointcut for a function with a duplicated body, the caller specifies which copy the join should apply to. By only adding instrumentation to one copy of the function body, the plug-in can create a function whose instrumentation can be turned on and off at runtime. Alternatively, a plug-in can create a function that can toggle between different kinds of instrumentation. Section 3.3.2 presents an example of using duplication to reduce overhead by sampling.

3.3 Applications

In this section, we present several example applications of the INTERASPECT API. The plug-ins we designed for these examples provide instrumentation that is tailored to specific problems (memory visualization, integer range analysis, code coverage). Though custom-made, the plug-ins themselves are simple to write, requiring only a small amount of code.

3.3.1 Heap Visualization

The heap visualizer uses the INTERASPECT API to expose memory events that can be used to generate a graphical representation of the heap in real time during program execution. Allocated objects are represented by rectangular nodes, pointer variables and fields by oval nodes, and edges show where pointer variables and fields point.

In order to draw the graph, the heap visualizer needs to intercept object allocations and deallocations and pointer assignments that change edges in the graph. Figure 3.10 shows a prototype of the visualizer using Graphviz [5], an open-source graph layout tool, to draw its output. The graph shows three nodes in a linked list during a bubble-sort operation. The `list` variable is the list's head pointer, and the `curr` and `next` variables are used to traverse the list during each pass of the sorting algorithm. (The `pn` variable is used as temporary storage for swap operations.)

The INTERASPECT code for the heap visualizer instruments each allocation (call to `malloc`) with a call to the `heap_allocation` advice function, and it instruments each pointer assignment with a call to the `pointer_assign` advice function. These advice functions update the graph. Instrumentation of other allocation and deallocation functions, such as `calloc` and `free`, is handled similarly.

The INTERASPECT code in Figure 3.11 instruments calls to `malloc`. The plug-in's main instrumentation function, `instrument_malloc_calls`, constructs a pointcut for all calls to `malloc` and then calls `aop_join_on` to iterate over all the calls in the pointcut. Only a short main function (not shown) is needed to set GCC to invoke `instrument_malloc_calls` during compilation.

The `aop_match_function_call` function constructs an initial pointcut that includes every function call. The `filter` functions narrows the pointcut to include only calls to `malloc`. First, `aop_filter_call_pc_by_name` filters out calls to functions that are not named `malloc`. Then, `aop_filter_pc_by_param_type` and `aop_filter_pc_by_return_type` filter out calls to functions that do not match the standard `malloc` prototype, which takes an unsigned integer as the first

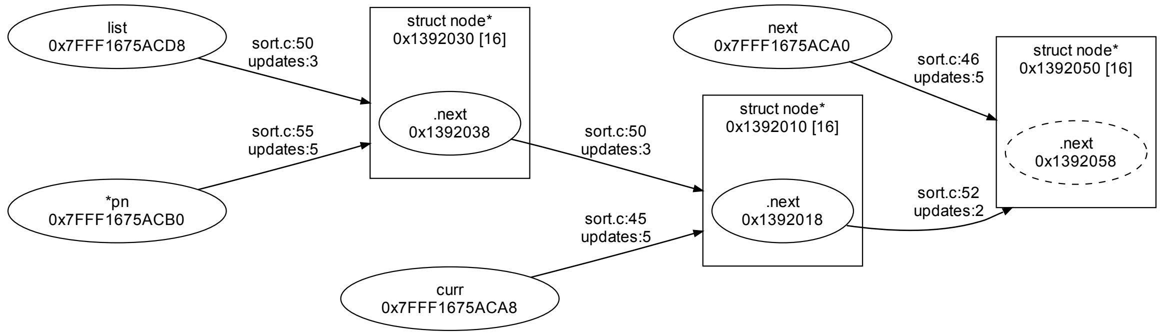


Figure 3.10: Visualization of the heap during a bubble-sort operation on a linked list
Boxes represent heap-allocated `struct`s: linked list nodes in this example. Each `struct` is labeled with is size, its address in memory, and the addresses of its field. Within a `struct`, ovals represent fields that point to other heap objects. Ovals that are not in a `struct` are global and stack variables. Each field and variable has an outgoing edge to the `struct` that it points to, which is labeled with 1) the line number of the assignment that created the edge and 2) the number of assignments to the source variable that have occurred so far. Fields and variables that do not point to valid memory (such as a `NULL` pointer) have dashed borders.

parameter and returns a pointer value. This filtering step is necessary because a program could define its own function with the name `malloc` but a different prototype.

For each join point in the pointcut (in this case, a call to `malloc`), `aop_join_on` makes a call to `malloc_callback`. The two `capture` calls in the callback function return `aop_dynval` objects for the call's first parameter and return value: the size of the allocated region and its address, respectively. Recall from Section 3.2 that an `aop_dynval` serves as a placeholder during compilation for a value that will not be known until runtime. Finally, `aop_insert_advice` adds the call to the advice function, passing the two captured values. Note that INTERASPECT chooses types for these values based on how they were filtered. The filters used here restrict `object_size` to be an unsigned integer and `object_addr` to be some kind of pointer, so INTERASPECT assumes that the advice function `heap_allocation` has the prototype:

```
void heap_allocation(unsigned long long, void *);
```

To support this, INTERASPECT code must generally filter runtime values by type in order to capture and use them.

The INTERASPECT code in Figure 3.12 tracks pointer assignments, such as

```
list_node->next = new_node;
```

The `aop_match_assignment_by_type` function creates a pointcut that matches assignments, which is additionally filtered by the type of assignment. For this application, we are only interested in assignments to pointer variables.

For each assignment join point, `assignment_callback` captures `address`, the address assigned to, and `pointer`, the pointer value that was assigned. In the above examples, these would

```
static void instrument_malloc_calls(void)
{
  /* Construct a pointcut that matches calls to: void *malloc(unsigned int). */
  struct aop_pointcut *pc = aop_match_function_call();
  aop_filter_call_pc_by_name(pc, "malloc");
  aop_filter_call_pc_by_param_type(pc, 0, aop_t_all_unsigned());
  aop_filter_call_pc_by_return_type(pc, aop_t_all_pointer());

  /* Visit every statement in the pointcut. */
  aop_join_on(pc, malloc_callback, NULL);
}

/* The malloc_callback() function executes once for each call to malloc() in the
   target program.  It instruments each call it sees with a call to
   heap_allocation(). */
static void malloc_callback(struct aop_joinpoint *jp, void *arg)
{
  struct aop_dynval *object_size;
  struct aop_dynval *object_addr;

  /* Capture the size of the allocated object and the address it is
     allocated to. */
  object_size = aop_capture_param(jp, 0);
  object_addr = aop_capture_return_value(jp);

  /* Add a call to the advice function, passing the size and address as
     parameters.  (AOP_TERM_ARG is necessary to terminate the list of arguments
     because of the way C varargs functions work.) */
  aop_insert_advice(jp, "heap_allocation", AOP_INSERT_AFTER,
                    AOP_DYNVAL(object_size), AOP_DYNVAL(object_addr),
                    AOP_TERM_ARG);
}
```

Figure 3.11: Instrumenting all memory-allocation events

be the values of `&list_node->next` and `new_node`, respectively. The visualizer uses `address` to determine the source of a new graph edge and `pointer` to determine its destination.

The function that captures `address`, `aop_capture_lhs_addr`, does not require explicit filtering to restrict the type of the captured value because an address always has a pointer type. The value captured by `aop_capture_assigned_value` and stored in `pointer` has a void pointer type because we filtered the pointcut to include only pointer assignments. As a result, INTERASPECT assumes that the `pointer_assign` advice function has the prototype:

```
void pointer_assign(void *, void *);
```

3.3.2 Integer Range Analysis

Integer range analysis is a runtime tool for finding anomalies in program behavior by tracking the range of values for each integer variable [26]. A range analyzer can learn normal ranges from training runs over known good inputs. Values that fall outside of normal ranges in future runs are reported as anomalies, which can indicate errors. For example, an out-of-range value for a variable used as an array index may cause an array-bounds violation.

Our integer range analyzer uses sampling to reduce runtime overhead. Missed updates because of sampling can result in underestimating a variable's range, but this trade-off is reasonable in

```
static void instrument_pointer_assignments(void)
{
  /* Construct a pointcut that matches all assignments to a pointer. */
  struct aop_pointcut *pc = aop_match_assignment_by_type(aop_t_all_pointer());

  /* Visit every statement in the pointcut. */
  aop_join_on(pc, assignment_callback, NULL);
}

/* The assignment_callback function executes once for each pointer assignment.
   It instruments each assignment it sees with a call to pointer_assign(). */
static void assignment_callback(struct aop_joinpoint *jp, void *arg)
{
  struct aop_dynval *address;
  struct aop_dynval *pointer;

  /* Capture the address the pointer is assigned to, as well as the pointer
     address itself. */
  address = aop_capture_lhs_addr(jp);
  pointer = aop_capture_assigned_value(jp);

  aop_insert_advice(jp, "pointer_assign", AOP_INSERT_AFTER,
                    AOP_DYNVAL(address), AOP_DYNVAL(pointer),
                    AOP_TERM_ARG);
}
```

Figure 3.12: Instrumenting all pointer assignments

many cases. Sampling can be done randomly or by using a technique like Software Monitoring with Controllable Overhead [34].

INTERASPECT provides function-body duplication as a means to add instrumentation that can be toggled on and off. Duplicating a function splits its body into two copies. A *distributor block* at the beginning of the function decides which copy to run. An INTERASPECT plug-in can add advice to just one of the copies, so that the distributor chooses between enabling or disabling instrumentation.

Figure 3.13 shows how we use INTERASPECT to instrument integer variable updates. The call to `aop_duplicate` makes a copy of each function body. The first argument specifies that there should be two copies of the function body, and the second specifies the name of a function that the distributor will call to decide which copy to execute. When the duplicated function runs, the distributor calls `distributor_func`, which must be a function that returns an integer. The duplicated function bodies are indexed from zero, and the `distributor_func` return value determines which one the distributor transfers control to.

Using `aop_join_on_copy` instead of the usual `aop_join_on` iterates only over join points in the specified copy of duplicate code. As a result, only one copy is instrumented; the other copy remains unmodified.

The callback function itself is similar to the callbacks we used in Section 3.3.1. The main difference is the call to `get_index_from_name` that converts the variable name to an integer index. The `get_index_from_name` function (not shown for brevity) also takes the variable's scope so that it can assign different indices to local variables in different functions. It would be possible to directly pass the name itself (as a string) to the advice function, but the advice function would then incur the cost of looking up the variable by its name at runtime. This optimization illustrates the benefits of INTERASPECT's callback-based approach to custom instrumentation.

```
static void instrument_integer_assignments(void)
{
  struct aop_pointcut *pc;

  /* Duplicate the function body so there are two copies. */
  aop_duplicate(2, "distributor_func", AOP_TERM_ARG);

  /* Construct a pointcut that matches all assignments to an integer. */
  pc = aop_match_assignment_by_type(aop_t_all_signed_integer());

  /* Visit every statement in the pointcut. */
  aop_join_on_copy(pc, 1, assignment_callback, NULL);
}

/* The assignment_callback function executes once for each integer assignment.
   It instruments each assignment it sees with a call to int_assign(). */
static void assignment_callback(struct aop_joinpoint *jp, void *arg)
{
  const char *variable_name;
  int variable_index;
  struct aop_dynval *value;
  enum aop_scope scope;

  variable_name = aop_capture_lhs_name(jp);

  if (variable_name != NULL) {
    /* Choose an index number for this variable. */
    scope = aop_capture_lhs_var_scope(jp);
    variable_index = get_index_from_name(variable_name, scope);

    aop_insert_advice(jp, "int_assign", AOP_INSERT_AFTER,
                      AOP_INT_CST(variable_index), AOP_DYNVAL(value),
                      AOP_TERM_ARG);
  }
}
```

Figure 3.13: Instrumenting integer variable updates

The `aop_capture_lhs_name` function returns a string instead of an `aop_dynval` object because variable names are known at compile time. It is necessary to check for a `NULL` return value because not all assignments are to named variables.

To better understand InterAspect's performance impact, we benchmarked this plug-in on the compute-intensive `bzip2` compression utility using empty advice functions. The `bzip2` package is a popular tool included in most Linux distributions. It has 137 functions in about 8,000 lines of code. The instrumented `bzip2` contains advice calls at every integer variable assignment, but the advice functions themselves do nothing, allowing us to measure the overhead from calling advice functions independently from actual monitoring overhead. With a distributor that maximizes overhead by always choosing the instrumented function body, we measured 24% runtime overhead. Function duplication by itself contributes very little to this overhead; when the distributor always chooses the uninstrumented path, the overhead from instrumentation was statistically insignificant.

3.3.3 Code Coverage

A straightforward way to measure code coverage is to choose a pointcut and measure the percentage of its join points that are executed during testing. INTERASPECT's ability to iterate over each join point makes it simple to label join points and then track them at runtime.

```
static void instrument_function_entry_exit(void)
{
  struct aop_pointcut *entry_pc;
  struct aop_pointcut *exit_pc;

  /* Construct two pointcuts: one for function entry and one for function exit. */
  entry_pc = aop_match_function_entry();
  exit_pc = aop_match_function_exit();

  aop_join_on(entry_pc, entry_exit_callback, NULL);
  aop_join_on(exit_pc, entry_exit_callback, NULL);
}

/* The entry_exit_callback function assigns an index to every join
   point it sees and saves that index to disk. */
static void entry_exit_callback(struct aop_joinpoint *jp, void *arg)
{
  int index, line_number;
  const char *filename;

  index = choose_unique_index();
  filename = aop_capture_filename(jp);
  line_number = aop_capture_lineno(jp);

  save_index_to_disk(index, filename, line_number);

  aop_insert_advice(jp, "mark_as_covered", AOP_INSERT_BEFORE,
                    AOP_INT_CST(index), AOP_TERM_ARG);
}
```

Figure 3.14: Instrumenting function entry and exit for code coverage

The example in Figure 3.14 adds instrumentation to track coverage of function entry and exit points. To reduce runtime overhead, the `choose_unique_index` function assigns an integer index to each tracked join point, similar to the indexing of integer variables in Section 3.3.2. Each index is saved along with its corresponding source filename and line number by the `save_index_to_disk` function. The runtime advice needs to output only the set of covered index numbers; an offline tool uses that output to compute the percentage of join points covered or to list the filenames and line numbers of covered join points. For brevity we omit the actual implementations of `choose_unique_index` and `save_index_to_disk`.

3.4 Tracecuts

In this section, we present the API for the INTERASPECT Tracecut extension, and discuss the implementation of the associated tracecut monitoring engine. We also present two illustrative examples of the Tracecut extension: runtime verification of file access and GCC vectors.

Our INTERASPECT Tracecut extension showcases the flexibility of INTERASPECT's API. Since one of our goals for this extension is to serve as a more powerful example of how to use INTERASPECT, its instrumentation component is built modularly on INTERASPECT: all of its access to GCC are through the published INTERASPECT interface.

```
struct tc_tracecut *tc_create_tracecut(void);
```

Create an empty tracecut.

```
enum tc_error tc_add_param(struct tc_tracecut *tc, const char *name,
                           const struct aop_type *type);
```

Add a named parameter to a tracecut.

Figure 3.15: Function for initializing tracecuts

Whereas pointcut advice is triggered by individual events, tracecut advice can be triggered by sequences of events matching a pattern [66]. A tracecut in our system is defined by a set symbols, each representing a possibly parameterized runtime event, and one or more rules expressed as regular expressions over these symbols. For example, a tracecut that matches a call to `exit` or `execve` after a fork would specify symbols for `fork`, `exit`, and `execve` function calls and the rule `fork` (`exit` | `execve`), where juxtaposition denotes sequencing, parentheses are used for grouping, and the vertical bar "|" separates alternatives.

Each symbol is translated to a function-call pointcut, which is instrumented with advice that sends the symbol's corresponding event to the monitoring engine. The monitoring engine signals a match whenever some suffix of the string of events matches one of the regular-expression rules.

Parameterization allows a tracecut to separately monitor multiple objects [2, 14]. For example, the rule `fclose fread`, designed to catch an illegal read from a closed file, should not match an `fclose` followed by an `fread` to a different file. When these events are parameterized by the file they operate on, the monitoring engine creates a unique monitor instance for each file.

A tracecut with multiple parameters can monitor properties on sets of objects. A classic example monitors data sources that have multiple iterators associated with them. When a data source is updated, its existing iterators become invalid, and any future access to them is an error. Parameterizing events by both data source and iterator creates a monitor instance for each pair of data source and iterator.

The monitoring engine is implemented as a runtime library that creates monitor instances and forwards events to their matching monitor instances. Because rules are specified as regular expressions, each monitor instance stores a state in the equivalent finite-state machine. The user only has to link the monitoring library with the instrumented binary, and the tracecut instrumentation calls directly into the library.

3.4.1 Tracecut API

A tracecut is specified by a C program that calls tracecut API functions. This design keeps the tracecut extension simple, eliminating the need for a custom parser but still allowing concise definitions. A tracecut specification can define any number of tracecuts, each with its own parameters, events, and rules.

Defining Parameters The functions in Figure 3.15 create a new tracecut and define its parameters. Each parameter has a name and a type. The type is necessary because parameters are used to capture runtime values.

```
enum tc_error tc_add_call_symbol(struct tc_tracecut *tc, const char *name,
                                 const char *func_name,
                                 enum aop_insert_location location);
```

Define a named event corresponding to calls to the function named by `func_name`.

```
enum tc_error tc_bind_to_call_param(struct tc_tracecut* tc,
                                    const char *param_name,
                                    const char *symbol_name, int call_param_index);
```

Bind a function call parameter from an event to one of the tracecut's named parameters.

```
enum tc_error tc_bind_to_return_value(struct tc_tracecut *tc,
                                      const char *param_name,
                                      const char *symbol_name);
```

Bind the return value of an event to one of the tracecut's named parameters.

```
enum tc_error tc_declare_call_symbol(struct tc_tracecut *tc, const char *name,
                                     const char *declaration,
                                     enum aop_insert_location location);
```

Define a named event along with all its parameter bindings with one declaration string.

Figure 3.16: Functions for specifying symbols

Defining Symbols The `tc_add_call_symbol` function adds a new symbol that corresponds to an event at every call to a specified function. The `tc_bind` functions bind a tracecut parameter to one of the function call's parameters or to its return value. Figure 3.16 shows `tc_add_call_symbol` and the `tc_bind` functions.

The tracecut API uses the symbol and its bindings to define a pointcut. Figure 3.17 shows an example symbol along with the INTERASPECT API calls that Tracecut makes to create the pointcut. In a later step, Tracecut makes calls needed to capture the bound return value and pass it to an advice function.

As a convenience, the API also provides the `tc_declare_call_symbol` function (also in Figure 3.16), which can define a symbol and its parameter bindings with one call using a simple text declaration. The declaration is syntactically similar to the C prototype for the function that will trigger the symbol, but the function's formal parameters are replaced with tracecut parameter names or with a question mark "`?`" to indicate that a parameter should remain unbound. The code in Figure 3.17(c) defines the same symbol as in Figure 3.17(a).

Defining Rules After symbols and their parameter bindings are defined, rules are expressed as strings containing symbol names and standard regular expression operators: `(`, `)`, `*`, `+`, and |. The function for adding a rule to a tracecut is shown in Figure 3.18.

3.4.2 Monitor Implementation

The monitoring engine maintains a list of monitor instances for each tracecut. Each instance has a value for each tracecut parameter and a monitor state. Instrumented events pass the values of their parameters to the monitoring engine, which then determines which monitor instances to update. This monitor design is based on the way properties are monitored in Tracematches [2] and MOP [14].

```
struct tracecut *tc = tc_create_tracecut()
tc_add_param(tc, "object", aop_all_pointer());
tc_add_call_symbol(tc, "create", "create_object", AOP_INSERT_AFTER);
tc_bind_to_return_value(tc, "object", "create");
```

(a) Code to define a tracecut symbol.

```
pc = aop_match_function_call();
aop_filter_call_pc_by_name(pc, "create_object");
aop_filter_call_pc_by_return_type(pc, aop_all_pointer());
```

(b) The values that the tracecut API will pass to INTERASPECT functions to create a corresponding pointcut.

```
struct tracecut *tc = tc_create_tracecut()
tc_add_param(tc, "object", aop_all_pointer());
tc_declare_call_symbol(tc, "create", "(object)create_object()",
                       AOP_INSERT_AFTER);
```

(c) A more compact way to define the event in Figure 3.17(a).

Figure 3.17: An example of how the tracecut API translates a tracecut symbol into a pointcut Because the `create` symbol's return value is bound to the `object` param, the resulting pointcut is filtered to ensure that its return value matches the type of `object`.

```
enum tc_error tc_add_rule(struct tc_tracecut *tc, const char *specification);
```

Define a tracecut rule. The specification is a regular expression using symbol names as its alphabet.

Figure 3.18: Function for defining a tracecut rule

When a symbol is fully parameterized—it has a binding for every parameter defined in the tracecut specification—the monitoring engine updates exactly one instance. If no instance exists with matching parameter values, one is created.

For partially parameterized symbols, like `push` in Figure 3.21, the monitoring engine only requires the specified parameters to match. As a result, events corresponding to these symbols can update multiple monitor instances. For example, a `push` event updates one monitor for every `element_pointer` associated with the updated vector. As in the original MOP implementation, partially parameterized symbols cannot create a new monitor instance [14]. (MOP has since defined semantics for partially parameterized monitors [45].)

When any monitor instance reaches an accepting state, the monitoring engine reports a match. The default match function prints the monitor parameters to `stderr`. Developers can implement their own tracecut advice by overriding the default match function. Function overriding is possible in C using a linker feature called *weak linkage*. Placing a debugger breakpoint at the match function makes it possible to examine program state when a match occurs.

Monitoring instances get destroyed when they can no long reach an accepting state. The tracecut engine does not attempt to free instances parameterized by freed objects because it is not always possible to learn when an object is freed in C and because parameters are not required to be pointers to heap-allocated objects.

A developer can ensure that stale monitor instances do not waste memory by designing the rule to discard them. The easiest way to do this is to define a symbol for the function that deallocates an object but not to include the symbol anywhere in the tracecut's rule. Deallocating the object then generates an event that makes it impossible for the tracecut rules to match.

3.4.3 Verifying File Access

As a first example of the tracecut API, we consider the runtime verification of file access. Like most resources in C, the FILE objects used for file I/O must be managed manually. Any access to a FILE object after the file has been closed is a memory error which, though dangerous, might not manifest itself as incorrect behavior during testing. Designing a tracecut to detect these errors is straightforward.

```
tc = tc_create_tracecut();

tc_add_param(tc, "file", aop_t_all_pointer());

tc_declare_call_symbol(tc, "open", "(file)fopen()", AOP_INSERT_AFTER);
tc_declare_call_symbol(tc, "read", "fread(?, ?, ?, file)", AOP_INSERT_BEFORE);
tc_declare_call_symbol(tc, "read_char", "fgetc(file)", AOP_INSERT_BEFORE);
tc_declare_call_symbol(tc, "close", "fclose(file)", AOP_INSERT_BEFORE);

tc_add_rule(tc, "open (read | read_char)* close (read | read_char)");
```

Figure 3.19: A tracecut for catching accesses to closed files
For brevity, the tracecut only checks read operations.

The tracecut in Figure 3.19 defines symbols for four FILE operations: open, close, and two kinds of reads. The rule matches any sequence of these symbols that opens a file, closes it, and then tries to read it.

The rule matches as soon as any read is performed on a closed FILE object, immediately identifying the offending read. We tested this tracecut on bzip2 (which we also use for evaluation in Section 3.3.2); it caught an error we planted without reporting any false positives.

3.4.4 Verifying GCC Vectors

We designed a tracecut to monitor a property on a vector data structure used within GCC to store an ordered list of GIMPLE statements. The list is stored in a dynamically resized array. The vector API provides an iterator function to iterate over the GIMPLE statements in a vector. Figure 3.20 shows how the iterator function is used. At each execution of the loop, the element variable points to the next statement in the vector.

A common tracecut property for data structures with iterators checks that the data structure is not modified while it is being iterated, as can occur in Figure 3.20. Figure 3.21 specifies a tracecut that detects violations of this property.

The tracecut monitors two important vector operations: the VEC_gimple_base_iterate function, which is used in the guard of a for loop to advance to the next element in the list, and the VEC_gimple_base_quick_push function, which inserts a new element at the end of a vector. With the symbols defined, the rule itself is simple: iterate push iterate. Any push in between two iterate operations indicates that the vector was updated within the iterator loop.

Parameterizing the iterate symbol on both the vector and the element_pointer used to iterate makes it possible to distinguish different iterator loops over the same vector. This distinction is necessary so that a program that finishes iterating over a vector, updates that vector, and then iterates over it again does not trigger a match. Though, the tracecut monitor will observe events

```
int i;
gimple element;

/* Iterate over each element in a vector of GIMPLE statements. */
for (i = 0; VEC_gimple_base_iterate(vector1, i, &element); i++) {
  /* If condition holds, copy this element into vector2. */
  if (condition(element))
    VEC_gimple_base_quick_push(vector2, element);
}
```

Figure 3.20: Standard pattern for iterating over elements in a GCC vector of GIMPLE statements This example copies elements matching some condition from `vector1` to `vector2`. If `vector1` and `vector2` happen to point to the same vector, this code may modify that vector while iterating over its elements.

```
tc = tc_create_tracecut();

tc_add_param(tc, "vector", aop_t_all_pointer ());
tc_add_param(tc, "element_pointer", aop_t_all_pointer ());

tc_declare_call_symbol(tc, "iterate",
                       "VEC_gimple_base_iterate(vector, ?, element_pointer)",
                       AOP_INSERT_BEFORE);
tc_declare_call_symbol(tc, "push", "VEC_gimple_base_quick_push(vector, ?)",
                       AOP_INSERT_BEFORE);

tc_add_rule(tc, "iterate push iterate");
```

Figure 3.21: A tracecut to monitor vectors of GIMPLE objects in GCC

for the symbols `iterate push iterate`, the first and last `iterate` events (which are from different loops) will normally have different values for their `element_pointer` parameter.

When monitoring this same property in Java, usually an *iterator object* serves the purpose of parameterizing an iterator loop. In Figure 3.20, the `element` variable is analogous to an iterator, as it provides access to the current list element at each iteration of the loop. The `element_pointer` identifies the iterator-like variable by its address.

Keeping specifications simple is especially important in C because the language does not provide any standard data structures. A tracecut written for one program's vector type is not likely to be useful for monitoring any other program.

We applied the tracecut in Figure 3.21 to GCC itself, verifying that, in our tests, GCC did not update any vectors while they were being iterated. The tracecut did match a synthetic error that we added for testing, in the form of a call to `VEC_gimple_base_quick_push` that we deliberately placed in an iterator loop.

3.5 Related Work

Aspect-oriented programming was first popularized for Java with AspectJ [22,37]. There, weaving takes place at the bytecode level. The AspectBench Compiler (abc) [6] is a more recent extensible research version of AspectJ that makes it possible to add new language constructs [10]. Similarly to INTERASPECT, it manipulates a 3A intermediate representation (Jimple) specialized to Java.

Other frameworks for Java, including Javaassist [16] and PROSE [49], offer an API for instrumenting and modifying code, and hence do not require the use of a special language. Javaassist is a class library for editing bytecode. A source-level API can be used to edit class files without knowledge of the bytecode format. PROSE has similar goals.

AOP for other languages such as C and C++ has had a slower uptake. AspectC [17] was one of the first AOP systems for C, based on the language constructs of AspectJ. ACC [46] is a more recent AOP system for C, also based on the language constructs of AspectJ. It transforms source code and offers its own internal compiler framework for parsing C. It is a closed system in the sense that one cannot augment it with new pointcuts or access the internal structure of a C program in order to perform static analysis.

The XWeaver system [55], with its language AspectX, represents a program in XML (srcML, to be specific), making it language-independent. It supports Java and C++ . A user, however, has to be XML-aware. Aspicere [53] is an aspect language for C based on LLVM bytecode. Its pointcut language is inspired by logic programming. Adding new pointcuts amounts to defining new logic predicates. Arachne [21, 23] is a dynamic aspect language for C that uses assembler manipulation techniques to instrument a running system without pausing it.

AspectC++ [60] is targeted towards C++. It can handle C to some extent, but this does not seem to be a high priority for its developers. For example, it only handles ANSI C and not other dialects. AspectC++ operates at the source-code level and generates C++ code, which can be problematic in contexts where only C code is permitted, such as in certain embedded applications. OpenC++ [15] is a front-end library for C++ that developers can use to implement various kinds of translations in order to define new syntax and object behavior. CIL [48] (C Intermediate Language) is an OCaml [35] API for writing source-code transformations of its own 3A code representation of C programs. CIL requires a user to be familiar with the less-often-used yet powerful OCaml programming language.

Additionally, various low-level but mature tools exist for code analysis and instrumentation. These include the BCEL [3] bytecode-instrumentation tool for Java, and Valgrind [63], which works directly with executables and consequently targets multiple programming languages.

INTERASPECT Tracecut is informed by several runtime monitoring systems, including Declarative Event Patterns [66], which introduced the term *tracecut*. Monitor parameterization is based on the monitor implementations in Tracematches [2] and MOP [14]. These three systems are designed to monitor Java programs. For C, Arachne and Aspicere provide tracecut-style monitoring. Arachne can monitor pointcut *sequences* which have similar semantics to INTERASPECT Tracecut's regular expressions [21]. The cHALO extension to Aspicere adds predicates for defining sequences [1]. These predicates are designed to give developers better control over the amount of memory used to track monitor instances. Using the INTERASPECT API for our tracecut monitoring greatly simplified its design, which we believe makes a case for the extensibility of the tracecut API.

3.6 Conclusions

We have presented INTERASPECT, a framework for developing powerful instrumentation plug-ins for the GCC suite of production compilers. INTERASPECT-based plug-ins instrument programs compiled with GCC by modifying GCC's intermediate language, GIMPLE. The INTERASPECT API simplifies this process by offering an AOP-based interface. Plug-in developers can easily specify pointcuts to target specific program join points and then add customized instrumentation at those join points. We presented several example plug-ins that demonstrate the framework's ability to customize runtime instrumentation for specific applications. Finally, we developed a more full-featured application of our API: the INTERASPECT Tracecut extension, which monitors formally defined runtime properties. The API and the tracecut extension are available under an open-source license [36]. To that we also intend to add the source code for our Redflag system, discussed in the previous chapter.

As future work, we plan to add pointcuts for all control flow constructs, thereby allowing instrumentation to trace a program run's exact path of execution. We also plan to investigate API support for pointcuts that depend on dynamic information, such as AspectJ's `cflow`. Dynamic pointcuts can already be implemented in INTERASPECT with advice functions that maintain and use appropriate state, or even with tracecut advice, but API support would eliminate the need to write such advice functions.

Chapter 4

Adaptive Runtime Verification

4.1 Introduction

In [61], we introduced the concept of *runtime verification with state estimation* (RVSE), and showed how it can be used to estimate the probability that a temporal property is satisfied by a partially or incompletely monitored program run. In such situations, there may be *gaps* in observed program executions, making accurate estimation challenging.

Incomplete monitoring can arise from a variety of sources. For example, in real-time embedded systems, the sensors might have intrinsically limited fidelity, or the scheduler might skip monitoring of internal or external events due to an impending deadline for a higher-priority task. Incomplete monitoring also arises from overhead control frameworks, such as [34], which repeatedly disable and enable monitoring of selected events, to maintain the overall overhead of runtime monitoring at a specified target level. Regardless of the cause, simply ignoring the fact that unmonitored events might have occurred gives poor results.

The main idea behind RVSE is to use a statistical model of the monitored system, in the form of a Hidden Markov Model (HMM), to "fill in" gaps in event sequences. We then use an extended version of the forward algorithm of [54] to calculate the probability that the property is satisfied. The HMM can be learned automatically from training runs, using standard algorithms [54].

When the cause of incomplete monitoring is overhead control, a delicate interplay exists between RVSE and overhead control, due to the runtime overhead of RVSE itself: the matrix-vector calculations performed by the RVSE algorithm to process an observation symbol—which can be a program event or a gap symbol paired with a discrete probability distribution describing the length of the gap—are expensive. Note that we did not consider this interplay in [61], because the RVSE calculations were performed post-mortem in the experiments described there.

The relationship between RVSE and overhead control can be viewed as an accuracy-overhead trade-off: the more overhead RVSE consumes processing an observation symbol, with the goal of performing more accurate state estimation, the more events are missed (because less overhead is available). Paradoxically, these extra missed events result in more gap symbols, making accurate state estimation all the more challenging.

This tension between accurate state estimation and overhead control can be understood in terms of Heisenberg's uncertainty principle, which essentially states that the more accurately one measures the position of an electron, the more its velocity is perturbed, and vice versa. In the case

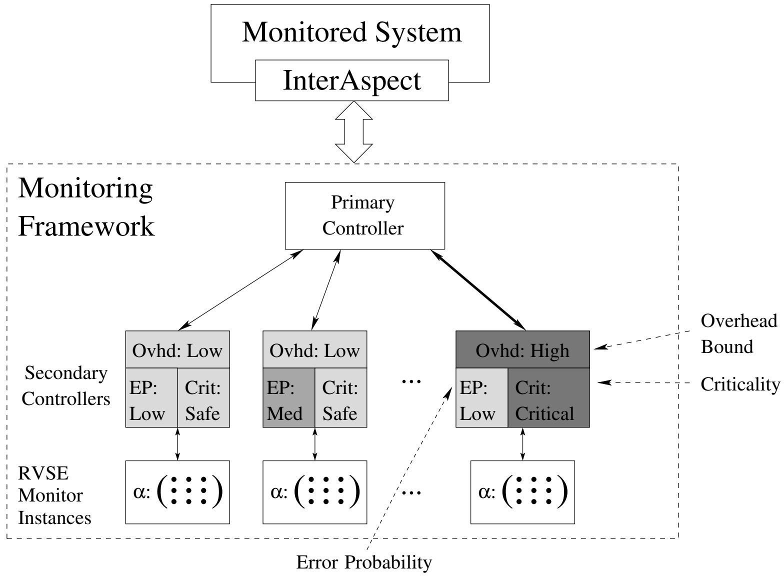


Figure 4.1: The Adaptive Runtime Verification Framework

of RVSE, we are estimating the position (state) and velocity (execution time) of a "computation particle" (program counter) flowing through an instrumented program.

With these concerns in mind, we present *Adaptive Runtime Verification* (ARV), a new approach to runtime verification in which overhead control, runtime verification with state estimation, and predictive analysis are synergistically combined. In ARV, as depicted in Figure 4.1, each monitor instance[1] has an associated *criticality level*, which is a measure of how "close" the instance is to violating the property under investigation. As criticality levels of monitor instances rise, so will the fraction of monitoring resources allocated to these instances, thereby increasing the probability of violation detection and concomitant adaptive responses to property violations.

The main contributions discussed in this chapter are:

- In ARV, the overhead-control subsystem and the RVSE-enabled monitoring subsystem are coupled together in a feedback control loop: overhead control introduces gaps in event sequences, whose resolution requires HMM-based state estimation (RVSE); state estimation informs overhead control, closing the loop. Up-to-date state estimates enable the overhead-control subsystem to make intelligent, criticality-based decisions about how to allocate the available overhead among monitor instances.

- A key aspect of the ARV framework is a new algorithm for RVSE that performs the calculations offline (in advance), dramatically reducing the runtime overhead of RVSE, at the cost of introducing some approximation error. We analyze the cumulative approximation error incurred by this algorithm.

[1] A *monitor instance* is a runtime instance of a parameterized monitor. For example, our monitor for concurrency errors in the Linux kernel is parameterized by the id (address) of the structure being monitored.

- To compute the criticality levels of monitor instances, the ARV framework performs reward-based reachability queries over the Discrete Time Markov Chain (DTMC) derived from the composition of the HMM model of the monitored program and the monitor, represented as a Deterministic Finite State Machine (DFSM). These queries determine the expected distance to the monitor's error state. These queries are also computed in advance, and the results are stored in a data structure.
- We demonstrate the utility of the ARV approach on a significant case study involving runtime monitoring of concurrency errors in the Linux kernel.

4.2 Background

Hidden Markov Models (HMMs) An HMM [54] is a tuple $H = \langle S, A, V, B, \pi \rangle$ containing a set S of states, a transition probability distribution A, a set V of observation symbols (also called "outputs"), an observation probability distribution B, and an initial state distribution π. The states and observations are indexed (i.e., numbered), so S and V can be written as $S = \{s_1, s_2, \ldots, s_{N_s}\}$ and $V = \{v_1, \ldots, v_{N_o}\}$, where N_s is the number of states, and N_o is the number of observation symbols. Let $\Pr(c_1 \mid c_2)$ denote the probability that c_1 holds, given that c_2 holds. The transition probability distribution A is an $N_s \times N_s$ matrix indexed by states in both dimensions, such that $A_{i,j} = \Pr(\text{state is } s_j \text{ at time } t+1 \mid \text{ state is } s_i \text{ at time } t)$. The observation probability distribution B is an $N_s \times N_o$ matrix indexed by states and observations, such that $B_{i,j} = \Pr(v_j \text{ is observed at time } t \mid \text{ state is } s_i \text{ at time } t)$. Following tradition, we define $b_i(v_k) = B_{i,k}$. Prior distribution π_i is the probability that the initial state is s_i.

An example of an HMM is depicted in Figure 4.3 a). Each state is labeled with observation probabilities in that state; for example, `P(LOCK)`=0.99 in state s_1 means $B_{1,LOCK} = 0.99$. Edges are labeled with transition probabilities; for example, 0.20 on the edge from s_2 to s_3 means $A_{2,3} = 0.20$.

Learning HMMs Given a set of traces of a system and a desired number of states of the HMM, it is possible to learn an HMM model of the system using standard algorithms [54]. The main idea behind these algorithms is to maximize the probability that the HMM generates the given traces. In our experiments, we chose an HMM model with three states, used the Baum-Welch learning algorithm [8], and provided the learning algorithm with 1,000 traces as input. Figure 4.3 a) depicts the transition and observation probability distributions of the resulting HMM model. The related case study (Section 4.6) provides further details.

Deterministic Finite State Machines (DFSMs) We assume that the temporal property ϕ to be monitored is expressed as a parametrized deterministic finite state machine. A DFSM is a tuple $M = \langle S_M, m_{init}, V, \delta, F \rangle$, where S_M is the set of states, m_{init} in S_M is the initial state, V is the alphabet (also called the set of input symbols), $\delta : S_M \times V \rightarrow S_M$ is the transition function, and F is the set of accepting states (also called "final states"). Note that δ is a total function. A trace O satisfies the property iff it leaves M in an accepting state.

$$
\begin{aligned}
p_i(m,n) &= \sum_{v \in V \text{ s.t. } \delta(m,v)=n} b_i(v) && (4.1)\\
g_0(i,m,j,n) &= (i = j \wedge m = n)\,?\,1:0 && (4.2)\\
g_{\ell+1}(i,m,j,n) &= \sum_{i' \in [1..N_s], m' \in S_M} g_\ell(i,m,i',m') A_{i',j} p_j(m',n) && (4.3)
\end{aligned}
$$

$$
\begin{aligned}
\alpha_1(j,n) &= && (4.4)\\
&\begin{cases}
(n = \delta(m_{init}, O_1))\,?\,\pi_j b_j(O_1) : 0 & \text{if } O_1 \in V\\
L(0)(n = m_{init}\,?\,\pi_j : 0) + \sum_{\ell>0, i\in[1..N_s]} L(\ell)\pi_i g_\ell(i, m_{init}, j, n) & \text{if } O_1 = gap(L)\\
(n = m_{init} \wedge n \in S_p)\,?\,\pi_j : 0 & \text{if } O_2 = peek(S_p)
\end{cases}\\
&\text{for } 1 \le j \le N_s \text{ and } n \in S_M
\end{aligned}
$$

$$
\alpha_{t+1}(j,n) = \begin{cases}
\left(\sum_{\substack{i\in[1..N_s]\\ m\in \mathrm{pred}(n, O_{t+1})}} \alpha_t(i,m) A_{i,j} \right) b_j(O_{t+1}) & \text{if } O_{t+1} \in V\\
L(0)\alpha_t(j,n) + \sum_{\ell>0} L(\ell) \sum_{\substack{i\in[1..N_s]\\ m\in S_M}} \alpha_t(i,m) g_\ell(i,m,j,n) & \text{if } O_{t+1} = gap(L)\\
(n \in S_p)\,?\,\alpha_t(j,n) : 0 & \text{if } O_{t+1} = peek(S_p)
\end{cases} \tag{4.5}
$$

$$\text{for } 1 \le t \le T-1 \text{ and } 1 \le j \le N_s \text{ and } n \in S_M$$

Figure 4.2: Forward algorithm for Runtime Verification with State Estimation
$\mathrm{pred}(n, v)$ is the set of predecessors of n with respect to v in the DFSM, i.e., the set of states m such that M transitions from m to n on input v.

RVSE Algorithm In [61], we extended the forward algorithm to estimate the probability of having encountered an error (equivalent to be in an accepting state) in the case where the observation sequence O contains the symbol $gap(L)$ denoting a possible gap with an unknown length. The length distribution L is a probability distribution on the natural numbers: $L(\ell)$ is the probability that the gap has length ℓ.

The Hidden Markov Model $H = \langle S, A, V, B, \pi \rangle$ models the monitored system, where $S = \{s_1, \ldots, s_{N_s}\}$ and $V = \{v_1, \ldots, v_{N_o}\}$. Observation symbols of H are observable actions of the monitored system. H need not be an exact model of the system.

The property ϕ is represented by a DFSM $M = \langle S_M, m_{init}, V, \delta, F \rangle$. For simplicity, we take the alphabet of M to be the same as the set of observation symbols of H. It is easy to allow the alphabet of M to be a subset of the observation symbols of H, by modifying the algorithm so that observations of symbols outside the alphabet of M leave M in the same state.

The goal is to compute $\Pr(\phi \mid O, H)$, i.e., the probability that the system's behavior satisfies ϕ, given observation sequence O and model H. Let $Q = \langle q_1, q_2, \ldots, q_T \rangle$ denote the (unknown) state sequence that the system passed through, i.e., q_t denotes the state of the system when observation O_t is made. We extend the forward algorithm [54] to compute $\alpha_t(i,m) = \Pr(O_1, O_2, \ldots, O_t, q_t = s_i, m_t = m \mid H)$, i.e., the joint probability that the first t observations yield $O_1, O_2, \ldots, O_t$ and that q_t is s_i and that m_t is m, given the model H. We refer to a pair (j,n) of an HMM state and a DFSM state as a *compound state*, and we sometimes refer to α_t as a probability distribution over compound states.

In this work, we further extend the algorithm to revise its probability estimate when O contains symbols of the form $peek(S_p)$, where $S_p \subset S_M$. A *peek* event represents knowledge from an oracle that the DFSM can only be in one of the states in the set S_p: i.e., $O_t = peek(S_p)$ implies that $\forall s_n \in S_M, 1 \leq j \leq N_s : s_n \in S_p \vee \alpha_t(j,n) = 0$.

Peek events are useful when some DFSM states imply program properties that can be independently checked at any time. In our case study (Section 4.6), one of the DFSM states implies that a lock is held. At any time, the runtime verification framework can examine the lock directly; if the lock is not held, the framework rules out that DFSM state by emitting a peek observation.

The extended algorithm appears in Figure 4.2. The desired probability $\Pr(\phi \mid O, H)$ is the probability that the DFSM is in an accepting state after observation sequence O, which is $p_{\text{sat}}(\alpha_{|O|+1})$, where $p_{\text{sat}}(\alpha) = \sum_{j \in 1..N_s, n \in F} \alpha(j,n) \;/\; \sum_{j \in 1..N_s, n \in S_M} \alpha(j,n)$. The probability of an error (i.e., a violation of the property) is $p_{\text{err}}(\alpha) = 1 - p_{\text{sat}}(\alpha)$.

4.3 The ARV Framework: Architecture and Principles

Figure 4.1 depicts the architecture of the ARV framework. ARV uses GCC plug-ins [13] to insert code that intercepts monitored events and sends them to the *monitoring framework*. The monitoring framework maintains a separate RVSE-enabled monitor instance for each monitored object.

Each monitor instance uses the RVSE algorithm in Section 4.4 to compute its estimate of the composite HMM-DFSM state; specifically, it keeps track of which pre-computed probability distribution over compound states characterizes the current system state.

Each instance uses this probability distribution over compound states to compute the its *error probability* (EP), i.e., the probability that a property violation has occurred, as described in Section 4.4. Each instance also uses this probability distribution over compound states to compute its *criticality level*, based on the expected number of transitions before a violation occurs, using the predictive analysis of Section 4.5.

The overhead-control subsystem is structured, as in SMCO [34], as a *cascade controller* comprising one primary controller and a number of secondary controllers, one per monitor instance. The primary controller allocates monitoring resources (overhead), and the secondary controllers enforce the overhead allocation by disabling monitoring when necessary. A key feature of ARV's design is the ability to redistribute overhead so that more critical monitor instances are allowed more monitoring overhead.

4.4 Pre-computation of RVSE Distributions

Performing the matrix calculations in the RVSE algorithm during monitoring incurs very high overhead. This section describes how to dramatically reduce the overhead by pre-computing

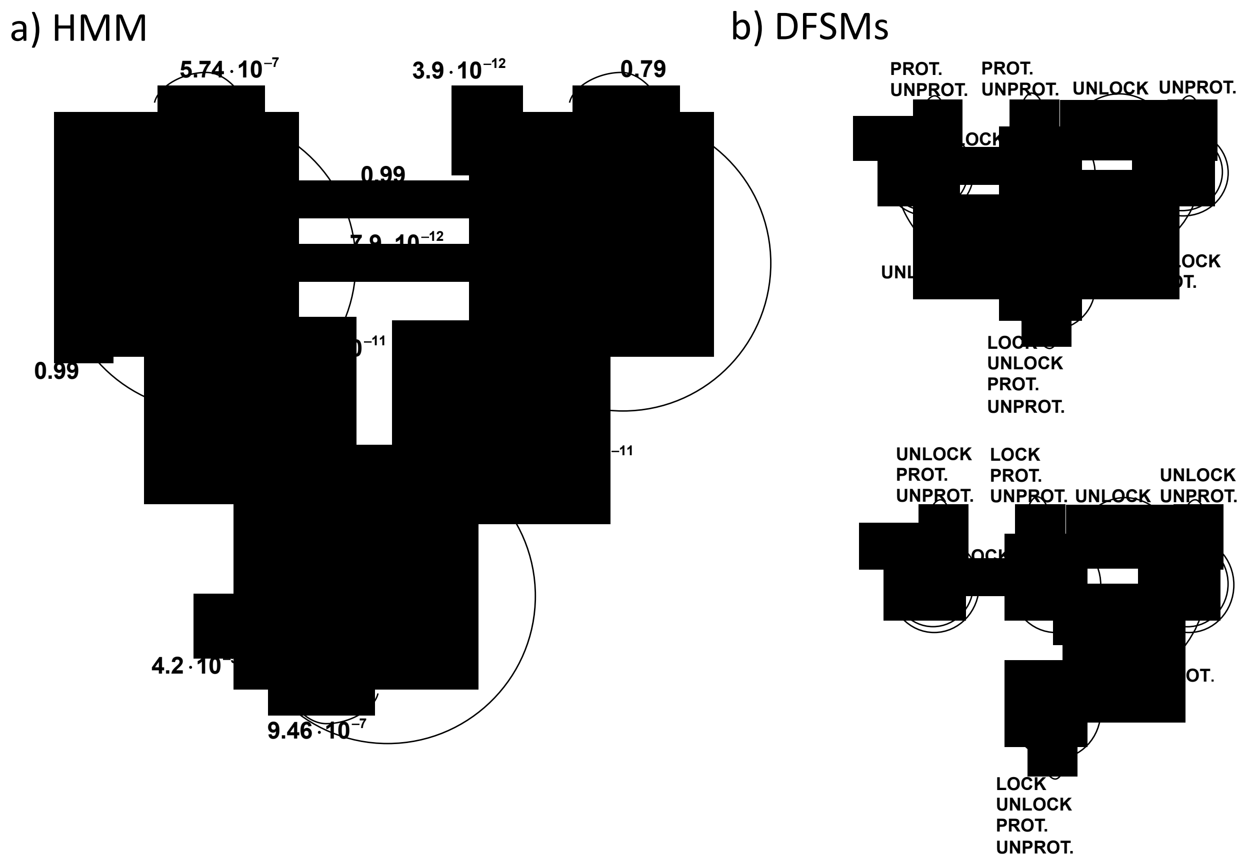


Figure 4.3: HMM and DFSM examples
Left (a): An example of an HMM. Right (b): Two examples of DFSM. States with a double border are accepting states.

compound-state probability distributions α and storing them in a rooted graph. Each edge of the graph is labeled with an observation symbol. At runtime, the algorithm maintains (for each monitor instance) a pointer $curNode$, indicating the node associated with the current state. The probability distribution in the current state is given by the matrix associated with $curNode$. Initially, $curNode$ points to the root node. Upon observing an observation symbol O, the algorithm finds the node n' reachable from $curNode$ by an edge labeled with O, and then assigns n' to $curNode$. Note that this takes constant time, independent of the sizes of the HMM and the monitor.

In general, an unbounded number of probability distributions may be reachable, in which case the graph would be infinite. We introduce an approximation in order to ensure termination. Specifically, we introduce a binary relation $\mathrm{closeEnough}$ on compound-state probability distributions, and during the graph construction, we identify nodes that are close enough.

Pseudo-code for the graph construction appears in Figure 4.4. $\mathrm{successor}(\alpha, O)$ is the probability distribution obtained using the forward algorithm—specifically, Equation 4.5—to update compound-state probability distribution α based on observation of observation symbol O. Note that each edge is marked as *exact* or *approximate*, indicating whether it introduces any inaccuracy. $\mathrm{normalize}(\alpha)$ is the probability distribution obtained by computing $\sum_{j,n} \alpha(j, n)$ and then

g = input gap probability distribution
V_{peek} = input set of possible peek events
α_0 = the probability distribution with $\alpha_0(j, m_{init}) = \pi_0(j)$, and $\alpha_0(j, n) = 0$ for $n \neq m_{init}$
$workset = \{\alpha_0\}$
$nodes = \{\alpha_0\}$
while $workset \neq \emptyset$
  $\alpha = workset.\text{removeOne}();$
  for each observation symbol O in $(V \cup V_{peek} \cup \{gap(g)\})$
    $\alpha' = \text{normalize}(\text{successor}(\alpha, O))$
    if $\text{dead}(\alpha')$
      continue
    endif
    if $\alpha' \in nodes$
      add an exact edge labeled with O from α to α'
    else if there exists α'' in $nodes$ such that $\text{closeEnough}(\alpha', \alpha'')$
      add an approximate edge labeled with O from α to α''
    else
      add α' to $nodes$ and $workset$
      add an exact edge labeled with O from α to α'
    endif
  endfor
endwhile

Figure 4.4: Pseudo-code for graph construction

dividing every entry in α by this sum; the resulting matrix α' satisfies $\sum_{j,n} \alpha'(j, n) = 1$. Normalization has two benefits. First, it helps reduce the number of nodes, because normalized matrices are more likely to be equal or close-enough than un-normalized matrices. Second, normalization helps reduce the inaccuracy caused by the use of limited-precision numerical calculations in the implementation (*cf.* [54, Section V.A]), which uses the term "scaling" instead of "normalization"). Normalization is compatible with our original RVSE algorithm—in particular, it does not affect the value calculated for $\Pr(\phi \mid O, H)$—and it provides the second benefit described above in that algorithm, too, so we assume hereafter that the original RVSE algorithm is extended to normalize each matrix α_t.

A state s of a DFSM is *dead* if it is non-accepting and all of its outgoing transitions lead to s. A probability distribution is *dead* if the probabilities of compound states containing dead states of the DFSM sum to 1. The algorithm does not bother to compute successors of dead probability distributions, which always have error probability 1.

During construction, the $nodes$ list is stored as a k-d tree, in which each node is a vector with dimension $(|S||S_M|)$. Searching $nodes$ for an element that satisfies $closeenough$ becomes the most time-consuming operation of the construction once $nodes$ contains more than a thousand elements. The k-d tree is optimized for this style of search, providing a best-case $O(log(n))$ lookup when the tree is perfectly balanced [9].

We define the close-enough relation by: $\text{closeEnough}(\alpha, \alpha')$ iff $||\alpha - \alpha'||_{\text{sum}} \leq \epsilon$, where ϵ is an implicit parameter of the construction, and $||\alpha||_{\text{sum}} = \sum_{i,j} |\alpha(i,j)|$. Note that, if we regard α as a vector, as is traditional in HMM theory, then this norm is the vector 1-norm.

Termination We prove termination of the graph construction using the pigeonhole principle. Consider the space of $N_s \times N_m$ matrices with entries in the range $[0..1]$, where $N_m = |S_m|$. Partition this space into cells (hypercubes) with edge length $\epsilon/N_s N_m$. If two matrices α and α' are in the same cell, then the absolute value of the largest element in $\alpha - \alpha'$ is at most $\epsilon/N_s N_m$, and $||\alpha - \alpha'||_{\text{sum}}$ is at most the number of elements times the largest element, so $||\alpha - \alpha'||_{\text{sum}} \leq N_s N_m \epsilon / N_s N_m$, hence $||\alpha - \alpha'||_{\text{sum}} \leq \epsilon$. The contrapositive of this conclusion is: if two matrices satisfy $||\alpha - \alpha'||_{\text{sum}} > \epsilon$, then they are in different cells. Therefore, the number of nodes in the graph is bounded by the number of cells in this grid, which is $(N_s N_m/\epsilon)^{N_s N_m}$. Note that this termination proof applies even if normalization is omitted from the algorithm.

Cumulative Inaccuracy Use of the closeEnough relation during graph construction introduces inaccuracy. We characterize the inaccuracy by bounding the difference between the probability distribution matrix associated with $curNode$ and the probability distribution matrix that would be computed by the original RVSE algorithm. Let $\alpha'_1, \alpha'_2, \ldots, \alpha'_t$ be the sequence of matrices labeling the nodes visited in the graph, for a given observation sequence O. Let $\alpha_1, \alpha_2, \ldots, \alpha_t$ be sequence of matrices calculated by the RVSE algorithm for the same observation sequence O. The cumulative inaccuracy is expressed as a bound err_t on $||\alpha_t - \alpha'_t||_{\text{sum}}$. First, we consider inaccuracy assuming that the original and new RVSE algorithms do *not* normalize the probability distributions (recall that normalization is not needed to ensure soundness or termination), and we show that the cumulative inaccuracy does not increase along an exact edge and increases by at most ϵ along an approximate edge.

We define err_t inductively. The base case is $err_0 = 0$. For the induction case, we suppose $||\alpha'_t - \alpha_t||_{\text{sum}} \leq err_t$ and define err_{t+1} so that $||\alpha'_{t+1} - \alpha_{t+1}||_{\text{sum}} \leq err_{t+1}$.

If the transition from α'_t to α'_{t+1} traverses an exact edge, then the inaccuracy remains unchanged: $err_{t+1} = err_t$. To prove this, we show that the following inequality holds:

$$||\text{successor}(\alpha'_t, O_{t+1}) - \text{successor}(\alpha_t, O_{t+1})||_{\text{sum}} \leq err_t.$$

To prove this, we expand the definition of successor and simplify. There are two cases, depending on whether O_{t+1} is a gap. If O_{t+1} is not a gap,

$$\begin{array}{ll}
& \sum_{j\in[1..N_s],n\in S_M} |\text{successor}(\alpha'_t, O_{t+1}) - \text{successor}(\alpha_t, O_{t+1})| \\
= & \sum_{j\in[1..N_s],n\in S_M} \sum_{i\in[1..N_s],m\in \text{pred}(n,O_{t+1})} |\alpha'_t(i,m) - \alpha_t(i,m)| A_{i,j} b_j(O_{t+1}) \\
& \text{// } M \text{ is deterministic, so each } m \text{ is predecessor of at most one } n \text{ for given } O_{t+1}, \\
& \text{// so for any } f, \sum_{n\in S_M, m\in\text{pred}(n,O_{t+1})} f(m) \leq \sum_{m\in S_M} f(m). \\
\leq & \sum_{j\in[1..N_s],i\in[1..N_s],m\in S_M} |\alpha'_t(i,m) - \alpha_t(i,m)| A_{i,j} b_j(O_{t+1}) \\
& A \text{ is stochastic, i.e., } \sum_{j\in S_M} A_{i,j} = 1, \text{ and } b_j(O_{t+1}) \leq 1 \\
\leq & \sum_{i\in[1..N_s],m\in S_M} |\alpha'_t(i,m) - \alpha_t(i,m)| \\
\leq & err_t
\end{array}$$

If O_{t+1} is a gap,

$$
\begin{array}{rl}
& \sum_{j\in[1..N_s],n\in S_M} |\mathrm{successor}(\alpha'_t, O_{t+1}) - \mathrm{successor}(\alpha_t, O_{t+1})| \\
= & \sum_{j\in[1..N_s],n\in S_M} L(0)|\alpha'_t(j,n) - \alpha_t(j,n)| \\
& + \sum_{\ell>0} L(\ell) \sum_{i\in[1..N_s],m\in\mathrm{pred}(n,O_{t+1})} |\alpha'_t(i,m) - \alpha_t(i,m)| g_\ell(i,m,j,n) \\
& \text{// definition of } err_t \\
\leq & L(0)err_t \\
& + \sum_{\ell>0} L(\ell) \sum_{i\in[1..N_s],m\in\mathrm{pred}(n,O_{t+1})} |\alpha'_t(i,m) - \alpha_t(i,m)| \sum_{j\in[1..N_s],n\in S_M} g_\ell(i,m,j,n) \\
& \text{// } g_\ell(i,m,\cdot,\cdot) \text{ is stochastic, i.e., } \sum_{j\in[1..N_s],n\in S_M} g_\ell(i,m,j,n) = 1 \text{, and def. of } err_t \\
\leq & L(0)err_t + \sum_{\ell>0} L(\ell) err_t \\
& \text{// } \sum_{\ell\geq 0} L(\ell) = 1 \\
\leq & err_t
\end{array}
$$

If the transition from α'_t to α'_{t+1} traverses an approximate edge, then, by definition of the $\mathrm{closeEnough}$ relation, the traversal may add ϵ to the cumulative inaccuracy, so $err_{t+1} = err_t + \epsilon$. Note that the same argument used in the case of an exact edge implies that the inaccuracy in err_t is not amplified by traversal of an approximate edge.

Now we consider the effect of normalization on cumulative inaccuracy. We show that normalization does not increase the inaccuracy. Let $\hat{\alpha}'_t$ and $\hat{\alpha}_t$ be the probability distributions computed in step t before normalization; thus, $\alpha'_t = \mathrm{normalize}(\hat{\alpha}'_t)$ and $\alpha_t = \mathrm{normalize}(\hat{\alpha}_t)$. Note that $\sum_{j,n} \hat{\alpha}'_t(j,n)$ and $\sum_{j,n} \hat{\alpha}_t(j,n)$ are at most 1; this is a property of the forward algorithm (*cf.* [54, Section V.A]). Also, every element of $\hat{\alpha}'_t$, $\hat{\alpha}_t$ α'_t, and α_t is between 0 and 1. Thus, normalization moves each element of $\hat{\alpha}'_t$ and $\hat{\alpha}_t$ to the right on the number line, closer to 1, or leaves it unchanged. For concreteness, suppose $\sum_{j,n} \hat{\alpha}'_t(j,n) < \sum_{j,n} \hat{\alpha}_t(j,n)$; a completely symmetric argument applies when the inequality points the other way. This inequality implies that, on average, elements of $\hat{\alpha}'_t$ are to the left of elements of $\hat{\alpha}_t$. It also implies that, on average, normalization moves elements of $\hat{\alpha}'_t$ farther (to the right) than it moves elements of $\hat{\alpha}_t$. These observations together imply that, on average, corresponding elements of $\hat{\alpha}'_t$ and $\hat{\alpha}_t$ are closer to each other after normalization than before normalization, and hence that $||\alpha'_t - \alpha_t||_{\mathrm{sum}} \leq ||\hat{\alpha}'_t - \hat{\alpha}_t||_{\mathrm{sum}}$. Note that elements of $\hat{\alpha}'_t$ cannot move so much farther to the right than elements of $\hat{\alpha}_t$ that they end up being farther, on average, from the corresponding elements of $\hat{\alpha}_t$, because both matrices end up with the same average value for the elements (namely, $1/N_sN_m$).

Stricter Close-Enough Relation To improve the accuracy of the algorithm, a slightly stricter close-enough relation is used in our experiments: $\mathrm{closeEnough}(\alpha, \alpha')$ holds iff $||\alpha - \alpha'||_{\mathrm{sum}} \leq \epsilon \wedge (p_{\mathrm{dead}}(\alpha) = 0 \Leftrightarrow p_{\mathrm{dead}}(\alpha') = 0)$, where $p_{\mathrm{dead}}(\alpha)$ is the sum of the elements of α corresponding to compound states containing a dead state of M. It is easy to show that the algorithm still terminates, and that the above bound on cumulative inaccuracy still holds.

On-Demand Computation We have observed that monitored executions usually visit only a small fraction of the pre-computed nodes in the graph. Because the graph goes mostly unused, it is practical to compute the graph on demand, as the monitored program executes.

The on-demand algorithm operates similarly to Figure 4.4, except there is no workset. Instead, individual edges get chosen for computation when they are needed: i.e., when a monitor instance in state α needs to process observation O but there is not yet an outgoing edge from α with label

O. This approach is similar to memoization of the RVSE computation, except that edges can be placed approximately.

Computing the entire graph can take hours to days, depending on what ϵ the user chooses (i.e., how much approximation is allowed), so on-demand computation can save a lot of time. The downside to on-demand computation is that event processing takes a variable amount of time: some events require only an edge traversal, while others incur an expensive RVSE computation. It makes sense to use on-demand computation alongside overhead control techniques that measure and account for event processing time, such as SMCO [34].

4.5 Predictive Analysis of Criticality Levels

Criticality Level We define the *criticality level* of a monitor instance to be the inverse of the expected distance (number of steps) to a violation of the property of interest. To compute this expected distance for each compound state, we compute a Discrete Time Markov Chain (DTMC) by composing the HMM model H of the monitored program with the DFSM M for the property. We then add a reward state structure to it, assigning a cost of 1 to each compound state. We use PRISM [38] to compute, as a reward-based reachability query, the expected number of steps for each compound state to reach compound states containing dead states of M. Note that these queries are issued in advance of the actual runtime monitoring, with the results stored in a table for efficient access.

Discrete-Time Markov Chain (DTMC) A Discrete-Time Markov Chain (DTMC) [38] is a tuple $\mathcal{D} = (S_D, \tilde{s}_0, \mathbf{P})$, where S_D is a finite set of states, $\tilde{s}_0 \in S_D$ is the initial state, and $\mathbf{P} : S_D \times S_D \to [0, 1]$ is the transition probability function. $\mathbf{P}(\tilde{s_1}, \tilde{s_2})$ is the probability of making a transition from $\tilde{s_1}$ to $\tilde{s_2}$.

Reward Structures DTMCs can be extended with a reward (or cost) structure [38]. A *state reward function* $\underline{\rho}$ is a function from states of the DTMC to non-negative real numbers, specifying the reward (or cost, depending on the interpretation of the value in the application of interest) for each state; specifically, $\underline{\rho}(\tilde{s})$ is the reward acquired if the DTMC is in state $\tilde{s}$ for 1 time-step.

Composition of an HMM with a DFSM Given an HMM $H = \langle S, A, V, B, \pi \rangle$ and a DFSM $M = \langle S_M, m_{init}, V, \delta, F \rangle$, their composition is a DTMC $\mathcal{D} = (S_D, \tilde{s}_0, \mathbf{P})$, where $S_D = (S \times S_M) \cup \{\tilde{s}_0\}$, $\tilde{s}_0$ is the initial state, and the transition probability function $\mathbf{P}$ is defined by:

- $\mathbf{P}(\tilde{s}_0, (s_i, m_{init})) = \pi$, with $1 \leq i \leq |S|$,
- $\mathbf{P}((s_{i_1}, m_{j_1}), (s_{i_2}, m_{j_2})) = A_{i_1, i_2} \sum_{\forall v_k \in V : \delta(m_{i_1}, v_k) = m_{i_2}} b_{i_1}(v_k)$.

We extend $\mathcal{D}$ with the state reward function such that $\underline{\rho}(\tilde{s}) = 1$ for all $\tilde{s} \in S_D$. With this reward function, we can calculate the expected number of steps until a particular state of the DTMC occurs.

Computing the Expected Distance The expected distance $ExpDist(\tilde{s}, T)$ of a state $\tilde{s}$ of the DTMC to reach a set of states $T \subseteq S_D$ is defined as the expected cumulative reward and is computed as follows:

$$\mathrm{ExpDist}(\tilde{s}, T) = \begin{cases} \infty & \text{if } \mathrm{PReach}(\tilde{s}, T) < 1 \\ 0 & \text{if } \tilde{s} \in T \\ \underline{\rho}(\tilde{s}) + \sum_{\tilde{s}' \in S_D} \mathbf{P}(\tilde{s}, \tilde{s}') \cdot \mathrm{ExpDist}(\tilde{s}', T) & \text{otherwise} \end{cases}$$

where $\mathrm{PReach}(\tilde{s}, T)$ is the probability to eventually reach a state in T starting from $\tilde{s}$. For further details on quantitative reachability analysis for DTMCs, see [38]. The expected distance for a monitor instance with compound-state probability distribution α is then defined by $\mathrm{ExpDist}(\alpha, T) = \sum_{i,j} \alpha(i, j) \cdot \mathrm{ExpDist}((s_i, m_j), T)$.

4.6 Case Study

We evaluate our system by designing a monitor for the lock discipline property and applying it to the Btrfs file system. This property is implicitly parameterized by a `struct` type S that has a lock member, protected fields, and unprotected fields. Informally, the property requires that all accesses to protected fields occur while the lock is held.

The DFSM $M^{LD}(t, o)$ for the lock discipline property is parameterized by a thread t and an object o, where o is an instance of the `struct` with type S. There are four kinds of events: $\mathrm{LOCK}(t, o)$ (thread t acquires the lock associated with object o), $\mathrm{UNLOCK}(t, o)$ (thread t releases the lock associated with object o), $\mathrm{PROT}(t, o)$ (thread t accesses a protected field of object o), and $\mathrm{UNPROT}(t, o)$ (thread t accesses an unprotected field of object o). The DFSM $M^{LD}(t, o)$ is shown in the lower part of Figure 4.3(b); the parameters t and o are elided to avoid clutter. It requires that thread t's accesses to protected fields occur while thread t holds the lock associated with object o, except for accesses to protected fields before the first time t acquires that lock (such accesses are assumed to be part of initialization of o).

When it is not in its error state (state 4), the state of DFSM $M^{LD}(t, o)$ represents whether thread t holds the lock on object o. Note that a $\mathrm{LOCK}(t, o)$ event from any non-error state will transition the DFSM to state 2, and an $\mathrm{UNLOCK}(t, o)$ will never transition the DFSM to state 2. As a result, when t holds the lock on o, the actual DFSM state must be state 2 or the error state, and when it does not hold the lock, the DFSM must be in some state other than state 2.

This correspondence between the state of the lock and the state of the DFSM makes peek events (discussed in Section 4.2) useful in this case study. Our runtime framework has the ability to directly inspect an object o to see if it is held by a thread t and then generate a $\mathrm{LOCK\_HELD}(t, o)$ or $\mathrm{LOCK\_NOT\_HELD}(t, o)$ peek observation as appropriate. These events cause our state estimation algorithm (shown in Figure 4.2) to zero out the probabilities of DFSM states that are ruled out by the inspection: states 1 and 3 for $\mathrm{LOCK\_HELD}(t, o)$ and state 2 for $\mathrm{LOCK\_NOT\_HELD}(t, o)$.

4.7 Implementation

Implementing the case study requires a gap-aware monitor and instrumentation that can intercept monitored events. Both these subsystems must integrate with our overhead control mechanism.

The monitor must be able to recognize potential gaps caused by overhead control decisions, and the instrumentation must provide a means for the controller to disable monitoring by halting the interception of events. In addition, our implementation adapts to RVSE's criticality estimates by allocating hardware debugging resources to exhaustively monitor a small number of risky objects. This section discusses the implementation of these systems.

4.7.1 Gaps

On updating a monitor instance, the monitor processes a gap event before processing the current intercepted event if monitoring was disabled since the last time the monitor instance was updated. The gap event indicates that the monitor may have missed one or more events for the given instance during the time that monitoring was disabled.

The monitor determines whether a gap event is necessary by comparing the time of the last update to the monitor instance's state, which is stored along with the state, with the last time that monitoring was disabled for the current thread. For efficiency, we measure time using a counter incremented each time monitoring is disabled—a logical clock—rather than a real-time clock.

4.7.2 Peeking

When peeking is enabled, the monitor potentially performs a peek operation on a monitor instance immediately after processing a gap event for that instance. This operation examines a variable within the object's lock that indicates which thread currently holds it. The $\mathrm{LOCK\_HELD}$ or $\mathrm{LOCK\_NOT\_HELD}$ event is sent to the instance depending on whether the thread that holds the lock is the thread associated with the instance.

Peeking is designed to work in scenarios where directly inspecting a monitored object may be expensive. For example, determining whether an iterator points to an element in a list would require an $O(n)$ search through the list. The monitor can use an overhead control mechanism to determine whether or not to perform the peek. Our implementation has the ability to augment a user-specified fraction of gap events with peek operations, but it would also be possible to support an approach that applies peek operations to the most critical monitor instances.

4.7.3 Instrumentation

For our case study, we monitor the lock discipline property for the `btrfs_space_info` struct in the Linux Btrfs file system. Each `btrfs_space_info` object has a spinlock, eight fields protected by the spinlock, and five fields not protected by the spinlock.

Using a custom GCC plug-in, we instrument every kernel function that may operate on a `btrfs_space_info` object, either by accessing one of its fields or by acquiring or releasing its spinlock. The instrumented function first has its function body *duplicated* so that there is an *active* path and an *inactive* path. Only the active path is instrumented for full monitoring. This allows monitoring to be efficiently enabled or disabled at the granularity of a function execution. Selecting the inactive path effectively disables monitoring. When a duplicated function executes, it first calls a *distributor* function that calls the overhead control system to decide which path to take. We enable and disable monitoring at the granularity of function executions, because deciding to enable or disable monitoring at the granularity of individual events would incur too much overhead.

Every `btrfs_space_info` operation in the active path is instrumented to call the monitor, which updates the appropriate monitor instance, based on the thread and the `btrfs_space_info` object involved. For fast lookup, all monitor instances associated with a thread are stored in a hash table local to that thread and indexed by object address.

4.7.4 Hardware Supervision

Our system prioritizes monitoring of objects with high criticality by placing them under hardware supervision. Specifically, we use debug registers to monitor every operation on these objects even when other monitoring is disabled (i.e., when the inactive path is taken). The debug registers cause the CPU to raise a debug exception whenever an object under hardware supervision is accessed, allowing the monitor to observe the access. Note that this allows monitoring to be enabled and disabled on a per-object basis, for a limited number of objects, in contrast to the per-function-execution basis described above. The overhead remaining after monitoring the hardware supervised objects is distributed to the other objects in the system using the normal overhead control policy.

Our current implementation keeps track of the most critical object in each thread. Each thread can have its own debug register values, making it possible to exhaustively track events for one monitor instance in each thread for any number of threads.

Because an x86 debug register can at most watch one 64-bit memory location, we need a small amount of additional instrumentation to monitor all 13 fields in a supervised `btrfs_space_info` object. Our plug-in instruments every `btrfs_space_info` field access in the *inactive* path with an additional read to a dummy field in the same object. Setting the debug register to watch the dummy field of a supervised object causes the program to raise a debug exception whenever any field of that object is accessed from the inactive path. The debug exception handler calls the monitor to update the monitor instance for the supervised object.

For `btrfs_space_info` spinlock acquire and release operations, we instrument the inactive path with a simple check to determine if the spinlock belongs to one of the few supervised objects that should be updated even though monitoring is disabled. We could use debug registers to remove the need for this check, but we found that overhead from checking directly was very low, because lock operations occur infrequently compared to field accesses.

4.7.5 Training

We collected data from completely monitored runs to train the HMM and learn the gap length distribution. During training runs for a given overhead level, the distributor makes monitoring decisions as if overhead control were in effect but does not enforce those decisions; instead, it always takes the active path. As a result, the system knows which events would have been missed by taking the inactive path. Based on this information, for each event that would have triggered processing of a gap event, we compute the actual number of events missed for the corresponding monitor instance. The gap length distribution for the given overhead level is the distribution of those numbers.

Our case study uses a simple overhead-control mechanism in which the target "overhead level" is specified by the fraction f of function executions to be monitored. For each function execution, the distributor flips a biased coin, which says "yes" with probability f, to decide whether to monitor

Sampling	No Supervision		Random Supervision		Adaptive Supervision	
Probability	FalseAlarm	ErrDet	FalseAlarm	ErrDet	FalseAlarm	ErrDet
50%	30.3	23.0%	11.7	57.4%	12	50.1%
75%	47	31.2%	36	69.3%	17	79.4%
85%	5502	34.1%	5606	72.3%	5449	85.1%

Table 4.1: Benchmark results for ARV on BTRFS

the current function execution. We tested three different sampling probabilities: 50%, 75%, 85%, and 95%. For each sampling probability, we precomputed the RVSE distributions with $\epsilon = 0.1$, thereby obtaining four RVSE graphs having 12,177, 33,234, 30,645 and 11,622 nodes, respectively.

4.7.6 Evaluation

We used two different tests to measure how well our prioritization mechanism improved ARV's effectiveness. The first test runs with an unmodified version of Btrfs, which does not contain any lock discipline violations, in order to test how well prioritization avoids false alarms. The second test runs on a version of Btrfs with an erroneous access that we inserted, to test if prioritization improves our chances of detecting it. For both of these tests, we run Racer [62], a workload designed specifically to stress file system concurrency, on top of a Btrfs-formatted file system, and we report results that are averaged over multiple runs.

We tested three configurations: 1) hardware supervision disabled, 2) randomly assigned hardware supervision, and 3) adaptive hardware supervision that prioritizes critical objects, as described above. Most threads in the Racer workload had two associated monitor instances. At any time, our prioritization chose one of those from each thread to supervise.

The table below shows the results for these tests. Each row in the table is for one of the three sampling probabilities. For our false alarm test, the columns labeled FalseAlarm in the table show how many monitor instances had an error probability higher than 0.8 at the end of the run. Because the run had no errors, lower numbers are better in this test. For our error detection test, we checked the corresponding monitor instance immediately after our synthetic error triggered; the columns labeled ErrDet in the table show the percentage of the times that we found that monitor instance to have an error probability higher than 0.8, indicating it correctly inferred a likely error. For this test, higher numbers are better. All results are averaged over multiple runs.

In all cases, hardware supervision improved the false alarm rate and the error detection rate. For the 75% and 85% sampling profiles, adaptive prioritization provides greater improvement than simply choosing objects at random for supervision. With 50% sampling, adaptive sampling does worse than random, however. In future work, we intend to improve our criticality metric so that it performs better at lower overheads. The table also shows that ARV takes advantage of increased sampling rates, successfully detecting more errors in the error detection test. However, we see that false alarm rates increase with higher sampling rate. Our experiments on a micro-benchmark provide insight into this problem.

Micro-benchmark

We constructed a micro-benchmark, `arv-bench`, to measure ARV's performance when there are many objects per thread. An `arv-bench` run spawns five threads, which access 100 shared objects, allowing for a total of 500 monitor instances (one for each possible pair of thread and object). Each thread runs in a loop that continously calls a work function.

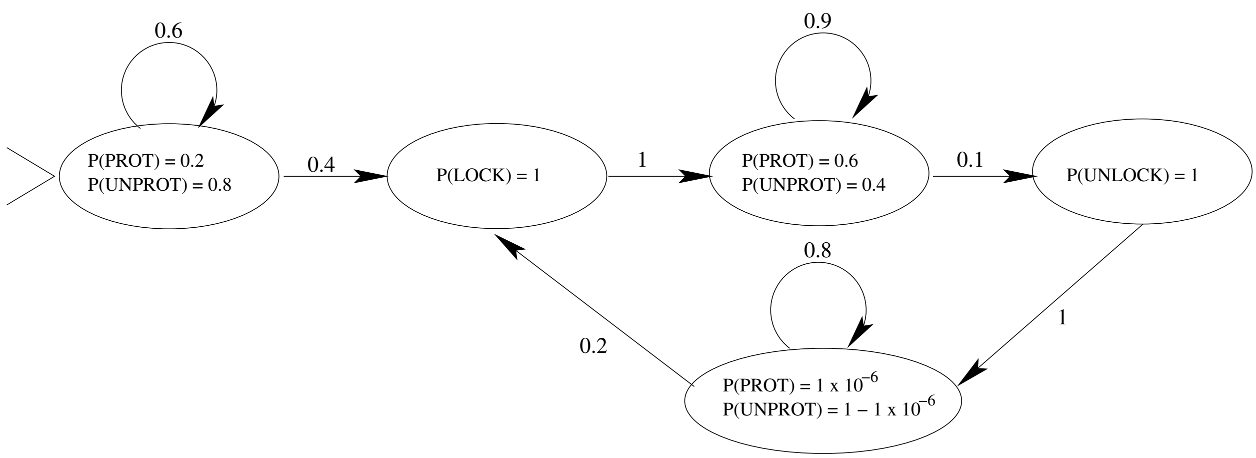


Figure 4.5: HMM used to generate work function operations in `arv-bench`.

The work function is itself a loop, each iteration of which randomly chooses one of the 100 objects to perform a randomly chosen operation on. The operation itself, which is one of LOCK, UNLOCK, PROT, or UNPROT (as described in Section 4.6), is chosen according to a manually constructed HMM. We use this same manually constructed HMM in the RVSE computation; there is no need to learn an HMM for the micro-benchmark, because we know a priori that objects follow the HMM we specified. The HMM that `arv-bench` uses is shown in Figure 4.5.

Depending on the decision made by the distributor, either all of the operations performed by an execution of the work function will be monitored or none of them will. The total number of operations that any work function execution will perform is chosen randomly according to the normal distribution with $\mu = 2$ and $\sigma = 0.5$ and rounded to the nearest non-negative integer. Grouping operations into random-length function executions makes `arv-bench`'s gap distribution more realistic.

Additionally, we need to restrict the operation of the benchmark so that the threads never enter a deadlock. We enforce a lock order by restricting which objects threads are allowed to operate on. The benchmark objects are indexed, and a thread will never choose to operate on an object with index n if it holds the lock on an object with an index greater than n, thereby ensuring that locks are always acquired in the same order.

So that threads do not hold locks for a long time, we give precedence to the highest-indexed locked object when randomly choosing an object to operate on, so as to unlock it sooner. The object choice chooses this preferred object with probability $1/3$; otherwise it chooses from the remaining available objects with uniform probability.

Sampling Probability	No Supervision FalseAlarm	ErrDet	Random Supervision FalseAlarm	ErrDet	Adaptive Supervision FalseAlarm	ErrDet
50%	68.2	92.2%	68.1	91.9%	68.1	92.3%
75%	124	96.2%	121	96.0%	123	96.0%
85%	127	96.1%	128	97.1%	128	96.9%
95%	107	95.9%	104	95.5%	106	95.5%

Table 4.2: Benchmark results for `arv-bench` with $\epsilon = 0.03$

Sampling Probability	No Supervision FalseAlarm	ErrDet	Random Supervision FalseAlarm	ErrDet	Adaptive Supervision FalseAlarm	ErrDet
50%	1.60	20.8%	1.20	32.5%	0.900	29.4%
75%	8.40	50.7%	7.10	47.6%	6.60	49.4%
85%	12.9	62.9%	10.8	60.2%	9.90	58.3%
95%	16.4	76.9%	15.2	72.4%	12.7	72.7%

Table 4.3: Benchmark results for `arv-bench` with $\epsilon = 0.0005$

As with BTRFS, our initial results for `micro-bench` (Table 4.2) also show that increasing the percentage of monitored function executions causes an unexpected increase in the number of false alarms.

We found that the false alarms for higher distributor probabilities are a result of error caused by approximate edges in our offline RVSE computation. When we repeated the `micro-bench` test with RVSE computed on the fly, there were *no false alarms* for any of the distributor probability values we tested.

Because running RVSE on the fly is prohibitively expensive, we sought to instead reduce approximation error by using a lower value for ϵ. Table 4.3 shows that with $\epsilon = 0.0005$, false alarms occur at a lower rate than with $\epsilon = 0.03$. The lower ϵ would have created an impractically large graph, so we used the on-demand RVSE computation (described in Section 4.4) for this test, allowing us to compute only the portion of the graph that was necessary for each execution.

The results in Table 4.3 more closely match our expectations: increasing the amount of monitoring increases error detection monotonically, and even though false alarms increase as well, they increase only modestly. The higher ϵ (lower accuracy) experiment in Table 4.2 has higher error detection results across the board but only because the monitor flags so many instances as erroneous that it is likely to flag most errors. We do not see consistently better performance when enabling hardware supervision in these tests. Because there are many more instances per thread in `arv-bench`, it is more difficult for supervision to choose which objects are likely to cause a fault.

The micro-benchmark also shows that on-demand RVSE dramatically reduces the number of nodes we need to compute in the RVSE graph. For example, with $\epsilon = 0.03$, the largest graph we computed (for the 95% sampling probability) has 164,671 nodes, but a monitored run of `arv-bench` visits 830 of these on average: only 0.5% of the total nodes. In the smallest graph (for the 50% sampling probability), an average `arv-bench` execution visits 866 of 47,494 nodes, or 1.8%. This picture is even more dramatic when $\epsilon = 0.0005$. With 95% sampling probability, an averaged monitored run computed 6,334 on-demand nodes. An offline computation of the

RVSE graph computed more than 13 million nodes, over the course of three days, before we finally terminated it.

Though we expected peeking to improve results in `arv-bench`, we actually found that enabling peeking increased the false alarm rate, even for on-the-fly RVSE. These false alarms occur when a peek result eliminates all state possibilities except the error state. That is, a monitor instance sees a peek event O_t, and as a result $\forall j, n \in F : \alpha_t(j, n) = 0$ (according to the RVSE formula described in Section 4.2). Informally, consider a monitor instance that believes either its lock is held or it is in the error state. When it sees a peek event indicating its lock is not held, the error probability becomes 1. We are still investigating how to prevent these false positives so that we can use peeking to improve our error detection accuracy.

4.8 Related Work

In [19], the authors propose a method for the automatic synthesis and adaptation of invariants from the observed behavior of an application. Their overall goal is adaptive application monitoring, with a focus on interacting software components. In contrast to our approach, where we learn HMMs, the invariants learned are captured as finite automata (FA). These FA are necessarily much larger than their corresponding HMMs. Moreover, error uncertainty, due to inherently limited training during learning, must be dealt with at runtime, by modifying the FA as needed. They also do not address the problem of using the synthesized FA for adaptive-control purposes.

A main aspect of our work is our approximation of the RVSE forward algorithm for state estimation, which pre-computes compound-state probability distributions and stores them in a graph. In the context of the runtime monitoring of HMMs, the authors of [59] propose a complementary method for accelerating the estimation of the current (hidden) state: Particle filters [32]. This sequential Monte-Carlo estimation method is particularly useful when the number of states of the HMM is very large, in particular, much larger than the number of particles (i.e., samples) necessary for obtaining a sufficiently accurate approximation. This, however, is typically not the case in our setting, where the HMMs are relatively small. Consequently, the Particle filtering method would have introduced at least as much overhead as the forward algorithm, and would have therefore also required a priori (and therefore approximate) state estimation.

The runtime verification of HMMs is explored in [31, 58], where highly accurate deterministic and randomized methods are presented. In contrast, we are considering the runtime verification of actual programs, while using probabilistic models of program behavior in the form of HMMs to fill in gaps in execution sequences.

Chapter 5

Conclusion

Any effort to make useful systems concurrency verification tools faces many challenges. Software systems consist of millions of lines of code, and that code is not designed with verification in mind. Developer assumptions about which regions should be atomic or when a data structure needs to be protected by a lock are not formally specified, and the assumptions can be subtle, as in the case of multi-stage escape (Section 2.1.4). And for any debugging tool, any extra slowdown makes the tool less valuable in the eyes of developers. At the kernel level, these tools must be designed to work even in interrupt context, where delays can slow down the entire system.

This work addresses several of these challenges. We use targeted logging and monitoring to cope with the size of systems codebases. Our analysis tools are designed with complex systems code in mind, using LOA analysis to infer assumptions about object life cycles and taking into account false positives caused by bitfield accesses or idempotent operations. For the monitoring and logging itself, we focus on performance while still ensuring that we can capture all the information necessary to diagnose problems. Logging captures full stack traces for all events and never drops events, even when they occur within interrupt handlers.

Our INTERASPECT framework addresses the instrumentation challenges inherent in runtime monitoring. GCC plug-ins are an effective platform for targeted instrumentation because of their access to compiler type information, and INTERASPECT streamlines plug-in development by hiding GCC's internal complexity. Using INTERASPECT to design GCC plug-ins, developers can quickly implement instrumentation for new runtime monitors.

Finally, Adaptive Runtime Verification (ARV) demonstrates how to make the most out of available resources when it is not practical to monitor every event in a system. Techniques like ARV may be the key to moving verification techniques from debugging toolboxes to production systems, where computing resources are at a premium.

It is our hope that the contributions presented here will benefit both the research and development communities. The INTERASPECT source is already available for download, along with complete documentation of its API [36].

5.1 Future Work

Locking performance

Though the runtime verification techniques we have described so far verify correctness, we could apply similar techniques to discover performance bottlenecks. As with correctness, performance problems at the system level are magnified by the fact that they can become problems for all of the applications running on the system.

Unnecessarily long critical sections make lock contention more likely, potentially squeezing parallelism out of the system. Using a profiler to find contended critical sections, we could design analyses to see where they can be broken up without introducing new data races or atomicity violations.

On the other hand, overly fine-grained locking also has a performance cost. In the absence of contention, breaking up critical sections introduces overhead from lock acquire and release functions without actually allowing more parallelism. Merging critical sections will not introduce races or atomicity violations, so we would only need to check for potential deadlock. At the kernel level, threads are not permitted to sleep while holding spinlocks, so we would also need analysis to ensure that merging a pair of spinlock-protected critical sections does not pull blocking operations into the spinlock.

Hardware support

Clever applications of existing hardware can sometimes improve performance of online runtime analysis. The NAP detector, for example, uses memory protection hardware to selectively monitor some regions for utilization without any performance penalty for accesses to other regions [34]. To check for data races, DataCollider uses debug registers to efficiently check if a specific memory operation occurs concurrently with another access to the same address [25].

We could augment the online atomicity checker in Section 2.2 to use debug registers instead of shadow memory to find violating accesses. Debug registers would only able to monitor a small number of variables for violations at any one time, but they could monitor these variables very efficiently. If we had a metric to determine which variables are more likely to be involved in a violation, we could prioritize those, as we do with hardware supervision in our ARV implementation (Section 4.7.4).

Schedule perturbation

For concurrency verification techniques that require a violating schedule in order to report an error, such as the online atomicity checker we present in Section 2.2, it is helpful to have a scheduler that is designed to trigger violating schedules. We would like to design a scheduler that tracks which variables are in the working set of each active atomic region so that it can schedule atomic regions together when they are accessing the same variables. Our hope is that encouraging atomic regions to access data structures simultaneously will make errors more likely.

Initial experiments with a modified scheduler did not show a measurable improvement in the probability of triggering a violating schedule. Further investigation is necessary to understand how we can better determine which atomic regions should be scheduled together to expose more errors. Additionally, scheduler perturbation could be useful for the LOA analysis we presented in Section 2.1.4, which gives more accurate results when it observes a wider variety of interleavings.

Further acknowledgements Part of the research described herein was carried out at the Jet Propulsion Laboratory (JPL), California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Research described here was supported in part by NSF Grants CCF-0926190, CCF-0613913, CNS-0831298, CNS-0509230, CCF-1018459, and CCF-0926190, ONR Grants N00014-07-1-0928 and N00014-09-1-0651. and AFOSR Grant FA9550-09-1-0481.

Bibliography

[1] B. Adams, C. Herzeel, and K. Gybels. cHALO, stateful aspects in C. In *ACP4IS '08: Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*, pages 1–6, New York, NY, USA, 2008. ACM.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*. ACM Press, 2005.

[3] BCEL. `http://jakarta.apache.org/bcel`.

[4] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 7–18, New York, NY, USA, 2010. ACM.

[5] AT&T Research Labs. Graphviz, 2009. `www.graphviz.org`.

[6] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *Proceedings of the Fourth International Conference on Aspect-Oriented Software Development*. ACM Press, 2005.

[7] J. Bacik. Possible race in btrfs, 2010. `http://article.gmane.org/gmane.comp.file-systems.btrfs/5243/`.

[8] L. E. Baum, T. Petrie, G. Soules, and N. Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970.

[9] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[10] E. Bodden and K. Havelund. Racer: Effective race detection using AspectJ. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 155–165. ACM, 2008.

[11] B. B. Brandenburg and J. H. Anderson. Feather-Trace: A light-weight event tracing toolkit. In *In Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'07)*, pages 61–70, 2007.

[12] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 107–120, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] S. Callanan, D. J. Dean, and E. Zadok. Extending GCC with modular GIMPLE optimizations. In *Proceedings of the 2007 GCC Developers' Summit*, pages 31–37, Ottawa, Canada, July 2007.

[14] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, 2007.

[15] S. Chiba. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 285–299, October 1995.

[16] S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming, LNCS*, volume 1850, pages 313–336. Springer Verlag, 2000.

[17] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 88–98, 2001.

[18] J. Corbet. write(), thread safety, and POSIX, 2006. `http://lwn.net/Articles/180387/`.

[19] G. Denaro, L. Mariani, M. Pezze, and D. Tosi. Adaptive runtime verification for autonomic communication infrastructures. In *Proc. of the International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, volume 2, pages 553–557. IEEE Computer Society, 2005.

[20] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. *SIGPLAN Not.*, 26(12):85–96, 1991.

[21] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with Arachne. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*. ACM Press, 2005.

[22] The Eclipse Foundation. AspectJ. `www.eclipse.org/aspectj`.

[23] Arachne. `www.emn.fr/x-info/arachne`.

[24] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252. ACM Press, 2003.

[25] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–12. USENIX Association, 2010.

[26] L. Fei and S. P. Midkiff. Artemis: Practical runtime monitoring of applications for errors. Technical Report TR-ECE-05-02, Electrical and Computer Engineering, Purdue University, 2005. `docs.lib.purdue.edu/ecetr/4/`.

[27] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*, pages 256–267, New York, NY, USA, 2004. ACM.

[28] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 338–349. ACM Press, 2003.

[29] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 293–303. ACM Press, 2008.

[30] GCC 4.5 release series changes, new features, and fixes. `http://gcc.gnu.org/gcc-4.5/changes.html`.

[31] K. Gondi, Y. Patel, and A. P. Sistla. Monitoring the full range of omega-regular properties of stochastic systems. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'09)*, pages 105–119, Savannah, GA, USA, 2009.

[32] N. J. Gordon, D. J. Salmond, and A. F. M. Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. In *IEEE Proceedings on Radar and Signal Processing*, volume 140, pages 107–127. IEEE, 1993.

[33] Tim Harris, Adrian Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, may-june 2007.

[34] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok. Software monitoring with controllable overhead. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(3):327–347, 2012.

[35] Objective Caml. `http://caml.inria.fr/index.en.html`.

[36] InterAspect. `www.fsl.cs.stonybrook.edu/interaspect`.

[37] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–355. LNCS, Vol. 2072, 2001.

[38] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In *Formal Methods for Performance Evaluation*, volume 4486 of *Lecture Notes in Computer Science*, pages 220–270. Springer, 2007.

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

[40] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, 46:779–782, July 1997.

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

[42] S. Lu, J. Tucek, F. Qin, and Y. 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 (ASPLOS-XII)*, pages 37–48. ACM, 2006.

[43] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 134–143, New York, NY, USA, 2009. ACM.

[44] Paul E. McKenney and Jonathan Walpole. What is RCU?, Parts 1-3. `http://lwn.net/Articles/262464/`, `http://lwn.net/Articles/263130/`, `http://lwn.net/Articles/264090/`, 2007-2008.

[45] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 2011. to appear.

[46] ACC. `http://research.msrg.utoronto.ca/ACC`.

[47] I. Molnar and A. van de Ven. *Runtime locking correctness validator*, 2006. `www.kernel.org/doc/Documentation/lockdep-design.txt`.

[48] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228. Springer-Verlag, 2002.

[49] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, systematic, and efficient code replacement for running Java programs. In *Proceedings of the ACM EuroSys Conference*, Glasgow, Scotland, UK, April 2008.

[50] J. Olsa. [PATCH 0/2] net: fix race in the receive/select, June 2009. `https://lkml.org/lkml/2009/6/29/216`.

[51] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, University of Cambridge Computer Laboratory, March 2009.

[52] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 25–36. ACM, 2009.

[53] Aspicere. `http://sailhome.cs.queensu.ca/~bram/aspicere`.

[54] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[55] O. Rohlik, A. Pasetti, V. Cechticky, and I. Birrer. Implementing adaptability in embedded software through aspect oriented programming. *IEEE Mechatronics & Robotics*, pages 85–90, 2004.

[56] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analaysis of data races and atomicity. In *Proceedings of the Tenth ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–94, June 2005.

[57] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. ERASER: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[58] A. P. Sistla and A. R. Srinivas. Monitoring temporal properties of stochastic systems. In *Proceedings of the Ninth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'08)*, pages 185–212, San Francisco, CA, USA, 2008.

[59] A. P. Sistla, M. Zefran, and Y. Feng. Monitorability of stochastic dynamical systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV 2011)*, volume 6806 of *Lecture Notes in Computer Science*, pages 720–736. Springer, 2011.

[60] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Know.-Based Syst.*, 20(7):636–651, 2007.

[61] S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok. Runtime verification with state estimation. In *Proc. 2nd International Conference on Runtime Verification (RV'11)*, San Fransisco, CA, September 2011. (**Won best paper award**).

[62] Subrata Modak. Linux Test Project (LTP), 2009. `http://ltp.sourceforge.net/`.

[63] Valgrind. `http://valgrind.org`.

[64] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 334–345. ACM, 2006.

[65] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *Proceedings of the 6th ESEC/SIGSOFT International Symposium on Foundations of Software Engineering (FSE '07)*, pages 205–214. ACM, 2007.

[66] R. Walker and K. Viggers. Implementing protocols via declarative event patterns. In R. Taylor and M. Dwyer, editors, *ACM Sigsoft 12th International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159–169. ACM Press, 2004.

[67] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Proceedings of the Third Workshop on Runtime Verification (RV)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*, pages 191–209. Elsevier, July 2003.

[68] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, 2006.

[69] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 1–14. ACM, 2005.

[70] E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000. USENIX Association.