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

A Dissertation Proposal 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

Technical Report FSL-12-01

January 2012

Abstract of the Dissertation Proposal

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 comprehensive testing.

Runtime verification is a powerful technique for finding concurrency errors. Existing runtime verification tools can check for 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 only detect violations that actually occur, but they are less prone to false positives and computationally faster, making them useful for 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.

We propose to extend our instrumentation to support online analysis, along with a modified scheduler that yields to threads that are likely to violate concurrency properties, catching infrequent errors sooner. We will also explore bounded-overhead monitoring, which integrates overhead control and state estimation. The overhead controller will enforce an overhead bound by ignoring some events, and state estimation will make it possible to check concurrency properties even though some events go unobserved. Finally, we will develop analyses for memory model errors, which are especially difficult to find in testing because they occur under rare circumstances. This direction is motivated by our thesis that software designed for highly parallel systems will require ever more sophisticated verification to expose its most subtle concurrency 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, and Mandar Joshi. Their sweat is in here too.

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.

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 [40].

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 [17,51] checks for data races by verifying each variable's *lock discipline*. A lapse in lock discipline, meaning a variable that is not consistenly 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 [59, 60] 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.

We propose an algorithm to detect a specific kind of error, which we call a *star-crossed data race*, that involves a pair of threads coordinating state without locking. A star-crossed data race can lead to a sequentially inconsistent execution on most architectures, including 32- and 64-bit x86, and we have observed reports of this type of error in more than one system, including the Linux kernel [43]. The details of our proposed algorithm are in Section 4.1.

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.

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.

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 4.2, we propose an atomicity verification algorithm that can run online in kernel context.

Our proposed 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 interefered 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.

We also intend to improve this trade-off by introducing schedule perturbations to make erroneous schedules more likely to occur during verification. Our proposed scheduler modifications will force atomic regions to yield, with the aim of widening race windows, and give priority to threads that are most likely to interfere with currently running atomic regions.

State Estimation

Finally, we focus on verification for environments in which overhead is the primary consideration. Our proposed tool for verifying lock discipline will integrate overhead control [27] and state estimation [54] to design a monitor that can operate effectively within any overhead requirements.

SMCO is a technique that allows a user to choose an explicit overhead bound for a runtime monitor. A controller enforces the bound, turning monitoring off when necessary to stay within the target overhead. Because monitoring is sometimes disabled, the monitor will experience “gaps” in its observations of program events.

State estimation is a technique designed to cope with these gaps. The monitor uses a system model to infer which events might have occured during gaps, allowing it to estimate the probability that the monitored property holds for an overheaad-controlled execution.

Chapter 2

Offline Analysis of Kernel Concurrency

As the kernel underlies all of a system's concurrency, it is the most important front for eliminating concurrency errors. In order 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 [21] 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 [50] 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. It has two main parts:

1. *Fast Kernel Logging* uses compiler plug-ins to provide *modular* instrumentation that targets specific data structures in specific kernel subsystems for logging. It reserves an in-memory buffer to log operations on the targeted data structures with the best possible performance.

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

Currently, Redflag implements two kinds of concurrency analyses: *Lockset* [51] analysis for data races and *block-based* [60] analysis for atomicity violations. We developed several enhancements to improve the accuracy of these algorithms, including *Lexical Object Availability* (LOA) analysis, which eliminates false positives caused by complicated initialization code. We also augmented Lockset to support Read-Copy-Update (RCU) [37] synchronization, a synchronization tool new to the Linux kernel.

2.1 Design

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. Compiler plug-ins execute during compilation and have direct access to GCC's intermediate representation of the code [11]. Redflag's GCC plug-ins search for relevant operations and instrument them with function calls that serve as hooks into Redflag's logging system. These logging calls pass information about instrumented events directly to the logging system as function arguments. For an access to a field, the logging call passes, among other things, the address of the struct, the index of the field, and a flag that indicates whether the access is a read or write.

We need to log four types of operations for our current analyses: (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 `structs` 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. Redflag can list all `structs` and global locks defined in a directory; this provides a useful starting point for most components.

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 [51].

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. The corresponding release removes the lock.

When an access to a variable is processed, the variable's candidate lockset is refined by intersecting it with the thread's current lockset. In other words, the algorithm sets the variable's candidate lockset to be the set of locks that were held for *every* access to the variable. When a candidate lockset becomes empty, the algorithm revisits every previous access to the same variable, and 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 positives (false alarms) due to pairs of accesses that do not share locks but cannot occur concurrently for other reasons. We discuss two of these next.

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.

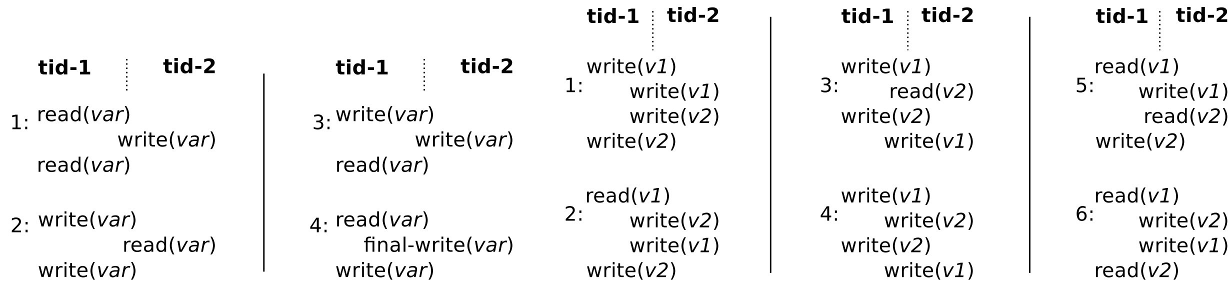
The Eraser algorithm solves this problem by tracking which threads access variables to determine when each variable become shared by multiple threads [51]. Note that this makes the algorithm more sensitive to thread schedule in the monitored execution. We implement a variant of this idea: when a variable 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, allocating new data structures in the same memory can cause false positives 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 [51]. Redflag also logs calls to allocation functions, so that it can similarly account for reuse. In the kernel, it is also necessary to track remapping of physical pages into virtual memory. Redflag logs calls to `kmap`, which creates new virtual mappings, and treats new mappings as new allocations.

2.1.3 Block-Based Algorithms

Redflag includes two variants of Wang and Stoller's block-based algorithm [59, 60]. 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 in parallel 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



(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 [60].

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 more than one variable.

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 a block if it is made without holding any of the block's locks, and the interleaving is unserializable if it matches any of the patterns in Figure 2.1(a).

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 include more detail about the locking information saved for each block [60].

Together, these two variants are sufficient to determine whether any two syscalls in a trace can violate each other's atomicity [60]. In other words, these algorithms can detect atomicity violations involving any number of variables.

Analogues of the refinements to Lockset described in Section 2.1.2 are used in the block-based algorithm to eliminate false positives due to infeasible interleavings.

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 were not addressed by previous work on detecting concurrency violations. This section discusses two new synchronization methods that Redflag handles: multi-stage escape and RCU.

Multi-stage escape. As explained in Section 2.1.2, objects within their initialization phases are effectively protected against 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, inode objects go through two stages of escape. First, after a short first-stage initialization, the inode gets 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 any file syscalls. Interleavings are possible with functions that traverse the superblock's inode list, such as 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 accesses for each targeted `struct`. Intuitively, the LOA relation encodes observed ordering 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, any access from a file syscall serves as evidence that both first- and second-stage initialization are finished, meaning that accesses from those initialization routines are no longer possible. An access from the writeback thread is weaker evidence, showing that first-stage initialization is finished.

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 `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 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.

Although we designed LOA analysis specifically for multi-stage escape, it can also infer other kinds of order-enforcing synchronization. For example, we found that the kernel sometimes uses condition variables to protect against certain operations to inodes that are in a startup state, which lasts longer than its initialization. We constructed the happened-before relation [31] to determine which potentially interleavings were precluded by condition variables, but we found that all these interleavings were already filtered by LOA. LOA analysis can also infer *destruction* phases, when objects typically return to being exclusive to one thread.

Because LOA filters interleavings based on the observed order of events, it can cause false negatives (i.e., 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 has the information it needs to know that such an access is possible. Dynamic escape analysis addresses this problem by determining precisely when an object becomes accessible to other threads [60], but it accounts for only one level of escape.

Syscall interleavings. Engler and Ashcraft observed that dependencies on data prevent some kinds of syscalls from interleaving [21]. For example, a `write` operation on a file never executes in parallel with an `open` operation on the same file, because userspace programs have no way to call `write` before `open` finishes.

These kinds of 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 only from one syscall, our LOA analysis already rules out impossible interleavings between 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 to contain entries of the form $((syscall, statement), (syscall, statement))$. As a result, LOA analysis treats a function called from different syscalls as separate functions. Statements that do not execute in a syscall are instead paired with the name of the kernel thread they execute in. The augmented LOA relations can express dependencies caused by both multi-stage escape during initialization and dependencies among syscalls.

RCU. Read-Copy Update (RCU) synchronization is a recent addition to the Linux kernel that allows very efficient read access to shared variables [37]. A typical RCU-write first 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 does not protect against lost updates, so writers must use their own locking. A reader needs only to surround read-side critical sections with `rcu_read_lock()` and `rcu_read_unlock()`. These functions ensure that the shared data

```
/* [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.

structure does not get freed during the critical section.

We extended our Lockset implementation to test for correctness of RCU use. When a thread enters a read-side critical section by calling `rcu_read_lock()`, the updated 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, as RCU synchronization alone does not protect against racing writes.

2.1.5 Filtering False Positives and Benign Warnings

This section describes types of false positives and benign violations that Redflag filters out.

Bit-level granularity We found that many false positives in 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 any individual 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 [22].

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 conditional 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 positives 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 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 violations 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_write()` function in the Btrfs file system 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 does not need to be.

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. The user can examine these potential fenceposts to see whether they lie on the boundaries of logical atomic regions in the code.

2.2 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. 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 [55], 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

	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 2.1: 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.

resulting from it: parallel writes to a file may write their data to the same location in a file, in violation of POSIX requirements [16]. 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.

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 well 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 positives due to untraced locks: 2 for Wrapfs, and 11 for Noveau. These false positives 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 2.1 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 test). These reports were produced because we told Redflag to instrument all accesses to targeted VFS structures, but they are not relevant to Wrapfs development.

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

Table 2.2: Number of false positives filtered out by various techniques

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 instrument.

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.

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. This can cause false positives, in the same way that bit-level operations can (*cf.* Section 2.1.5). These false positives can be eliminated by adding the non-targeted struct to the list of targeted structs.

Filtering. Table 2.2 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. The "unfiltered" column shows the number of reports not filtered by any of these techniques. 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 positives 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 just after the inode is linked to a dentry. This linking is the inode's second stage of escape, as discussed 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 file operations vector initialization 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 positive.

Combining dynamic escape analysis with LOA analysis may improve the accuracy of both. We are investigating ways to relax the LOA relation, allowing it to consider some interleavings it has not witnessed, using information about dynamic escape to avoid relaxing too much. We are also considering how best to incorporate active analysis, which perturbs the schedule at runtime, forcing threads to yield after newly created objects escape, so that LOA analysis observes more interleavings.

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.

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 of $2.44\times$ for an instrumented kernel without logging, and $2.65\times$ for an instrumented kernel 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.

Schedule sensitivity of LOA. Although LOA is very effective at removing false positives, it is sensitive to the observed ordering of events, potentially resulting in false negatives, as discussed in Section 2.1.4. We evaluated LOA's sensitivity to event orderings by repeating a workload 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.

2.3 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.

Runtime race detection Our Lockset algorithm is based on the Eraser algorithm [51]. Several other variants of Lockset exist, implemented for a variety of languages. LOA analysis is the main distinguishing feature of our version. 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 [36].

Microsoft Research's DataCollider [22] is the only other runtime data race detector that has been applied to an OS kernel, to the best of our knowledge. Specifically, it has been applied to several modules in the Windows kernel and detected numerous races. It detects actual data races when they occur, in contrast to Lockset-based algorithms that analyze synchronization to detect possible races. At runtime, DataCollider pauses a thread about to perform a memory access and then uses hardware watchpoints to intercept conflicting accesses that occur within the pause interval. This approach produces no false positives but may take longer to find races and may miss races that happen only rarely. DataCollider uses sampling to reduce overhead.

Static analysis Static analysis tools, typically based on the Lockset approach of finding variables that lack a consistent locking discipline, have uncovered races even in some large systems. For example, RacerX [21] and RELAY [57] found data races in the Linux kernel. Static race detection tools generally produce many false positives, due to the well-known difficulties of analyzing aliasing, function pointer values, calling context, etc.

Static analysis of atomicity has been studied (e.g., [25, 50]) but not applied to large systems software. Generally, these analyses check whether the code follows certain safe synchronization patterns. Static analysis of atomicity, like static analysis of races, often produces numerous false positives.

Runtime atomicity checking. To the best of our knowledge, no runtime atomicity checker, other than ours, has been applied to components of an OS kernel. Although we used the block-based algorithms, there are other runtime techniques for checking atomicity—or similar properties—that could be adapted to work on Redflag's logs. Atomicity checkers based on Lipton's reduction theorem [34], such as the algorithms described by Flanagan [24] and Wang [60], are computationally much cheaper than the block-based algorithms, because they check a simpler condition that is sufficient but not necessary for ensuring atomicity. As a result, however, they usually produce more false positives.

AVIO [35] and CTrigger [45] use heuristics to infer programmers' expectations about atomicity, and then check for violations thereof (i.e., atomicity violations). An important difference from our work is that the block-based algorithm reports potential and actual atomicity violations, while AVIO and CTrigger report only actual atomicity violations (i.e., atomicity violations that manifest in the monitored run). They actively perturb the schedule to increase the likelihood that atomicity bugs will manifest during testing. Also, they do not detect atomicity violations involving multiple variables. As a result, they are computationally cheaper and produce fewer false positives, but they are more schedule-sensitive and may miss bugs that the block-based algorithms would report. Their implementations use binary instrumentation and are not integrated with the compiler, so it would be difficult to target their analysis to specific data structures.

Logging Feather-Trace uses a lock-free buffer similar to ours to log kernel events [9]. 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.

2.4 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 any OS kernel.

Redflag's runtime analyses are designed to 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 that 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 [11], 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 [26] 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 runtime 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.

We also present the InterAspect *Tracecut extension* to generate program monitors directly from formally specified tracecuts. A tracecut [58] 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 [29].

To illustrate InterAspect's practical utility, we have developed a number of 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.

The rest of the article is structured as follows. Section 3.1 provides an overview of GCC and the InterAspect framework. Section 3.2 introduces the InterAspect API. Section 3.3 presents the three case studies: heap visualization, integer range analysis, and code coverage. Section 3.4 describes how we extended InterAspect with a tracecut system. Section 3.5 summarizes related work, and Section 3.6 concludes the article. A preliminary version of this article, which did not consider the tracecut extension, appeared last year [52].

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 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.

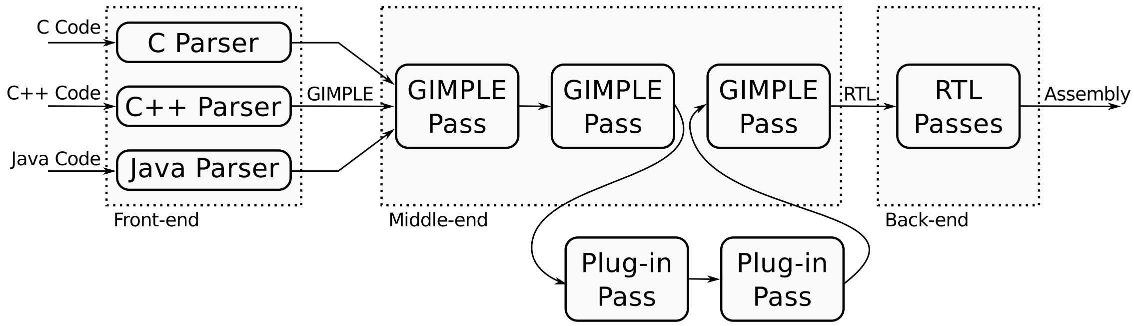


Figure 3.1: A simplified view of the GCC compilation process

```
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)

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,

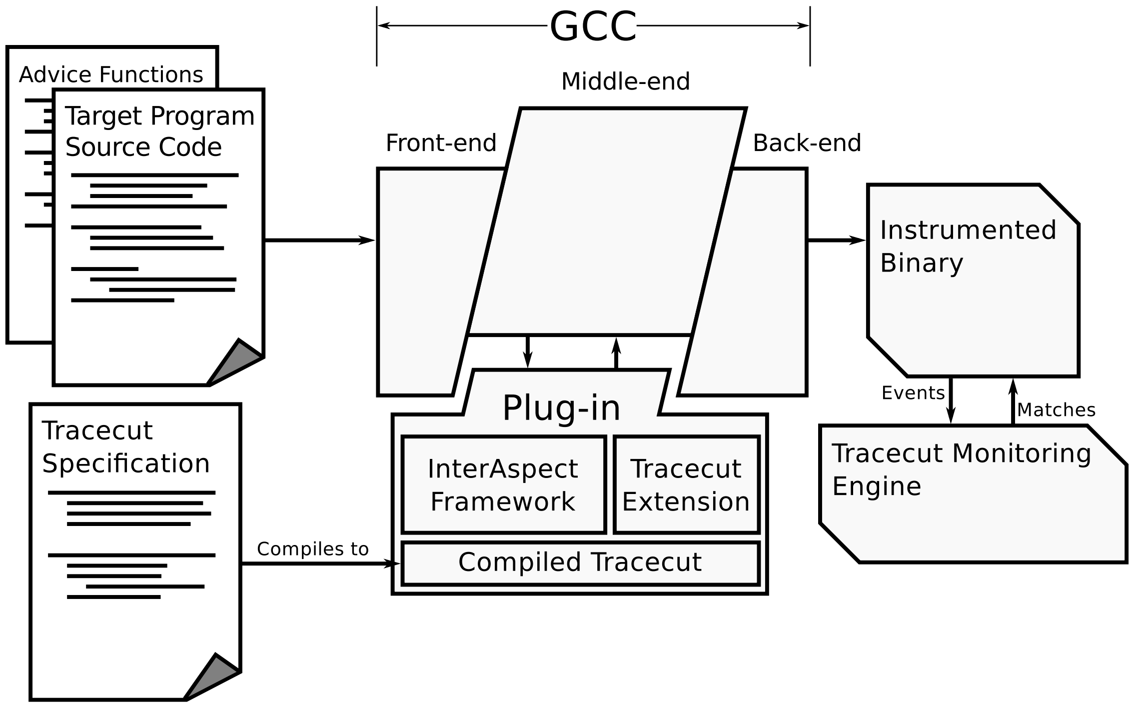


Figure 3.3: Architecture of the INTERASPECT framework with its tracecut extension
The tracecut specification is a simple C program. The tracecut extension translates events in the specification to pointcuts, and the INTERASPECT framework directly instruments the pointcuts using GCC's GIMPLE API. The 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

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 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`.

```
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

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.

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.

```
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

AOP systems like AspectJ [30] 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 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 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

```
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

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 API 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.

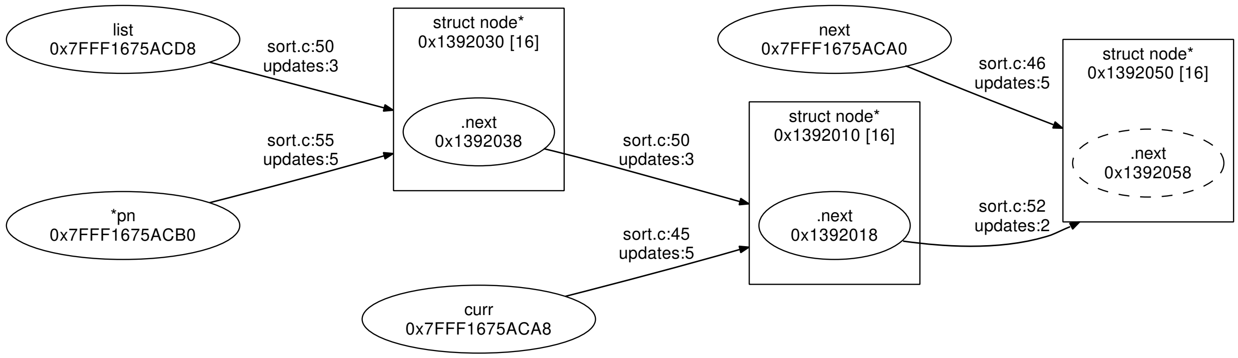


Figure 3.10: Visualization of the heap during a bubble-sort operation on a linked list
Boxes represent heap-allocated `structs`: linked list nodes in this example. Each `struct` is labeled with is size, its address in memory, and the addresses of its field. Within a `struct`, ovals represent fields that 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.

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 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` calls `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;
```

```
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

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 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 [23]. A range analyzer can learn normal ranges from

```
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

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 many cases. Sampling can be done randomly or by using a technique like Software Monitoring with Controllable Overhead [27].

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.

```
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 `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.

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 IN-

```
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

TERASPECT, its instrumentation component is built modularly on INTERASPECT: all of its access to GCC are through the published INTERASPECT interface.

Whereas pointcut advice is triggered by individual events, tracecut advice can be triggered by sequences of events matching a pattern [58]. 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, 12]. 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 [12].

```
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 [12]. (MOP has since defined semantics for partially parameterized monitors [38].)

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 call to `VEC_gimple_base_quick_pus` that we deliberately placed in an iterator loop.

3.5 Related Work

Aspect-oriented programming was first popularized for Java with AspectJ [19,30]. 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 [8]. Similarly to INTERASPECT, it manipulates a 3A intermediate representation (Jimple) specialized to Java.

Other frameworks for Java, including Javaassist [14] and PROSE [42], 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 [15] was one of the first AOP systems for C, based on the language constructs of AspectJ. ACC [39] 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 [49], 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 [47] 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 [18, 20] is a dynamic aspect language for C that uses assembler manipulation techniques to instrument a running system without pausing it.

AspectC++ [53] 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++ [13] 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 [41] (C Intermediate Language) is an OCaml [28] 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 [56], which works directly with executables and consequently targets multiple programming languages.

INTERASPECT Tracecut is informed by several runtime monitoring systems, including Declarative Event Patterns [58], which introduced the term *tracecut*. Monitor parameterization is based on the monitor implementations in Tracematches [2] and MOP [12]. 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 [18]. 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 [29]. 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

Proposed Work

In addition to the methods we discussed in Chapter 2, we believe that there is still great potential in applying verification to concurrency in the systems space. In this chapter, we propose several ways to extend our work so far. Section 4.1 introduces a new analysis to check for concurrency errors arising in common weak memory models. Section 4.2 discusses our plan to adapt Redflag to verify atomicity online, during execution. Finally, Section 4.3 proposes to integrate this online monitoring with two other efforts, overhead control and state estimation, to design a monitor that is practical to use in environments with strict overhead requirements.

4.1 Weak memory model errors

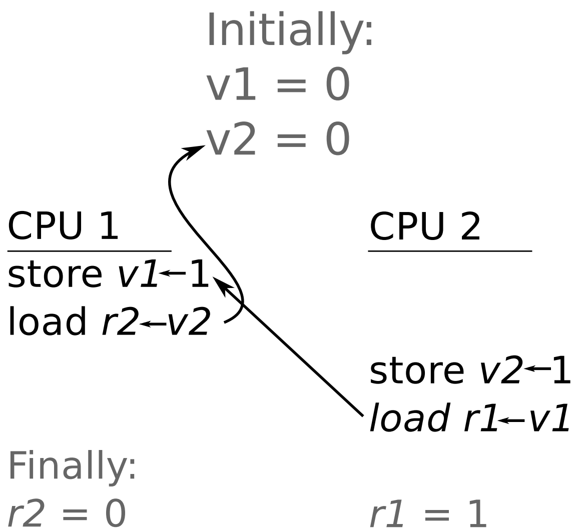
Weak memory models are an oft overlooked source of concurrency errors in systems code. Under weak memory models, the compiler and processor can reorder memory accesses for performance reasons. Reorderings are invisible to single-threaded code, but developers of multi-threaded programs must account for 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 [32]. Furthermore, the total order is consistent with each processor's order of execution. Figure 4.1(a) shows a sequentially consistent trace of a simple parallel program.

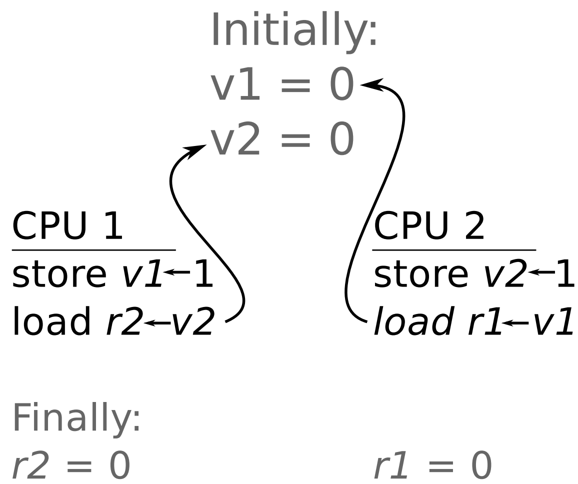
Weak memory models are those models that do not provide the sequential consistency guarantee. Either the global ordering of memory events may be inconsistent with some processor's execution order, or there might not be a global ordering. Figure 4.1(b) shows the same program executing on a weak memory 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 4.1(b) is possible, though, on architectures that implement *store buffering* because store buffers do not enfore 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,



(a) Sequentially consistent



(b) Violating sequential consistency

Figure 4.1: 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) [10].

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 [10, 44].

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 4.1 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 [43]. Burckhardt and Musuvathi found a similar producer/consumer bug in a Microsoft concurrency library [10].

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 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 4.1(b): one in each thread between the `store` and the `load`. Linux developers chose this approach to fix the previously mentioned network socket bug [43]. 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 usesful.

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

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

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 4.1(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 4.1, 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.

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 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.

4.2 Online analysis

Approaches for detecting atomicity violations discussed so far rely on examining an execution log once the program is finished running. This chapter presents a technique that detects atomicity violations online, as they occur.

Our online approach focuses only on the actual schedule, making it resilient to false positives. Atomicity checkers like the block-based algorithms widen their search radius by speculating on alternate schedules. But without sophisticated reasoning about schedule feasability, such as with our LOA analysis technique, speculation on infeasible schedules generates false positives.

Though checks without schedule speculation are less likely to catch rare bugs, online analysis can partly compensate for this because it can check longer runs than offline checkers, which must store large execution logs. We also plan to perturb the thread schedule to help expose these rare atomicity violations, as discussed below.

The single-variable block-based analysis provides straightforward criteria for what interleavings represent atomicity violations, which we use as the basis for our online atomicity checker. In one style of atomicity violation, a remote write interferes with a variable's value while it is in use by another thread (cases 1 and 3 in Figure 2.1(a)). When the write interleaves a pair of reads, the two reads observe different values, violating atomicity. Similarly, when the write interleaves a block comprising a read following a write, the value observed by the read does not match the value originally by the write.

To detect this style of violation using our online approach, each atomic region gets its own *shadow memory* that maintains isolated copies of every targeted field that the region accesses. Some software transactional memories use a similar kind of private memory to defer updates until an atomic region commits [33, 46]. Shadow memory does not defer writes to the global 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 update both the shadow and global memories. A read from the field triggers a check that the value in global memory matches the shadow copy. Any discrepancy means that an interleaved operation in another thread modified the field and indicates an atomicity violation.

A second style of atomicity violation (case 2 in Figure 2.1(a)) involves a pair of writes to the same variable. The first write stores an intermediate value that should not be visible to other threads. Any interleaving read by another thread exposes the first value, causing an atomicity violation.

For this style of violation, we add a mechanism for marking writes which must be *final* because they have been observed by a remote thread. If such a write proves to not be final, it is an atomicity violation. Each instrumented read searches for open atomic regions in other threads that have written to the same field, marking their shadow copies as final. A *written* flag, which is set when a write within an atomic region updates a shadow copy, makes it possible to determine which atomic regions have written the field. If an atomic region writes a field marked as final in its shadow memory, we report the violation.

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

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

We plan to study several schedule perturbation techniques to find which ones are most effective at exposing atomicity violations. The goal of perturbing the schedule is to widen the window between two operations in a block so that other threads have more opportunity to execute conflicting interleaved operations. A modified scheduler creates these wider windows by forcing threads to yield when they would normally continue to execute.

Yields should be inserted at vulnerable points, such as immediately after lock releases and interrupt enabling instructions. These events open up more variables for concurrent accesses and allow more threads to execute. For catching violations that do not involve a data race, yielding immediately before or after memory accesses is a poor strategy. An offending variable will always be protected by a lock when accessed, so no amount of waiting will allow a remote thread to perform an interleaved access.

Our perturbation strategy will focus on inducing violations in a single atomic region at a time and will only force that region to yield. Forcing other threads to yield could prevent them from executing conflicting operations during the focused region's widened violation windows.

When yielding, the perturbation strategy also decides which thread to yield to, prioritizing threads that are more likely to execute conflicting operations. We plan to develop models to predict what variables each thread is likely to access based on recent history. When preempting an atomic region, our scheduler will prioritize threads that are likely to access variables in the current thread's active set.

Temporal locality suggests a simple model for predicting which variables a thread is likely to access. The shadow memory can maintain the set of most recently used variables, under the assumption that they will probably be accessed again. We can augment this model by training it as the system runs. The training can learn how often each instrumented memory access instruction is followed by another access to the same variable. Among variables in the most recently used set, those accessed by an instruction with a high follow-up probability are especially likely to be accessed again.

4.3 State Estimation

Methods for managing overhead often sacrifice monitoring accuracy, but these less accurate monitors have the opportunity to run in many more environments. The analysis in the previous section is designed for a testing environment, where high overheads are an acceptable cost for the ability to find bugs. To monitor production systems, dramatically less overhead is a requirement, even if at the cost of accuracy.

Achieving these low overheads usually necessitates a sampling monitor that ignores some system events. To guarantee acceptable overheads for any environment, we intend to apply an *overhead control* [27] technique that allows the user to set the maximum monitoring overhead. We also propose the use of *state estimation* [54] to produce applicable results from a monitor that misses system events. State estimation uses a model to make inferences about how missed events might have affected the system in order to compute the probability that monitored properties hold. The primary contribution of this work will be the integration of these two techniques, overhead control and state estimation.

4.3.1 Lock Discipline Property Formulation

Our overhead-controlled Redflag will verify a *lock discipline* property for targeted `structs`, which is related to the more general property verified by Lockset. We define this property for a `struct` S, which has a lock, protected fields, and unprotected fields. Informally, the property requries that all accesses to protected fields occur while the lock is held.

We formally define the lock discipline property as a regular expression. First, we define four events on S, $LOCK_S(t,o)$, $UNLOCK_S(t,o)$, $ACCESS^p_S(t,o)$, $ACCESS^u_S(t,o)$, where t is a thread and o is an instance of S. $LOCK_S$ and $UNLOCK_S$ correspond to acquiring and releasing the lock member of S. $ACCESS^p_S$ and $ACCESS^u_S$ correspond to accessing a protected field or an unprotected field of S, respectively. The property takes the form:

$$\begin{aligned}\phi^{LD}_S = (ACCESS^p_S(t,o)|ACCESS^u_S(t,o))* \\ (LOCK_S(t,o)(ACCESS^p_S(t,o)|ACCESS^u_S(t,o))* \\ UNLOCK_S(t,o)ACCESS^u_S*(t,o))*\end{aligned} \tag{4.1}$$

With one exception, this property requires each $ACCESS^p_S$ to occur between $LOCK_S$ and $UNLOCK_S$, but $ACCESS^u_S$ is allowed anywhere. We assume that accesses to o by t before any $LOCK_S(t,o)$ operation are part of the initialization of o, so the property does allow $ACCESS^p_S(t,o)$ before the first $LOCK_S(t,o)$. The property also requires that every $LOCK_S$ is eventually matched with an $UNLOCK_S$ and that $LOCK_S, UNLOCK_S$ pairs do not nest.

To clarify Formula 4.1, it helps to separately look at its two phases. The first phase continues until the first $LOCK_S$ event, and any kind of access is allowed:

$$(ACCESS^p_S(t,o)|ACCESS^u_S(t,o))* \tag{4.2}$$

In the second phase, thread t holds a lock on the object o, and any kind of access is allowed until there is an $UNLOCK_S$ event, after which only $ACCESS^u_S$ access events are allowed:

$$LOCK_S(t,o)(ACCESS^p_S(t,o)|ACCESS^u_S(t,o))*UNLOCK_S(t,o)ACCESS^u_S*(t,o) \tag{4.3}$$

Note that, in Formula 4.1, this phase repeats when the next $LOCK_S$ event occurs (by way of a Kleene closure).

Formulating the property ϕ^{LD}_S as a deterministic finite-state automaton (DFA) allows for an efficient means of checking whether an execution violates the property. We show this DFA in Figure 4.2.

4.3.2 Monitoring

Monitoring ϕ^{LD}_S involves maintaining an instance $M^{LD}_S(t,o)$ of the DFA for each instance of (t,o). Each operation by a thread t on an object o changes the state of $M^{LD}_S(t,o)$ according to the DFA's

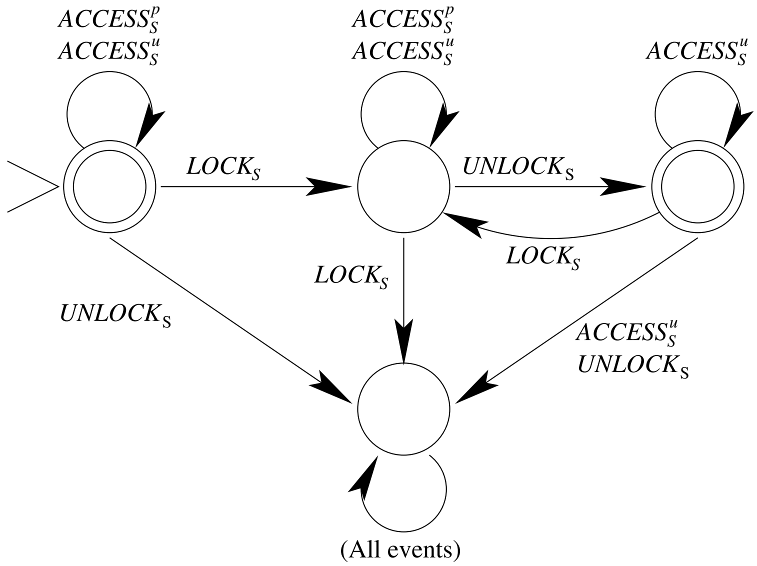


Figure 4.2: A DFA corresponding to the lock-discipline property in Formula 4.1

transition function. Going by the DFA in Figure 4.2, each monitor in a correct execution will stay in the top three states: the initial state, representing a thread that is initializing an object; the middle state, representing a thread that holds a lock on an object; and the right-most state, representing a thread that has released its lock on an object. Any error, including double locking, double unlocking, and an unlocked access to a protected field will transition the monitor instance to the bottom trap state. This strategy of monitoring with parameterized DFAs is the same as we used for INTERASPECT Tracecut in Section 3.4.

Monitoring with a DFA only works when the monitor can observe every event, however. Missed events are likely to cause false positives or false negatives. For example, if sampling ignores a $LOCK_S$ event, a subsequent $ACCESS_S^p$ will appear to be a violation.

4.3.3 Inferring Unobserved Events

State estimation will allow us to use M_S^{LD} to verify ϕ_S^{LD} even when events go unobserved [54]. The state estimation technique uses a learned model of system behavior to estimate how unobserved events may have affected the monitoring result.

The model itself is a Hidden Markov Model (HMM) learned from complete traces collected during test runs without sampling. We will use a standard learning algorithm to construct a model HMM_S from these traces, with each transition in HMM_S representing an internal state change to an object o that causes a transition in $M_S^{LD}(t, o)$ for some t.

When monitoring a trace with unobserved events, we will then use a modified version of the *forward algorithm* [48] described in Stoller et al. [54], that takes HMM_S, along with the probability that unobserved events occured since the last monitored event, and determines the probability for each state in $M_S^{LD}(t, o)$. The monitor will then be able to sum the probabilites of non-accepting states to determine the overall probability that a violation occurred.

Figure 4.3 shows a manually designed HMM for operations by thread t on an object o that will satisfy ϕ_S^{LD}. Modifying the HMM to include $ACCESS_S^p$ in the bottom state allows it to produce

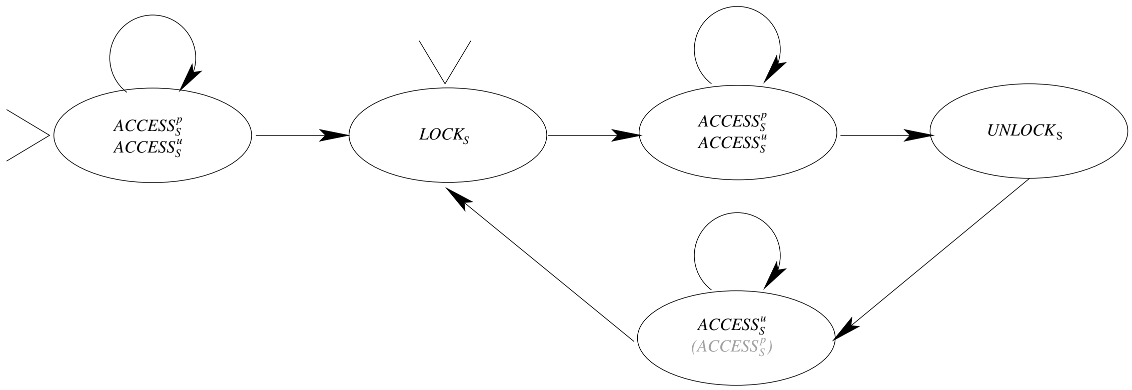


Figure 4.3: Manually-generated HMM for correct thread behavior

an erroneous access with some probability. (This addition is shown in gray.) We intend to use this model to evaluate HMMs generated by learning algorithms. Probabilites are not shown in Figure 4.3 because we intend to choose arbitrary probabilities for our evaluation.

A learned HMM will not be as structured as the HMM in Figure 4.3, which was designed with knowledge of the program's intended design, but detailed program structure is not necessary to make predictions useful for state estimation. For example, a learned model might observe that protected field `a` is usually accessed with unprotected field `b`. If an illegal access to `a` were to go unobserved, the HMM could predict that it occurred based on an observed access to `b`. This prediction would allow the monitor to report a high probability of a violation even if the reported violation never appeared in the HMM's training data.

4.3.4 Gap Distribution

The HMM can make its best predictions when it knows exactly where unobserved events, or *gaps*, fall. Unfortunately, even just recording gaps can lead to unacceptably high overheads. When there are multiple monitor instances, determining which instance is affected by an event is often the most expensive monitoring operation. While, it is practical to keep a count of unobserved events, any individual monitor instance does not know how many of those gaps belong to it.

We plan to consider ways to model how events are distributed across monitor instances. State estimation will then be able to take into account the probability that each gap is relevant to a particular monitor instance when it computes probabilies for each of the instance's states.

Because our monitor is partially parameterized by thread, we can improve our results by keeping a thread-local gap count. Even though the state estimation algorithm will not know which monitor instance a gap belongs to, it will know which *thread* it belongs to, narrowing down the possibilities.

4.3.5 Event Sampling

To implement sampling, we intend to use the overhead control mechanisms developed in "Software Monitoring with Controllable Overhead" (SMCO) [27], which allow the user to set a target overhead. The target overhead lets the user explicitly adjust the trade-off between monitoring overhead and confidence.

SMCO uses *feedback control*, computing the difference between the target overhead and actual monitoring overhead. When overhead is too high, SMCO disables monitoring, allowing events to go unobserved. The feedback controller computes the amount of time to disable monitoring so that the system meets its overhead goal.

Formula-Aware SMCO The current SMCO implementation attempts to allocate overhead evenly among objects in the system, but we could improve the odds of catching errors by favoring objects that are at high risk of violating the monitored property. We have two criteria in mind for a *formula-aware SMCO* algorithm to determine risk. The property formula itself provides one criteron: the fewer events necessary to transition a monitor instance to a failure state, the higher its risk. Additionally, the state estimation algorithm can be used to predict how likely those events are to occur, providing the second criterion.

Instrumentation We intend to use our INTERASPECT framework, described in Chapter 3, to instrument all events on S. With additional modification to allow targetting of pointcuts to `struct`s, INTERASPECT will make it straightfoward to develop plug-ins that capture events on S.

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.

We propose several new ways to approach these challenges. On top of the existing analysis we provide, we plan to implement a new technique for detecting star-crossed data races, which can execute incorrectly on architectures with weak memory models. Our online analysis tool will be able to verify long-running tests that execute many code paths and schedules. Finally, we will be able to extend verification to environments with strict overhead requirements using overhead control and state estimation.

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 [29].

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 absense 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 [27]. 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 [22].

We could augment the online atomicity checker in Section 4.2 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.

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 *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] E. Bodden and K. Havelund. Racer: Effective race detection using AspectJ. In *International Symposium on Software Testing and Analysis, Seattle, WA*, pages 155–165. ACM, 2008.

[9] 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.

[10] 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.

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

[12] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, 2007.

[13] 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.

[14] 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.

[15] 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*, pages 88–98, 2001.

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

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

[18] 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*. ACM Press, 2005.

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

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

[21] 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.

[22] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Berkeley, CA, USA, 2010. USENIX Association.

[23] 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/`.

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

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

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

[27] 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)*, 2011.

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

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

[30] 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.

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

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

[33] J. R. Larus and R. Rajwar. *Transactional Memory*, chapter 2: Programming Transactional Memory, pages 14–52. Morgan & Claypool, January 2006.

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

[35] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 37–48, New York, NY, USA, 2006. ACM.

[36] 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.

[37] Paul E. McKenney. *What is RCU?*, 2005. `http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.33.y.git;a=blob;f=Documentation/RCU/whatisRCU.txt`.

[38] 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.

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

[40] I. Molnar and A. van de Ven. *Runtime locking correctness validator*, 2006. `http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.33.y.git;a=blob;f=Documentation/lockdep-design.txt`.

[41] 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, London, England, 2002. Springer-Verlag.

[42] 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.

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

[44] 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.

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

[46] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 161–176, New York, NY, USA, 2009. ACM.

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

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

[49] 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.

[50] 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*, June 2005.

[51] 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.

[52] J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S. A. Smolka, S. D. Stoller, and E. Zadok. Aspect-oriented instrumentation with GCC. In *Proc. of the 1st International Conference on Runtime Verification (RV 2010)*, Lecture Notes in Computer Science. Springer, November 2010.

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

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

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

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

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

[58] 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.

[59] 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*. Elsevier, July 2003.

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