Stopping Data Races Using Redflag

A Thesis Presented
by

Abhinav Duggal

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

Technical Report FSL-10-02
May 2010

Stony Brook University

The Graduate School

Abhinav Duggal

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Dr. Erez Zadok, Thesis Advisor
Associate Professor, Computer Science

Dr. Scott Stoller, Thesis Committee Chair
Professor, Computer Science

Dr. Rob Johnson
Assistant Professor, Computer Science

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

Stopping Data Races using Redflag

by

Abhinav Duggal

Master of Science

in

Computer Science

Stony Brook University

2010

Although sophisticated runtime bug detection tools exist to root out all kinds of concurrency problems, the data they need is often not accessible at the kernel level; examining every potentially concurrent memory access for a system as central as a kernel is not feasible.

This thesis shows our runtime analysis system *Redflag* which brings these essential tools to the Linux kernel. Redflag has three components: custom GCC plug-ins for in kernel instrumentation, a logging system to record instrumented points, and at its core, an improved Lockset algorithm for the Linux kernel.

We used GCC plug-ins to instrument read and writes to global memory locations, memory allocations, and locks—including seldom-addressed locking primitives like RCU's . Our fast logging system can log any event caught by instrumentation. The logging system is also optimized using zero-copy I/O in the kernel, and various in-kernel optimizations for improved performance.

We customized the classic Lockset algorithm to prune false positives caused by subtle kernel synchronization. We present a number of techniques we applied to improve the accuracy of our analysis . We tested our system on several file systems including Wrapfs, Btrfs, and the kernel's VFS layer and found 2 real races and several benign races. We also injected data races in the kernel and our system was able to detect them accurately. Redflag's false positive rates are very low for most of the file systems.

Our system is versatile using a small automation language to make it easy to run and use. Redflag can help kernel developers in finding data races in the Linux kernel, and is easily applicable to other operating systems and asynchronous systems as well.

To my parents and my sister Shveta.

Contents

List of Figures

List of Tables

Acknowledgments

My immense gratitude to my advisor, Prof. Erez Zadok, for his constant guidance and motivation. I thank Prof. Scott Stoller and Prof. Rob Johnson for being on my defense committee, and providing valuable suggestions. I am thankful to everyone at the File Systems and Storage Lab (FSL), for making it a great learning experience. I also want to specially thank Justin Seyster and Prabakar Radhakrishnan, for their help through the project.

This thesis was made possible thanks to NSF award number CNS-0509230.

Chapter 1

Introduction

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 prevent race conditions from going undetected until they cause real problems in a production system. Understanding concurrency in the kernel is difficult. Unlike many user-level applications, almost the entire kernel must run in a multi-threaded context, and much of it is written by experts who rely on intricate synchronization techniques.

Static analysis tools like RacerX [13] can thoroughly check even large systems code bases for potential data race errors, but they cannot easily infer certain runtime properties like aliases. Also, they require manual annotation to relate impossible schedules like pagefault handler being invoked from any code in the kernel and schedules related to interrupt handlers and bottom halves.

Runtime analysis provides additional tools for finding problems in multi-threaded code that are powerful and flexible. They can target many kinds of concurrency errors, including data races [12, 32] and atomicity violations [14, 38]. We designed the *Redflag* system with the goal of airlifting these tools 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 to target specific kernel subsystems for logging. It then interacts with the file system to log all operations in the instrumented module with the best possible performance.

2. The offline Redflag analysis tool performs several post-mortem analyses on the resulting logs in order to automatically detect potential race conditions that are apparent from the log. This tool is designed to be versatile with the ability to support many kinds of analysis.

Currently, Redflag implements two kinds of concurrency analysis: *Lockset* [32] and block based [38]. We believe that these two types of analysis cover a wide range of potential bugs, from missing locks to complex interleavings involving multiple variables. We contribute several modifications to improve the accuracy of these algorithms, including *Lexical Object Availability* (LOA) analysis, which detects false positives caused by complicated initialization code. We added support for RCU synchronization, a new synchronization tool in the Linux Kernel. This thesis concentrates on concurrency analysis using Lockset only.

The rest of the thesis is organized as follows. In Chapter 2, we explain how our modular instrumentation and logging system works, and we describe our implementation of the Lockset Algorithm, along with the improvements we made to its accuracy. In Chapter 3, we present the results of our analysis of several kernel components and evaluate the performance of our instrumentation and logging. We discuss related work in Chapter 4. We conclude in Chapter 5 and discuss future work in Chapter 6.

Chapter 2

Design

Four goals underlie our design of the Redflag system: versatility, modularity, performance, and accuracy. Our decision to perform all analysis post-mortem is motivated by the goals of versatility and performance: once an execution trace is ready, it is possible to run any number of runtime analysis algorithms to search for different classes of concurrency errors. We discuss the Lockset algorithm here, even though our system also supports finding atomicity violations using block based algorithm. Besides these many other algorithms could potentially operate on the same log files. We also seek to make the algorithms themselves versatile; our implementations can take into account a variety of synchronization mechanisms used in the kernel.

Modularity is an important design goal because of the size of a modern kernel. Developers working on large projects are typically responsible for individual components. An on-disk file system developer may not be interested in potential errors in the networking stack. On the other hand, a network file system developer might be, and it is possible to target several components for one trace. Logging is broken down by data structure. The developer chooses specific data structures to target, allowing fine granularity in choosing which parts of the system to log.

Modularity is also crucial for system performance: the overhead from logging every event in the kernel would render Redflag's logging impractical. Even with targeted logging, however, a kernel execution can generate a large number of events for logging. Performance is therefore still a concern, and our in-kernel logging is designed to keep overheads low. When an instrumented event occurs, the system stores the event in a queue, so that it can immediately return control to the instrumented code and defer expensive I/O until after logging is complete.

Finally, because accuracy is important to make Redflag useful for developers, we implemented several measures to reduce the number of false positives that the system reports. We also ensured that Redflag can find errors by inserting bugs into the kernel and verifying that they result in error reports.

In Section 2.1, we explain how Redflag's logging targets specific components for logging. Section 2.2 explains the basic Lockset algorithm and improvements we made to Lockset. Section 2.3 discusses our stack trace support. Section 2.4 discusses further improvements to handle more kinds of synchronization.

2.1 Instrumentation and Logging

We are able to insert targeted instrumentation using a suite of GCC compiler plug-ins that we developed specifically for Redflag. Plug-ins are a new GCC feature which we have developed over the past few years. GCC Plugins were formally introduced in the April 14, 2010 public release of GCC 4.5 [17]. Compiler plug-ins execute during compilation and have direct access to GCC's intermediate representation of the code [5]. Redflag's GCC plug-ins search for relevant events and then 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. In the case of 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 if the access was a read or write.

For most concurrency analysis, we need to log 4 types of operations:

1. Field accesses: read from or write to a field in a `struct`.
2. Synchronization events: 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. (Though we can track deallocation events, they are not necessary for most analysis.)
4. Syscall boundaries: syscall entrance/exit, used by the block-based algorithm for checking syscall atomicity.

When compiling the kernel with the Redflag plug-ins, the developer provides a list of `struct`s to target for instrumentation. Field accesses and lock acquire/release operations are only instrumented if they operate on a targeted `struct`, resulting in a smaller, more focused trace. 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`, so the developer can also directly target these global locks by name. To simplify the process of targeting data structures, Redflag provides a script that searches a directory for the `struct`s and global locks defined in its header and source files; this provides a useful starting point for most components.

To minimize the performance penalty from executing an instrumented operation, Redflag's logging module stores logged operations on a queue. Queueing events allows Redflag to return control immediately to instrumented code, deferring I/O until after logging. Deferring disk writes also makes it possible to log events that occur in contexts where it is not possible to perform potentially blocking I/O, such as when holding a spinlock.

To enqueue an operation, a logging function first pulls an empty log record from a pool of pre-allocated, empty records and then populates it with a sequence number, the current thread ID, and any information that the instrumented event passed to the logging function. The logger chooses the sequence number using a global, atomically-incremented counter so that sequence numbers preserve the ordering of logged operations among all threads and CPUs. Without hardware support, the logger cannot determine the exact ordering of memory operations executed on separate processors as scheduled by the memory controller, but it does guarantee that the final order of logged operations is a possible order. For example, if threads on two processors request a spinlock at the same time, the log never shows the losing processor acquiring the lock before the winning processor releases it.

When the user stops logging, a backend thread empties the queue and stores the records to disk. With 1GB of memory allocated for the queue, it is possible to log 20M events, which was enough to provide useful results for all our analyses.

It is also possible to configure background threads to periodically flush the queue to disk to take logs that are too large to stay in memory. Background threads compress their output with zlib, which is already available in the kernel, to reduce overhead from writing to disk.The logs are written to the disk by implementing a zero-copy mechanism in the kernel. The default way of logging in the kernel is to allocate a page in the kernel, call the kernel write function which copies the page to a new page and then writes that page asynchronously to the disk. To prevent this extra memory copy we implemented a zero-copy mechanism by allocating a page directly on the page cache and dirtying this page so that it is asynchronously written to the disk by the kernel write-back thread. This is like an `mmap` mechanism to prevent extra memory copying.

In this approach, the size of the queue can be configured depending upon the system. With fixed size queue it is important to ensure that the queue does not get full if we want to log events for a longer time. But with multiple producers there is always a possibility of producers overrunning the consumer threads and it is not easy to estimate the minimum queue size required to log events for a long time. We approach this problem in two ways. First we improve the performance of our logging threads and second we implement a sleep/wakeup mechanism between producers and consumers.

We improve performance by a set of in kernel optimizations like increasing their priority, binding 1 thread to each CPU and using a per thread log file so that each thread can write to a separate file. Binding a thread to each CPU makes sure that we have at least one consumer thread on each CPU. Otherwise a fast producer running on one of the CPUs in a multi-core system could overflow the queue.

By using per thread log file we prevent threads contention on single file. Each record is given a new sequence number which is incremented when the record is en-queued. A user-level post-processing step merges the output from the logging threads, sorting operations by sequence number.

The producers wake up the consumer threads whenever they put data onto the queue. Waking up the consumer threads is expensive as it involves disabling interrupts for a short time and the overhead of frequent context switch is also high. So we implemented a low-watermark level so that consumers are not woken up until there is enough data on the queue. The producers go to sleep before waking up the consumers. The consumer threads dequeue all the data off the queue until either the low watermark is reached or the consumer is de-scheduled by the scheduler or goes to sleep when the logs are written to the disk. In the case where the low watermark level is reached, the consumers wake up the producers and they themselves go to sleep. When this happens the producers can proceed forward. We have kept the low watermark and high watermark as 1/4 and 3/4 respectively. The difference between low and high watermark is large enough so that there is not frequent waking up of the consumers by the producers. This helps us to utilize the queue in an efficient manner. There is also very little possibility of ping pong at the high and low watermark level. This is because once either producers or consumers get a chance they queue and dequeue large amount of data.

The second reason of consumers going to sleep is if the scheduler de-schedules the consumer. In that case, the CPU cannot be given to the producer who is sleeping because that producer can only be woken up by the consumer. The scheduler might schedule other processes. If these processes are potential producers then they will also be put to sleep if the watermark level is high. So no producer can fill the queue until the watermark is reduced to low watermarks.

The third reason of consumers going to sleep is when they write to the disk. The consumers cache the data in per-consumer pool of memory and write to disk only when the pool gets full. Thus the consumers can go to sleep on an I/O to the disk. In that case the consumer will be woken up by I/O interrupt when the data is available. If the producers fill the queue in the high watermark level they will be put to sleep and woken up when the consumers are woken up on I/O completion and they have emptied the queue upto low watermark level. This design seems to work well for logging huge amount of data in the kernel.

It is also important that producer threads are not made to sleep when they are in a context in which sleeping is not allowed. For example a thread cannot go to sleep inside a spinning lock like spinlock, RCU, readers/writers lock and from an interrupt context. The logging system checks for the context before putting the producers to sleep.

Redflag also has the ability to provide a stack trace for every logged operation. To prevent repeated output of same stack traces, Redflag stores each stack trace in a data structure which is a combination of hash table and trie. Section 2.3 describes our stack trace logging design.

Developers who are very familiar with a code base may choose to keep stack traces off for faster logging. When reported violations involve functions that are called through only one or two code paths, stack traces are not always necessary for the debugging effort. If reports implicate functions that can be called from many different locations, the developer can take another log with stack traces to get a clearer picture of what is causing the violations.

2.2 Lockset Algorithm

Lockset is a well known algorithm for detecting *data races* that can result from variable accesses that are not correctly protected by locks. We based our Lockset implementation on Eraser [32].

A data race occurs when two accesses to the same variable, at least one of them a write, can execute together without synchronization to enforce their order. Not all data races are bugs; a data race is benign when the ordering of the conflicting accesses does not affect the program's correctness. Data races that are not benign, however, can cause any number of serious errors.

Lockset maintains a *candidate set* of locks for each variable in the system. The candidate lock set represents the locks that consistently protect the variable. A variable with an empty candidate Lockset is potentially involved in a race. Note that before its first access, a variable's candidate Lockset is the set of all possible locks.

This section describes an improved Lockset Algorithm for data race detection and performance analysis. The following sections describe our algorithmic improvements, along with various refinements and our usability improvements. Later on we also extend the algorithm to deal with multi-variable escape, supporting double-check locking and RCU synchronization.

2.2.1 Lockset algorithmic improvements

When a thread allocates a new object, it can assume that no other thread has access to that object unless it encounters a serious memory error. This assumption allows another means of synchronization: until the thread stores a new object's address to globally accessible memory, no concurrent accesses to it are possible. Most initialization routines in the kernel take advantage of this assumption to avoid the cost of locking when creating objects, but these accesses may appear to be data races to the Lockset algorithm.

The Eraser algorithm solves this problem by tracking which threads access variables to determine when each variable become shared by multiple threads [32]. We implement a simplified version of this idea: when a variable is accessed by more than one thread or accessed while holding a lock, it is considered shared. This approach is similar to approach based on using barrier synchronization as a point when the Lockset is refined [30].

Accesses to a variable before its first shared access are marked as thread local, and we ignore them for the purposes of the Lockset algorithm. Other versions of Lockset improve on this idea to handle variables that transfer ownership between threads or variables that are written in one thread while being read in many threads [8, 32, 36], but we did not find these improvements to be necessary for the code we analyzed.

```
1: spin_lock(inode->lock);
2: inode->i_bytes++;
3: spin_unlock(inode->lock);

4: inode->i_bytes++;

5: spin_lock(inode->lock);
6: inode->i_bytes--;
7: spin_unlock(inode->lock);
```

Figure 2.1: An example function body with inconsistent locking. Redflag will report two data races from a trace of this function's execution: one between lines 2 and 4 and one between lines 4 and 6.

The algorithm processes each event in the log in order, tracking the current Lockset for each thread as it is processed. Each lock-acquire event adds a lock to its thread's lockset. The corresponding release removes the lock. Figure 2.1 shows an example of a function body with inconsistent locking. At lines 2 and 4, the executing thread's lockset will contain the `inode->lock` lock. At line 4, the thread's lockset will be empty.

When we process an access to a variable, the candidate lockset is refined by intersecting it with the thread's current lockset. That is, at each access, we set the variable's candidate lockset to be the set of locks that were held for *every* access to the variable. When Lockset processes the access from line 2 in our example, it will set the candidate lockset for the `i_bytes` field to contain just `inode->lock`. The access at line 4 will result in an empty candidate lockset for `i_bytes`, and the candidate lockset will remain empty for all further accesses to this instance of `i_bytes`.

To simplify debugging, errors are reported as *pairs* of accesses that can potentially race. On reaching an access with an empty candidate lockset, our Lockset implementation revisits every previous access to the same variable. If no common locks protected both accesses, we report the pair as a data race. Because the candidate lockset is empty, there will always be at least one pair of accesses without a common lock. When the `i_bytes` candidate lockset becomes empty at line 4 of the example, our Lockset reports a potential race with the previous access at line 2. Because the candidate lockset remains empty at line 6, Lockset also reports a race between lines 4 and 6 but not between 2 and 6, which share a lock in common. All the previous implementations we have seen report this as a false positive Cilk [7], Java [8, 28, 29, 36], C++ [30], [38] and RaceTrack for.NET platform [42]. We have found that extra overhead in keeping track of the current lockset per memory access is not too high as the locks held at any time by a process is typically small. Redflag only produces one report for any pair of lines in the source code so that the developer does not have to examine multiple results for the same bug or benign race. Though the function in Figure 2.1 may execute many times, resulting in many instances of the same two data races, Lockset only produces one report for each of the two races. Each report contains every stack trace that led to the race for both lines of code and the list of locks that were protecting each access.

In addition to the base algorithm, there are several common refinements to improve Lockset's accuracy. These additions are necessary because some pairs of accesses do not share locks but still cannot occur concurrently for other reasons, which we discuss here. Three refinements that we implement track stack variables, memory reuse, and the *happened-before* relation.

2.2.2 Ignoring Stack Variables

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

The logging system determines if a variable is stack local from its address and the process stack size. The size of the per process stack in the kernel is fixed per architecture. If the variable's address lies between current process stack pointer and start of the stack then the variable is local. The start of the stack is `sp - stacksize`.

2.2.3 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 locations in memory. If a variable is allocated to memory where a freed variable used to reside, the new variable incorrectly takes on the previous variable's candidate lockset. Additionally, if the previous value was marked as shared, the new variable is prematurely marked as shared.

This is the biggest cause of false positives in Lockset results on the Linux Kernel. Besides allocation routines like `kmalloc` and variants the Linux Kernel also has caching primitives like `kmem_cache_alloc` which are used extensively. Each time these routines gets called, it always assigns the same address to the memory region being allocated. So each new call to memory allocation routines have to be treated as a reuse of memory hence we split the accesses before the allocation and all accesses after the allocation in two different generations. The Lockset is initialized for each generation. The new variable allocated is considered thread local until it is accessed by second thread. Eraser solves the memory reuse problem by reinitializing the candidate lockset and shared state for every memory location in a newly allocated region [32]. This improvement

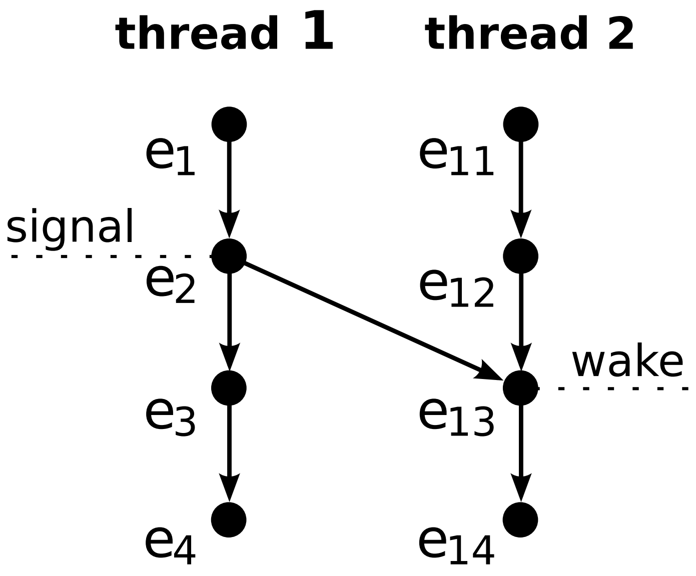

Figure 2.2: Example Hasse diagram for the happened-before relation on the execution of two threads synchronized by a condition variable. Because of the ordering enforced by the condition variable, there is an edge from the signaling event, e_2, to the waiting event, e_{13}.

reduces the false positives considerably as two variables from different generations cannot race with each other.

In some cases there would be a memory reuse of a variable even though there has been no explicit memory allocation to that variable. This problem is prominent because of memory allocation functions like `kmap/kunmap` and `get_free_pages`. A lot of places in the kernel memory is memory is mapped and unmapped from physical to virtual and vice versa using the `kmap/kunmap` functions. After obtaining the virtual address, the kernel would map a structure in some region of this virtual address and then read and write to this structure. But there has been no explicit allocation to this structure. The algorithm is made aware of this by keeping track of all memory mapped using `kmap` memory addresses and the map size. It then does a range query to find out if an access belongs to some mapped region and if so doing the generation update recursively to the structure mapped into part of this memory region. We have been able to eliminate a lot of false positives based on this idea in Btrfs file system.

2.2.4 Happened-Before Analysis

Threads can also synchronize accesses using fork and join operations and condition variables. The original Dinning and Schonberg Lockset implementation [12] was designed as an improvement to existing data-race detectors that used only Lamport's *happened-before relation* [21]. The happened-before relation is a partial order on the ordering of events in a program execution, so it can handle any form of explicit synchronization, including fork and join operations and condition variables.

The happened-before relation, HB, is a relation on the events in an execution trace. Events in the same thread are always ordered by HB in the order that they actually occurred: if events e_1 and e_2 are from the same thread and e_2 executed later in the thread than e_1, we can say that $(e_1, e_2) \in HB$.

Events that block on condition variables create edges in the happened-before graph between different threads. Figure 2.2 shows an example of an edge in HB representing a wait on a condition variable. The edge goes from the signaling event, e_2 in the example, to the waiting event, e_{13}. The happened-before relation assumes that all events in the signaling thread up to the signaling event (e_1 and e_2 in the example) must execute before any of the events in the waiting thread after it wakes up.

Our Lockset implementation uses the happened-before relation to determine when accesses are synchronized by a condition variable. If a happened-before ordering exists between two accesses, we assume that they cannot occur concurrently even if they share no common locks. Conversely, when the happened-before relation does not order the accesses, Lockset can report a potential race. Note that this assumption is not always safe; the happened-before relation can rule out feasible interleavings. Happened-before detectors usually also consider fork and join operations, but we did not find any accesses in the Linux kernel that depended on fork or join for synchronization. A fork operation spawns a new thread, and join waits for a thread to terminate.

2.2.5 Usability Improvements

The algorithm reports various race attributes like stack traces and lock usage statistics which improve usability of the system. If an object is accessed at different lines with inconsistent locks we output the lines and their corresponding stack traces. This helps to compare whether a particular type of object is accessed at different lines under different locks. The intuition is that objects which are accessed at different places with different locks are more likely to have bugs than those which are nowhere accessed with any locks. The algorithm also outputs any locks taken for the two accesses which can race so that it is easy to figure out which locks protect which objects. Another advantage of locking information is that objects which are inconsistently protected by spinning locks are more likely to have bugs as they are taken for short regions whereas a sleeping lock is taken for a coarser region and it might not be protecting the contended variable being accessed.

2.3 BLKTrace: Better Linux Kernel Tracer

2.3.1 Requirement for Stack Traces

Stack Traces are also required for analyzing results from our algorithms. Lockset emits violations in terms of line pairs. Two paths through system calls like read and write respectively can cause a race whereas two paths to the same two lines through different system calls like open and close may not cause the race. Thus, the context in which the race can occur is very important and stack traces help in understanding this context. Also, stack traces help understanding of the code in case of function pointers. Linux kernel has a number of layers across various subsystems. Running a test case might touch various subsystem layers. Stack traces help in understanding the interaction between these layers.

2.3.2 Kernel Support

Stack Traces are very useful in understanding bugs in the Linux Kernel. Whenever the kernel crashes or any type of error occurs, it emits the stack trace pointing out the set of events it was performing when the error occurred. Linux Kernel has a function called `dump_stack` which emits the stack trace. It can also be used for debugging purpose. It uses this to emit stack trace to the user space with a help of kernel thread named Ksyslogd. The mechanism makes use of architecture specific frame pointer support to emit a very precise stack trace.

Kernel prints stack traces when the errors occur and there is no easy mechanism of capturing these stack trace. The kernel logging mechanism is also very slow and redundant. Logging in the kernel uses a circular buffer. Ksyslogd kernel thread reads this buffer from time to time and copies it to the user space. This printing mechanism is designed to be called from either process context or interrupt context. So it disables interrupts in between and is thus very slow. Also, the data can get lost if the circular buffer is overrun. The overhead of copying to user space by syslogd also exacerbates the problem. Scalability is another problem with this mechanism. Capturing stack traces for read and write to every variable in a particular subsystem is very difficult.

2.3.3 Redflag Stack Trace Support

Redflag requires stack trace for each readwrite access which is not possible using the present mechanism. We built a novel mechanism for logging stack traces called BLKTrace. We designed the kernel `dump_stack` mechanism to log only unique stack traces. To log a stack trace BLKTrace goes through the process stack frame using the frame pointer. It uses the address of the top function in the stack trace as an index into a hash table. The hash table lookup is lock free using the Linux Kernel Read Copy Update (RCU) mechanism. However, the insertion into the hash table has to be protected by a spinlock to prevent concurrent writers from updating the hash table.

```
bar()
{
  foo();
}
foo()
{
  a = 1;
  b = 2;
  c = 3;
}
```

Figure 2.3: Function call sequence of two functions `foo() and bar()`

Indexing the hash table by the top function in the trace significantly reduces the number of stack traces. Consider two functions `foo()` and `bar()` as shown in the Figure 2.3. The `dump_stack` utility outputs three stack traces for variable accesses `a`,`b` and `c` whereas BKLTracer would only print one stack trace for the these accesses as shown in Figure 2.4

We maintain the list of stack traces as a trie. Each node of the trie is the starting address of a function. This data structure reduces the amount of memory usage for stack traces as all stack traces which reach the same function would share the trie node for that function entry. We have

```
dump_stack           BLKTracer
bar()                 bar()
foo() 0x00            foo() 0x0

bar()
foo() 0x04

bar()
foo() 0x08
```

Figure 2.4: Three stack traces for variables a,b,c using dump_stack and one stack trace using BLKTracer

seen that number of different paths reaching a particular point varies from 1 – 400, so keeping a trie like data structure reduces memory requirement.

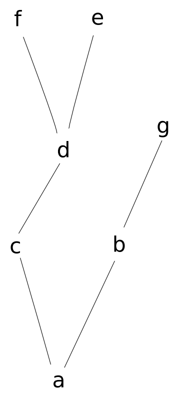

Figure 2.5: Stack Trace maintained as a trie

We also add to each path in the trie a node with a unique index for the stack trace. We append this index to each record in the trace obtained by the logger. We have found traversal of the trie can also slow down the system because the length of the stack trace can go up to 50. So along with the trie we also maintain a table sorted by the total length of the functions leading up to the first stack entry. Each entry of the table contains the length of the trace and the unique index we was appended to each path in the trie. Before we check the trie for a match we check the table using binary search to find a trace with the matching length. If the length is found then we return the length instead of searching the trie. We found in practice that this optimization reduces the stack trace overhead considerably. We maintain one table per function entry so using the stack length as a unique key for the trace works very well. This is because two traces leading to a same function entry, having the same length in terms of function size in bytes is highly unlikely.

We keep the stack traces in the memory only when the logging is taking place. Once the logging is disabled, all the traces are dumped to a file. The stack traces for reads and writes to all Btrfs structs and fields do not exceed more than 2 MB for a test like Racer. We also do not need to store the actual function name entries in the trie while logging. Each node of the trie contains only the address of the function. The function name corresponding to the symbol address can be easily retrieved through the symbol table lookup once the logging is disabled and before the stack traces are dumped to the file.

These stack traces are fed to the Lockset Algorithm which outputs the stack trace for each violation pair from top to bottom as shown in the Figure 2.6. The traces contain the address on which there is a lockset violation, the field of the structure, file line information, a unique stack trace index and any locking information. This figure shows that there was no lock held when access at file `fs/btrfs/inode.c` on line 251 and three locks held on the access to the same file with line 957. The type of lock is also printed with each report. This helps developers to prioritize looking at reports based on the type of locks. In our experience we have found that locks like spinlock and RCU which are held for a short duration are more likely to have serious errors. With this precise information, the developer can easily figure whether a report is a serious race or a benign race.

```
STRACE= ffff88009366a100.17  1 START=fs/btrfs/inode.c 5422 ST:251
ENTRY 251
                btrfs_getattr
                vfs_getattr
                vfs_fstatat
                vfs_stat
                sys_newstat
                system_call_fastpath
-------------------------------------------------------------------
STRACE= ffff88009366a100.17  1 START=fs/btrfs/inode.c 1378 ST:957
ffff88009366a178 fs/btrfs/extent_io.c:524 AC
ffff88009366a538 fs/btrfs/file.c:857 ML
ffff8800a11b2ac8 fs/btrfs/inode.c:1360 AC
ENTRY 957
                clear_state_bit
                btrfs_clear_bit_hook
                clear_extent_bit
                clear_extent_bits
                prepare_pages
                btrfs_file_write
                vfs_write
                sys_write
                system_call_fastpath
```

Figure 2.6: Lockset Output with Stack Traces

2.4 Other Algorithm Improvements

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. Though we observed these methods in the Linux kernel, they can be used in any concurrent system.

2.4.1 Multi-Stage Escape

As explained in Section 2.2, objects within their initialization phases are effectively protected against concurrent access because other threads do not have access to these newly created objects. However, an object's accessibility to other threads is not necessarily binary. An object can become available to a small 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 escapes during initialization. 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 lockset algorithm reported a race between accesses in the second-stage initialization and syscalls that operate on files, like `read()` and `write()`. These data races 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.

To avoid reporting these kinds of false inter-leavings, we introduce *Lexical Object Availability* (LOA) analysis. This analysis step produces a relation on field accesses for each targeted `struct`. Intuitively, this relation represents the order in which lines of code gain access to a `struct` as it is initialized. After constructing the LOA relations, we can use them to check the feasibility of inter-leavings reported by Lockset. LOA relations are based on object life cycles; the first step of the LOA algorithm is to divide the log file into sub-traces for each of the objects logged. A sub-trace contains all the accesses to a particular instance of a targeted `struct` in the same order as they appeared in the original log, from the first access following its allocation to the last access before deallocation.

The second step of the algorithm is to fill in the relation using the sub-traces. For each sub-trace, we add an edge between two statements in the LOA relation for that sub-trace's `struct` when we see evidence that one of the statements is allowed to occur after the other in another thread.

More formally, for a `struct` S and read/write statements a and b we include (a, b) in LOA_S iff there exists a sub-trace for an object of type 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 occurred in different threads, or there exists some e_c occurring between e_a and e_b in a different thread.

Our Lockset algorithm uses LOA to find out impossible interleavings. Our Lockset implementation 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.

2.4.2 Syscall interleavings

Engler and Ashcraft observed that dependencies on data prevent some kinds of syscalls from interleaving [13]. For example, a `write` operation on a file will not execute in parallel with a `open` operation because userspace programs have no way to call `write` before `open` finishes.

These kinds of dependencies are actually a kind of *multi-stage escape*, which we discuss in Section 2.4. The return from `open` is an escape for the file object, which then becomes available to other syscalls, such as `open`. For functions that are only ever called from one syscall, the LOA analysis we developed for multi-stage escape already rules out impossible interleavings between syscalls with this kind of dependency.

However, when a function is reused in several syscalls, the LOA relation 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.

We augmented the LOA relation to be a relation on the set of all pairs $(syscall, statement)$. As a result, LOA analysis treats a function that is executed by two different syscalls as two separate functions. Statements that do not execute in a syscall are instead identified by the name of the kernel thread they execute in. The augmented LOA relations can discover dependencies caused by both multi-stage escape during initialization and by dependencies among syscalls.

Though `open` and `write` cannot interleave operations on the same file, they can interleave operations on other shared objects, like the file system superblock. Because we keep a separate LOA relation for each `struct`, LOA analysis does not incorrectly rule out these interleavings because of dependencies related to other `struct`s.

2.4.3 RCU

Read-Copy Update (RCU) synchronization is a recent addition to the Linux kernel that allows very efficient read access to shared variables [25]. Even in the presence of contention, entering an RCU read-side critical section never blocks or spins, does not need to write to any shared memory locations, and does not require special atomic instructions that lock the memory bus.

A typical RCU-write first copies the protected data structure, modifies the local copy, and then replaces the pointer to the original copy with the updated copy. RCU synchronization does not protect against lost updates, so writers must use their own locking. A reader only needs to surround any read-side critical sections with `rcu_read_lock()` and `rcu_read_unlock()`. These functions ensure that the shared data structure does not get freed during the critical section.

We updated our Lockset implementation to also 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.

2.4.4 Double-Checked Locking

Double-check locking is an optimization technique frequently used in the Linux Kernel for performance reasons. If a read of a variable inside a spinlock is a conditional check, it is sometimes possible to do the check before taking a lock and if the check fails then try it with the lock. The check would fail if someone else modified the field just before the check took place. In that case we can do a check again under the lock to prevent concurrent writers from modifying the field at the same time. Failing in the first check of the lock should be no different from doing the check in the spinlock and the value changed just before taking a spinlock. Thus, it should be safe to change a lock for conditional variable to a double check lock. However, in highly concurrent systems its might reduce performance if there is high contention on the variable being protected. In this case the value might change very frequently so doing a double check would not produce much advantage. Doing a double checked lock should be no different of so that Taking locks is expensive and this optimization improves performance on the fast paths.

```
if ((inode->i_state & flags) == flags)
  return;
spin_lock(&inode_lock);
if ((inode->i_state & flags) != flags) {
  const int was_dirty = inode->i_state & I_DIRTY;

  inode->i_state |= flags;
}
```

Figure 2.7: Example of double-checked locking in VFS

Figure 2.7 shows an example of double-checked locking. We improved Lockset Algorithm to give hints for code paths suitable for double-checked locking. The algorithm finds out any access to global memory location inside a spinlock. If the access inside the spinlock is read and there is no write to any other memory location inside that spinlock, then possibly the read is just a conditional check. By code inspection we can find out if the read is indeed a conditional check: then the check can probably be moved before the spinlock and the lock is only taken if the check fails.

The Figure 2.8 shows a potential candidate for double-checked locking that we detected using this idea. The read of `reserved_extents` can be read without a spinlock first and if the value has changed then the spinlock does not need to be acquired.

2.5 AutoRedflag

This section describes a language `AutoRedflag` for automating the process of setting up Redflag. To set up Redflag we need to perform the following steps:

1) Import Redflag kernel changes to a vanilla kernel.

2) Create configuration files for the GCC for each of the plugins (i.e., field trace, lock trace, malloc-trace and syscall-trace). To instrument a particular subsystem we need to find its structs, fields, global and per struct locks and memory allocation routines.

```
spin_lock(&meta_sinfo->lock);
spin_lock(&BTRFS_I(inode)->accounting_lock);
if (BTRFS_I(inode)->reserved_extents <=
    BTRFS_I(inode)->outstanding_extents) {
  spin_unlock(&BTRFS_I(inode)->accounting_lock);
  spin_unlock(&meta_sinfo->lock);
  return 0;
}

can be changed to.....

spin_lock(&meta_sinfo->lock);
if (BTRFS_I(inode)->reserved_extents <=
    BTRFS_I(inode)->outstanding_extents) {
  spin_unlock(&meta_sinfo->lock);
  return 0;
}

spin_lock(&BTRFS_I(inode)->accounting_lock);
if (BTRFS_I(inode)->reserved_extents <=
    BTRFS_I(inode)->outstanding_extents) {
  spin_unlock(&BTRFS_I(inode)->accounting_lock);
  spin_unlock(&meta_sinfo->lock);
  return 0;
}
```

Figure 2.8: A candidate for double-checked locking

3) Compile the kernel image.
4) Install the kernel image.
5) Run a set of test cases on the installed kernel.
6) Run the algorithm on the logs collected and output the results to a file.

To ease the process of setting up Redflag we built a small language called AutoRedflag. AutoRedflag helps you specify various options for setting up the system, automatic patching of the kernel with Redflag changes, creating configuration files for plugins, creating kernel configuration file for enabling various options, compiling the kernel, running tests in parallel, copying log files and running algorithm over the logs.

Importing the Redflag changes is very easy. AutoRedflag patches the new kernel with our changes without any conflicts.

To configure the system, one needs to specify the kernel directory from which structs, locks and memory-allocation routines are extracted. It creates the configuration files which is then provided as an input to GCC plugins.

AutoRedflag has an option for enabling various options in the kernel configuration file. To enable an option just provide the option name and AutoRedflag creates a new configuration file with the specified option enabled. It can also compile the kernel, if the kernel compilation option is enabled.

Currently, installation of the kernel image on virtual machine has to be done manually but this process can be easily automated.

Once the image is installed, we need to boot the image. This operation is also done manually by the user by booting the installed image in the virtual machine.

Running tests in parallel is very simple using AutoRedflag. AutoRedflag is a client/server architecture. The AutoRedflag server takes input from AutoRedflag parser running on the host machine and a client running on VMware. The parser parses the AutoRedflag configuration file. The AutoRedflag configuration file has various options for running the system. The following options are currently supported

Command	Description
`ipclient`	IP address of the client running in virtual machine
`ipserver`	IP address of the server
`hook-inlay`	Configuration file for field trace plug-in
`sys-trace`	Configuration file for syscall trace plug-in
`malloc-trace`	Configuration file for malloc trace plug-in
`lock-trace`	Configuration file for lock trace plug-in
`log-dir`	Output directory for results from algorithm
`create_plugin_configs`	Enable plug-in config files
`config-kernel-path`	directory path for kernel config file
`save-config-dir`	directory for saving kernel configuration files
`kernel-path`	path of the kernel
`target-home-dir`	Directory on the Virtual machine in
which tests need to be run	
`binaries-dir`	Directory for specifying Redflag binaries
`patch`	Option to patch the new kernel
`source-patch-dir`	Kernel source directory option uses by patch command
`dest-patch-dir`	kernel destination directory option uses by patch command
`.config`	This option is used to create a new kernel configuration file.
`component`	Name of the component that needs to be enabled in the .config
`compile`	GCC compiler path
`compile-threads`	Number of make threads for kernel compile

Table 2.1: AutoRedflag configuration options.

Running the test is simple. The configuration file contains commands which are to be run as tests when the virtual machine is booted. When the virtual machine is up, a client is run on the virtual machine. This client establishes a communication with the AutoRedflag server. The server passes the commands to the client which are then run by the client as test cases. The configuration file also contains options for running these tests in parallel. The client can be configured to run on bootup using `init.d` scripts in Linux which runs when the kernel is booted up.

Once the tests finish, the log files are copied to a specified directory and AutoRedflag runs the algorithms on the log files. The results of these algorithms are saved to a set of output files for later analysis by the developer. Figure 2.9 shows a sample configuration file for the AutoRedflag.

```
homedir=/root/
TEST 1 START
name=racer
#compile=/home/abhinav/aristotle-32/aristotle/src/modular-gcc/install-svn/bin/gcc
#compile-threads=4

TASK 1 START
  CMD mkdir /mnt/wrapfs/
  CMD mkdir /mnt/ext4/
  CMD mount -t wrapfs -o lowerdir=/mnt/ext4/ none /mnt/wrapfs
CMD /root/racer/racer2.sh /mnt/wrapfs/ 60
TASK 1 END
EXECUTE TASK 1
TEST 1 END
```

Figure 2.9: An example configuration file for AutoRedflag

Each test can be configured to have multiple tasks either run in sequence or parallel. Commands to be run are specified by CMD option. The EXECUTE command executes these commands. To run multiple commands in parallel, multiple options are specified as arguments to the EXECUTE command. These commands are sent to a client running on VM machine where they are executed depending upon the options specified.

Chapter 3

Evaluation

To evaluate its accuracy and performance, we exercised Redflag on three kernel components: two file systems and one video driver. We took logs from each of these systems and analyzed those logs with our Lockset and block-based implementations. We present the results of that analysis here, along with performance benchmarks for our instrumentation and logging.

The two file systems we examine are Btrfs, a complex in-development on-disk file system, and Wrapfs, a pass-through stackable file system that serves as a stackable file system template, also in development. 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 logged each file system while running the Racer tool [35], which is designed to test a variety of file-system system calls concurrently to trigger rare schedules. Our analysis does not require a violating schedule to execute in order to detect it, but executing more schedules provides better information to our LOA analysis.

Nouveau, the video driver we examined, provides hardware 2D and 3D hardware acceleration for Nvidia video cards. We logged Nouveau data structures while playing a video and running several instances of `glxgears`, a simple 3D OpenGL example. We were not able to run more complicated 3D programs under Nouveau, which is still in early development.

3.1 Analysis Results

	Total	Bug	Benign	Stat	Untraced lock
Btrfs	8	0	8	0	0
Wrapfs	78	2	45	29	2
Nouveau	11	0	0	0	11

Table 3.1: Reported races from the Lockset algorithm. From left to right, the columns show: total reports, confirmed bugs, benign data races caused by `stat`, other benign data races, and false positives caused by untraced locks.

Lockset results Table 3.1 shows our analysis results for the Lockset algorithm. Our analysis of Wrapfs revealed two confirmed locking bugs. The first bug results from an unprotected access to a field in the file `struct`, which is one of the VFS data structures we included for our Wrapfs tests. A Lockset report (listed as "Bug" in Table 3.1) showed that two parallel calls to the `write` syscall can access the `pos` field simultaneously. Investigating the report, we found an article describing a bug resulting from the reported race: parallel writes to a file can sometimes write their data to the same location in a file, in violation of POSIX requirements for writes [10]. Because proposed fixes carried too high a performance cost, this bug is currently still present in the Linux kernel.

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 the Wrapfs developers and tested a simple fix to the locking in `wrapfs_setattr`.

Most of Lockset's reports are in fact benign races: data races that occur but that do not affect the correctness of the program. In particular, there are a number of benign races in the `stat` syscall. The `stat` syscall is responsible for copying information about a file from the file system's inode structure to the user process, but it does not lock the inode for the copy. The unprotected copy can race with several other file system operations, causing `stat` to return inconsistent results. An inconsistent `stat` result returns some fields from an inode before a concurrent syscall executed and some fields from after that syscall executed. This Linux community considers this behavior preferable to the performance cost that additional locking in `stat` would introduce [2]. We list these races in the "Stat" column of Table 3.1. The remaining benign races are in the "Benign" column.

All of the false positives in Nouveau resulted from variables that are protected by locks external to Nouveau. Because these locks do not belong to the `struct`s we targeted, they were not logged, making them invisible to our analysis. Untraced locks also caused two false positives in Wrapfs (the "Untraced Lock" column in Table 3.1). If reports from an untraced lock become overwhelming, the user can target the offending lock for instrumentation and produce a new log.

The "Untraced lock" false positive in the Nouveau driver is actually protected by the Big Kernel Lock (BKL), which we did not instrument. The BKL is a monolithic lock that protects data in many kernel systems. If kernel developers decide to replace the BKL in Nouveau, they will need to introduce a new lock to protect the two variables involved in this false positive.

3.2 Performance

Logging To measure the performance of our logging system, we tested logging overhead with FileBench using a workload of mixed reads and writes on a small data set. We kept the data set small enough to fit in RAM to ensure that the I/O cost of the workload did not dominate our benchmark. We benchmarked logging on a computer with a 2.4GHz quad-core Intel E5530 processor and 12GB of RAM. Instrumentation in our benchmark targeted the Btrfs file system running as part of the 2.6.33 release of the Linux kernel. Along with the instrumented kernel, the test system ran Ubuntu 9.10 with packages up-to-date as of April 27, 2010.

With all Btrfs data structures instrumented but logging turned off, the workload ran with 12% overhead. Turning logging on resulted in 35% overhead without stack trace logging. With stack traces, logging ran with 45 times overhead. The added overhead is almost entirely from the expense of reading the stack trace itself, including traversing the frame pointer and mapping return addresses to the locations of the functions they return to. The rest of logging overhead is caused by the cost of calling instrumentation functions, synchronizing the logging queue, and copying data into log records.

Analysis Though they run offline, we also measured performance results for each of our analyses. We measured the time each analysis took along with memory usage for our Btrfs log, which has 13.6 million events and 7,119 stack traces. Our analyses ran on a test machine with an identical hardware configuration to the computer we used to benchmark logging.

It took our Lockset implementation 15 minutes to analyze the Btrfs log, using 1.3GB of memory at its peak.

Chapter 4

Related Work

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

Lockset Our Lockset implementation is informed primarily by the 1997 Eraser tool [32], which is itself based on Dinning and Schonberg [12]. Eraser introduced memory reuse tracking and a technique for determining when variables become shared so that unprotected accesses during initialization do not cause false positives. Variations of Lockset exist for Cilk [7], Java [8, 28, 29, 36], C++ [30], and the .NET platform [42]. These tools all detect errors on-the-fly, and most focus on reducing the performance impact of computing and storing the lockset. Their overheads are typically better than Redflag's data-collection overhead, but they use optimizations that would not apply to other techniques, such as the block-based algorithm.

LiteRace uses sampling to track only a small percentage of accesses and synchronization events. One of its goals is to target cold code paths, which are more likely to hide potentially dangerous data races than frequently executed functions [24]. Users have no control over the sampling, however; they cannot choose to target debugging effort to specific program modules or data structures. LiteRace uses a purely happened-before–based detector, which is more likely to miss data races because of scheduling perturbations than Lockset-based approaches [12].

Static analysis Static analysis tools are very effective for finding data races in large systems, usually by employing a Lockset-style approach of finding variables that lack a consistent locking discipline [9, 13, 19, 31, 33, 41]. The RacerX tool, for example, found data races in both the Linux and FreeBSD kernels [13].

These tools cannot easily check for more complicated concurrency properties, such as atomicity. There is no static analysis equivalent to the block-based algorithms we used to check for the atomicity of system calls. Existing static analyzers designed for atomicity properties check for *stale values* and *method consistency*. The stale-value approach flags values that are saved within a critical section and then later reused outside that critical section, which can often violate atomicity [4]. The method-consistency approach [37] is based on the idea of *high-level data races* [1]; it characterizes methods by their *lock views*, the critical sections they execute, paired with the variables accessed in those critical sections. When methods' lock views do not *chain*, it can often mean inconsistent locking among a set of variables that are protected together. Both these ideas form a useful intuition about the kinds of atomicity properties programmers expect, but neither is equivalent to checking for atomicity. They can miss some kinds of atomicity violations, and they can also report interleavings that are not atomicity violations.

Model checking Model checking can verify concurrency properties by ensuring that invariants hold under all possible interleavings [6, 11, 18]. In particular, it is possible to show atomicity by proving that a function produces the same resulting state no matter how it is scheduled [3, 15, 34]. Such systems can detect any kind of concurrency error, but they do not scale to systems as large as those that we tested. The Calvin-R checker, which directly checks for atomicity using Lipton's reduction property [22], can verify a 1,200-line NFS implementation given a developer-written specification for every function it checks [16]. The largest file system we checked, Btrfs, has more than 54,000 lines of code.

The CHESS tool's systematic testing approach is similar in flavor to model checking [27]. Rather than exhaustively checking every possible interleaving,CHESS executes all the schedules that can result from a limited set of preemption points. The FiSC model checker was designed to operate on an entire production file system, and successfully found errors in several Linux file systems [40]. Its checking is single-threaded, however, and does not detect errors arising from parallel executions.

Runtime atomicity Though we focused on the Lockset and block-based algorithms for our analysis, there are several other runtime techniques for detecting different styles of concurrency problems. All of these techniques could be adapted to work with Redflag's logging output.

Like the block-based algorithm, analysis based on Lipton's reduction property [22] can also check operations for atomicity violations [14, 20, 38]. Although it checks for the same kinds of errors, Lipton's reduction is generally more efficient than the block-based algorithm, especially for very long traces [38].

Another potentially useful analysis checks for *high-level data races* [1]. As with method consistency, discussed earlier among the static analysis tools, high-level data races represent groups of variables that are protected together but accessed in an inconsistent way.

AVIO's analysis is similar to the block-based algorithm in that it reports pairs of instructions that are interleaved inconsistently [23]. Xu et al's analysis first infers what regions should be atomic and then checks that they follow a 2-phase locking protocol [39]. These approaches have the advantage that they do not need any programmer-supplied information about which code regions should be atomic. None of the runtime analysis tools discussed here operate at the kernel level. Currently, Linux kernel developers who want to dynamically check their code for concurrency errors are limited to checks for API misuse and potential deadlock [26].

Chapter 5

Conclusions

Redflag attempts to manage the complexity of concurrent systems software, targeting specific components and identifying possible interleavings in those systems that can lead to difficult-to-debug concurrency errors. We have shown that Redflag's infrastructure is versatile: it produces highly detailed logs of system execution on which it can run a variety of analyses. As Redflag is modular, logs can target specific system data, making them more valuable to developers who want to focus their efforts on individual system components.

We have shown that, although the cost of thorough system logging can be high, Redflag's performance is sufficient to capture traces that exercise many system calls and execution paths. The runtime analyses that Redflag uses are designed to find problems even if they exist in schedules that may occur only rarely, mitigating the problem of schedule perturbations resulting from logging overhead. We have also presented a number of techniques we used to improve the accuracy of our analysis. Besides finding data races our Lockset implementation also hints at places for performance optimizations related to double check locking. Redflag also logs RCU synchronization, so that its Lockset implementation can identify invalid synchronization of RCU-protected variables. Finally, we developed Lexical Object Availability (LOA) analysis to remove false positives caused by complicated initialization code that uses multi-stage escape.

Chapter 6

Future work

Memory barriers can cause subtle synchronization errors which are hard to debug. We plan to improve upon our preliminary algorithm for detecting memory barrier problems and make it more generic and concrete. We also plan to improve the algorithm to handle any type of re-ordering and not just store-load reordering.

We also believe that in addition to finding errors, Redflag could be applied to the problem of improving the performance of concurrent code. By examining locking and access patterns in execution logs, Redflag can also be used to predict data structures which would benefit from RCU based locking. Furthermore, if Redflag logged information about lock contention, it could find critical sections that are too coarse-grained, leading to contention, or too fine-grained, requiring unnecessary locking operations.

We can also extend our analysis to real-time kernel. Real-time kernel have variants of conventional locking mechanism like spinlocks and RCUs which are more suited to real-time systems performance.

Capturing stack traces is the most expensive part of our logging system. We plan to explore ways to read less stack trace data in order to improve performance. If we instrument a small number of function boundaries, we can capture only a partial stack trace or avoid reading the stack trace all together when we know we have not exited the current function.

Finally, we can improve our logging system to get our race detection instrumentation framework into the mainline kernel. The Linux kernel has a dynamic deadlock detection tool, Lockdep, but it does not have any tool for dynamic race detection. This is because race detection requires more sophisticated analysis. We think that our targeted approach with offline analysis can be brought into the mainline and is a more practical approach for race detection in the kernel.

Bibliography

[1] C. Artho, K. Havelund, and A. Biere. High-level data races. In *VVEIS'03, The First International Workshop on Verification and Validation of Enterprise Information Systems*, Angers, France, April 2003.

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

[3] D. L. Bruening. Systematic testing of mulithreaded Java programs. Master's thesis, Massachusetts Institute of Technology, 1999.

[4] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs: Research articles. *Concurr. Comput.: Pract. Exper.*, 16(12):1161–1172, 2004.

[5] 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.

[6] T. Cattel. Modeling and verification of sC++ applications. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 232–248, London, UK, 1998. Springer-Verlag.

[7] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 298–309, New York, NY, USA, 1998. ACM.

[8] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269. ACM Press, 2002.

[9] J.-D. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical Report RC22146, IBM Research Division, Thomas J. Watson Research Center, 2001.

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

[11] C. Demartini, R. Iosif, and R. Sisto. Modeling and validation of Java multithreading applications using SPIN. In *Proceedings of the 4th SPIN Workshop*, Paris, France, 1998.

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

[13] 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.

[14] 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.

[15] Cormac Flanagan. Verifying commit-atomicity using model-checking. In *In Proc. 11th Intl. SPIN Workshop on Model Checking of Software, volume 2989 of LNCS*, pages 252–266. Springer-Verlag, 2004.

[16] S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. In *Workshop on Formal Techniques for Java-like Programs*, Darmstadt, Germany, 2003.

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

[18] K. Havelund and T Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[19] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV'07: Proceedings of the 19th international conference on Computer aided verification*, pages 226–239, Berlin, Heidelberg, 2007. Springer-Verlag.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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`.

[26] 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`.

[27] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[28] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[29] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38(10):167–178, 2003.

[30] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190, New York, NY, USA, 2003. ACM.

[31] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Context-sensitive correlation analysis for race detection. *SIGPLAN Not.*, 41(6):320–331, 2006.

[32] 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.

[33] N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of the USENIX Winter Technical Conference*, pages 97–106, San Diego, CA, January 1993.

[34] S. D. Stoller. Model-checking multi-threaded distributed Java programs. In *International Journal on Software Tools for Technology Transfer*, pages 224–244. Springer, 2000.

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

[36] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 70–82, New York, NY, USA, 2001. ACM.

[37] C. von Praun and T. R. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(6):103–122, 2004. Special issue: ECOOP 2003 workshop on FTfJP.

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

[39] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Not.*, 40(6):1–14, 2005.

[40] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.

[41] M. Young and R. M. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Trans. Softw. Eng.*, 14(10):1499–1511, 1988.

[42] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2005. ACM.