LANDSLIDE: A SIMICS* EXTENSION FOR DYNAMIC TESTING OF KERNEL CONCURRENCY ERRORS

Contributors

Ben Blum
Department of Computer Science, Carnegie Mellon University

David A. Eckhardt
Department of Computer Science, Carnegie Mellon University

Garth Gibson
Department of Computer Science, Carnegie Mellon University

Landslide is a Simics module designed for finding concurrency bugs in operating system kernels, with a focus on Pebbles. Pebbles is a UNIX-like kernel specification used in course 15-410, the undergraduate operating systems class at Carnegie Mellon University, in which students implement such a kernel in six weeks from the ground up. Landslide's mechanism, called systematic testing, involves deterministically executing every possible interleaving of thread transitions in a given test case and identifying which ones expose bugs. In this article we explain the testing environment (the course, 15-410, and the kernel, Pebbles) and the testing technique; describe how Landslide takes advantage of certain features that Simics provides that other testing environments (such as virtualization) do not; outline Landslide's design, implementation, and user interface; present some results from a preliminary evaluation of Landslide, and discuss potential directions for future work.

Introduction

Race conditions are notoriously difficult to debug. Because of their nondeterministic nature, they frequently do not manifest at all during testing, and when they do manifest, it can be difficult to reproduce them reliably enough to collect enough information to help debugging.

Many techniques exist for dynamic testing of concurrent systems for race conditions. Systematic exploration, the strategy we focus on in this work, involves making educated guesses as to what points during execution a preemption would be most likely to expose a bug, enumerating the different possibilities for interleaving threads around these points, and forcing the system to execute all such interleavings to check if any of them results in incorrect behavior.[1] Systematic exploration provides a better alternative to conventional long-running stress tests, because it is less likely to overlook buggy execution patterns, and it enables a testing framework to report more thorough debugging information. Compared to other dynamic analyses, such as data race detection[2], systematic exploration is able to find a wider range of types of concurrency errors because of its ability to manipulate the execution of the system under test.

"Systematic exploration is able to find a wider range of types of concurrency errors"

In this article, we present Landslide, a Simics module that provides a framework for performing systematic testing on kernel-level code.[3] Landslide is designed with a focus on the testing environment used by students in course 15-410, the undergraduate operating systems class at Carnegie Mellon University (CMU). In 15-410, students implement a fully preemptible, UNIX-like kernel from the ground up over the course of a six-week project.[4] They

use the Simics simulator as their primary testing and development platform, although they must rely on conventional stress-testing techniques to find and track down concurrency bugs in their code. Landslide is an effort to improve this situation by making the more sophisticated technique of systematic testing accessible to developers of kernel code.

This article is structured as follows. In the section "15-410 and Pebbles," we discuss the course design, projects, and learning objectives for 15-410, with a detailed overview of the requirements of the kernel project. In the section "Systematic Testing," we introduce the technique of systematic testing, explaining its requirements, advantages, and challenges. In the section "Design and Implementation," we discuss the design of Landslide's architecture, describing the overall sequence of events involved in a systematic testing run, and the various components of Landslide and how they fit together. In the section "Use of Simics Features," we focus specifically on how Landslide and Simics fit together, highlighting the unique features that Simics offers that make Landslide's job possible. In the "User Interface" section, we present Landslide's user interface, describing the instrumentation process users must complete in order to use Landslide, and the interface Landslide offers for fine-tuning the search parameters and reasoning about uncovered bugs. In "Results" we discuss a user study we conducted with volunteer students from 15-410, in which Landslide was able to help the students uncover and fix previously-unknown race conditions in their own kernels, and finally, in "Future Work," we conclude with a discussion of the most promising future work directions for this research.

15-410 and Pebbles

15-410, the Operating Systems Design and Implementation course at CMU, is a semester-long project course comprising five projects. The projects are a stack tracer, kernel device drivers (for timer, keyboard, and console), a 1:1 user-space threading library to run on a Pebbles kernel, the Pebbles kernel itself, and a small extension to the Pebbles kernel. Simics is used as the main development and debugging environment for the latter four projects.

The course has many learning objectives, ranging from acquiring detailed factual knowledge about hardware features through practicing advanced cognitive processes such as open-ended design. Students study high-level concepts such as protection (least privilege, access control lists vs. capabilities), file-system internals, and log-based storage. We place emphasis on acquiring information from primary sources, including both manufacturer-provided hardware documentation and a non-textbook technical-literature reading assignment. Students begin with a blank slate rather than a kernel-source template or an existing operating system, so they must synthesize design requirements from multiple sources and must choose their own module boundaries and inter-module conventions. Due to the foundational nature of kernel code, the assignment design and grading encourage students to think about corner cases, including resource exhaustion, instead of being satisfied by "the right basic idea" implementations that handle only auspicious situations. Finally, most relevant to

"The assignment design and grading encourage students to think about corner cases, instead of being satisfied by 'the right basic idea' implementations."

this work, students gain substantial experience in analyzing and writing lock-based multi-threaded code and thread-synchronization objects. They practice detecting and documenting deadlock and race conditions, including both thread/thread concurrency and thread/interrupt concurrency.

Project Overview

In the course of a semester, students work on five programming assignments; the first two are individual, and the remaining three, including the kernel project itself, are the products of two-person teams. Here we are primarily concerned with the kernel project, though we will also briefly describe the others.

Introductory Projects

The first project is a stack crawler: when invoked by a client program, it displays the program's stack symbolically, rendering saved program-counter values as function names and printing function parameters in accordance with their types. This project enables students to review key process-model and language-runtime concepts from the prerequisite course[5]; it introduces students to our expectations about design, analysis, and making choices; finally, because C pointers are unsafe, it requires students to consider robustness.

The second project is a simple game, such as Hangman, which runs without an underlying operating system. The project requires students to implement a device driver library consisting of console output, keyboard input, and a hardware timer handler. This project and the remaining ones are written in C with some x86-32 assembly code, which is then compiled and linked into an ELF executable, stored into a 1.44-megabyte 3.5-inch floppy-disk image, and booted via GRUB. If the image is copied to a real floppy or embedded into an "El Torito" bootable compact disc image, it can be booted on standard PC hardware; however, students most often use Simics, to take advantage of its debugging facilities.

The third project is a 1:1 thread library for user-space programs, essentially a stripped-down version of POSIX Pthreads. Students begin by designing mutexes using any x86-32 atomic instructions they choose. They then write other thread-synchronization primitives (condition variables, semaphores, and reader/writer locks), infrastructure components (stack allocation/recycling and a thread registry), and low-level code to launch and shut down threads. Student library code is linked with small test programs provided by the course staff. The test programs run on a reference kernel written by the course staff and provided in binary form, the behavior of which is specified in a twelve-page document. In addition to providing a reliable execution substrate, the reference kernel schedules the execution of user-space threads created by student code according to a variety of interleaving policies.

"Two-student teams produce a kernel which implements the same specification as the reference kernel they previously relied on."

The Pebbles Kernel Project

For the fourth project, two-student teams produce a kernel which implements the same specification as the reference kernel they previously relied on. They design and implement some approach to synchronizing and blocking threads while they are in kernel space, a simple round-robin scheduler, basic virtual memory, a program loader, code to handle various x86 exceptions, and code

for setting up and tearing down threads and processes (they reuse their game-project device drivers). We briefly describe each of the 25 system calls in the Pebbles specification in Table 1.

Name	System Call Description
	Lifecycle Management
`fork`	Duplicates the invoking task, including all memory regions.
`thread_fork`	Creates a new thread in the current task.
`exec`	Replaces the program currently running in the invoking task with a new one.
`set_status`	Records the exit status of the current task.
`vanish`	Terminates execution of the calling thread.
`wait`	Blocks execution until another task terminates, and collects its exit status.
`task_vanish*`	Causes all threads of a task to vanish.
	Thread management
`gettid`	Returns the ID of the invoking thread.
`yield`	Defers execution to a specified thread.
`deschedule`	Blocks execution of the invoking thread.
`make_runnable`	Wakes up another descheduled thread.
`get_ticks`	Gets the number of timer ticks since bootup.
`sleep`	Blocks a thread for a given number of ticks.
`swexn`	Registers a user-space function as a software execption handler.
	Memory Management
`new_pages`	Allocates a specified region of memory.
`remove_pages`	Deallocates same.
	Console I/O
`getchar*`	Reads one character from keyboard input.
`readline`	Reads the next line from keyboard input.
`print`	Prints a given memory buffer to the console.
`set_term_color`	Sets the color for future console output.
`set_cursor_pos`	Sets the console cursor location.
`get_cursor_pos`	Retrieves the console cursor location
	Miscellaneous
`readfile`	Loads a given buffer with the names of files stored in the RAM disk "file system."
`halt`	Ceases execution of the operating system.
`misbehave*`	Selects among several thread-scheduling policies.

Table 1: The 25 system calls described in the Pebbles specification. Students are not required to implement the three system calls marked with an asterisk (*).
(Source: Pebbles kernel specification, 2013.[4])

"For most students in the class, this is the largest and most complicated software artifact they have produced."

For most students in the class, this is the largest and most complicated software artifact they have produced. Because the test suite and the grading criteria emphasize robustness and preemptibility of kernel code, there are many cross-cutting concerns. As students are responsible for ensuring the runtime invariants underlying all compiler-generated code in the system (kernel and user-space), they gain experience with debugging at both the algorithm level and the register/bit-field level.

Widely regarded as the most difficult concurrency problem in the project is that of coordinating a parent and a child task that "simultaneously" exit: when a task completes, live children and exited zombies must be handed off to the task's parent or to the system's "init" process, at a time when the task's parent may itself be exiting; meanwhile, threads in tasks that receive new children may need to be awakened from the wait() system call. Due to design constraints imposed by other parts of the kernel specification, solutions that are not carefully designed are prone to data races or deadlocks.

Students who complete the kernel project on time then work on a kernel-extension project, with varying content depending on the semester. Past projects have included writing a sound card driver, a file system, hibernation (suspend to disk), kernel profiling, and an in-kernel debugger. Two recent, more aggressive, projects have been adding paravirtualization so that their kernels can host guest kernels and adding multiprocessor support to their single-processor kernels.

"Unlike some emulators, which focus on fast execution of correct code, Simics provides very faithful support not only for correct code but also for kernels that accidentally abuse hardware."

Use of Simics

Simics serves as the main execution and debugging platform in 15-410. Unlike some emulators, which focus on fast execution of correct code, Simics provides very faithful bit-level support not only for code that behaves correctly but also for kernels that accidentally "abuse" hardware. Unlike hardware virtualization environments, Simics contains substantial debugger support: single-stepping, printing of source-level symbolic expressions, stack tracing, display of TLB entries, and even summaries of x86 hardware-defined descriptor tables. All of these features make Simics a helpful platform for students to test their code. A major advantage of using Simics over the QEMU emulator in particular is that QEMU issues timer interrupts only at basic-block boundaries, which would dramatically undermine our goal of teaching students that threads can interleave with each other at any time.[6]

Systematic Testing

The underlying idea of systematic testing is to view the set of all possible execution sequences, which can change due to concurrency nondeterminism, as an *execution tree*. The root of this tree denotes the start of the test case, each branch represents one execution sequence, and nodes in the tree are *decision points*: time points during the execution where Landslide should attempt to force a different thread to run, thereby making progress through the state space.

Example

Consider the example code in Code 1, which demonstrates how the *thread_fork()* system call might be implemented. If a timer interrupt occurs at line 4, the child thread can run, exit, and free its state, causing the access on line 5 to be a use-after-free. Here, the necessary decision point for finding the bug is at line 4. Landslide will know that there should be a decision point here because it automatically interprets new threads becoming runnable as important concurrency events. Other decision points may also exist, for example, during the construction of the new *thread_t* struct, or during the new thread's execution. Together, the set of decision points defines an *execution tree* that contains this bug, depicted in Figure 1.

```
1  int thread_fork() {
2          thread_t *child = construct_new_thread();
3          add_to_runqueue(child);
4          // note: at this point child may run and exit
5          return child->tid;
6  }
```

Code 1. Example implementation of the thread_fork() system call. This example contains a race condition, described in the comment on line 4. Source: Landslide: Systematic dynamic race detection in kernel space, 2011.[3]

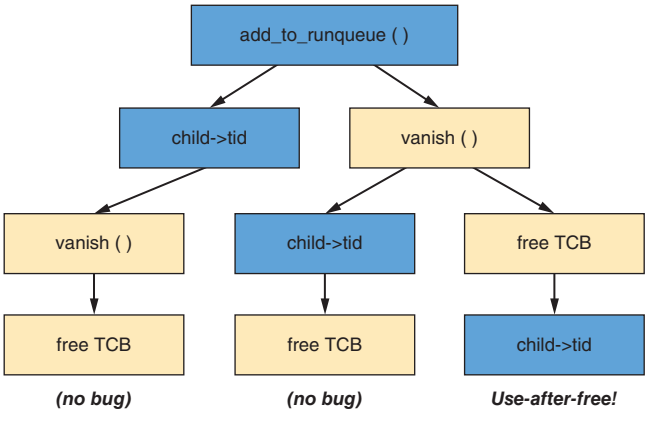

Figure 1: The set of possible execution sequences can be viewed as a tree of thread interleavings, in which a concurrency bug is only exposed in some branches. This particular tree is derived from the example code in Code 1. (Source: Landslide: Systematic dynamic race detection in kernel space, 2011.[3])

Challenges

In any systematic testing tool, there is an inherent tradeoff when defining the set of decision points: searching with few decision points results in coarser-grained interleavings, faster test completion, but less likelihood of finding unexpected bugs; whereas searching with more decision points results in the opposite. Accordingly, Landslide provides an interface for adjusting the set of

"There is an inherent tradeoff when defining the set of decision points."

considered decision points, which we discuss further in the section, "Use of Simics Features."

"Combining systematic testing with a kernel-space execution environment presents some additional challenges."

Combining the technique of systematic testing with a kernel-space execution environment presents some additional challenges. First, a testing tool must control all sources of nondeterministic input to the system, and account for all the scheduling options by each such source of input at each decision point. In the Pebbles environment, the only sources of nondeterminism are timer interrupts and keyboard input. With Landslide, we focus exclusively on timer interrupts, as they can be used to directly control the kernel's context switching.

A second challenge of systematic testing in kernel-space is that of the scheduler. Because kernels contain their own concurrency implementation, it can be difficult to find bugs in the scheduler itself while also being able to use assumptions about the scheduler's behavior to optimize our search for bugs in other parts of the kernel.

A third challenge is the issue of multiprocessor kernels: when multiple CPUs can be running different threads simultaneously, additional nondeterminism can arise from the order in which their instructions are executed. Some race conditions may even require multiple active CPUs in order to manifest. However, as 15-410 does not require student kernels to be capable of SMP execution, Landslide assumes kernels will only ever use one processor. Lifting this limitation is left to future research.

Design and Implementation

This section describes the important components of Landslide's architecture. Conceptually, Landslide is designed as follows. Students annotate their code so that Landslide knows which kernel thread is currently running. After one kernel thread has run for some time, Landslide triggers artificial clock interrupts to force the scheduler to run a different thread. When a test program finishes execution according to one pattern of thread switches, Landslide rewinds the kernel's state and resumes the test according to a different thread interleaving. After each instruction, Landslide applies several bug-detection predicates to the kernel's state to detect illegal heap accesses, deadlock, infinite loops, and panics. In theory, by forcing a thread switch after *every* non-scheduler instruction, Landslide could apply its bug-detection predicates to every reachable execution state. Because this would require a prohibitively large amount of time to complete, in practice Landslide uses a variety of techniques to thread-switch less often and to avoid repeating bug-equivalent execution paths.

In order to achieve this exploration of the state space, Landslide comprises several components, which are depicted visually in Figure 2 and described in the following sections.

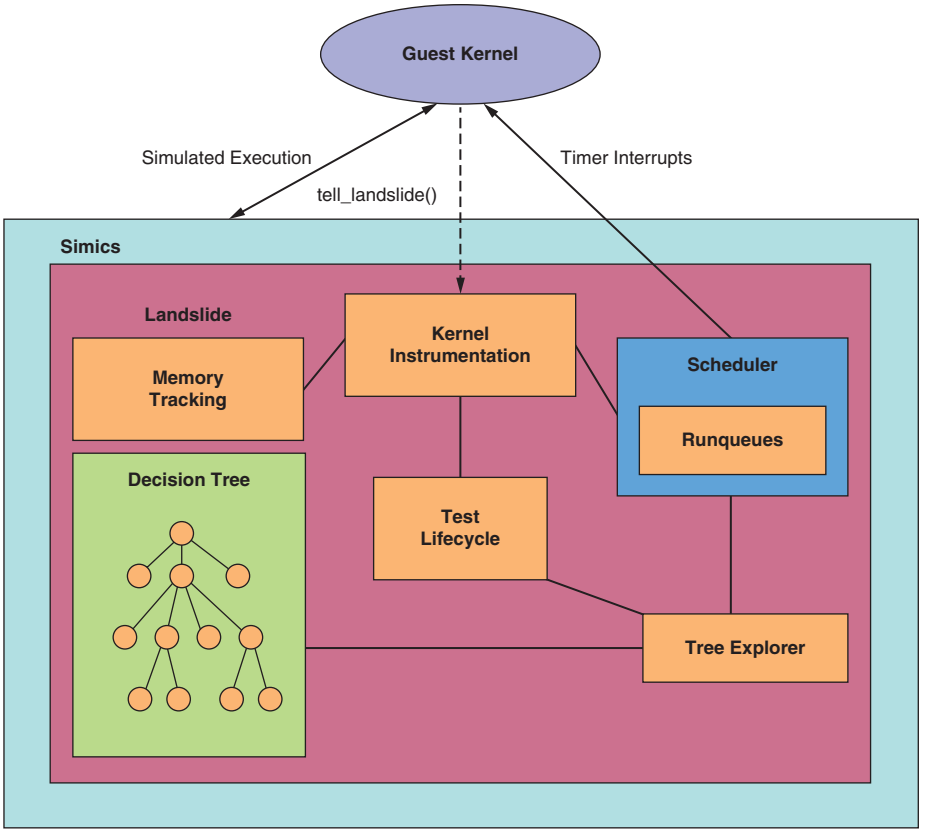

Figure 2: Visual representation of landslide's architecture and its interface with the kernel under test.
(Source: Landslide: Systematic dynamic race detection in kernel space, 2011.[3])

Thread Scheduler

The Landslide scheduler is responsible for keeping track of which threads exist in the guest kernel: which are runnable at any given time, and when they are created and destroyed. It maintains a "mirror image" of the guest kernel's scheduler state in the form of three queues, a pointer to the currently-running thread, and a pointer to the previously-running thread. The queues are the *runqueue*, containing the runnable threads, the *sleep queue*, containing threads which become runnable after a certain number of timer ticks, and the *deschedule queue*, which might not correspond to a data structure in the guest kernel, but contains all other threads that exist on the system that are not runnable for whatever reason.

Though we define timer interrupts as the only source of nondeterminism in our environment, it is more useful to view the concurrent behavior with a higher-level abstraction, in terms of the set of runnable threads and the ability to preempt the currently running thread with any different runnable one. Hence, the scheduler also contains the mechanism for translating the tree explorer's high-level decisions about which thread should run next into a lower-level sequence of timer interrupts (which trigger context switches). Note that multiple interrupts

may sometimes be necessary to force the desired thread to run; for example, if the kernel scheduler uses a round-robin policy and has a runqueue of thread IDs 1, 2, and 3 (with thread ID 1 currently running), if the Landslide scheduler desires to run thread 3, it will take 2 interrupts before thread 3 begins running.

Memory Access Tracking

Landslide maintains a mirror image of the guest kernel's dynamic allocation heap, so it can know at any point which memory ranges are allocated and which ranges used to be allocated but now are freed. This set is updated each time the guest kernel calls *malloc()* or *free()*. This heap tracking provides the ability to check for dynamic allocation errors (such as use-after-free and double-free bugs), in a similar fashion to the *Valgrind* debugging tool.

Landslide also maintains a set of shared memory accesses made since the last decision point, for use with the Partial Order Reduction state space technique (which we describe in the next section). This set of accesses allows Landslide to determine when certain actions of different threads may conflict with, or are independent from, each other. Landslide ignores shared memory accesses from the kernel's dynamic allocator itself, and it also ignores shared memory accesses from the components of the kernel's scheduler that run every transition.

Execution Tree Explorer

The execution tree explorer maintains a representation of the current branch of the decision tree. It is responsible for checkpointing the state of both Landslide and the guest kernel at each decision point, deciding at the end of the test which branch of the tree to execute next (that is, selecting which decision point should have been decided differently), and backtracking to appropriate points in the test's execution.

The explorer also identifies points during execution that should count as decision points. The selection is mainly controlled by the user, during the annotation and configuration process. However, the explorer also automatically identifies *voluntary reschedules*—points at which the kernel explicitly invokes a context switch of its own accord (for example, in *yield()*)—which comprise the "minimal necessary set" of decision points.

"DPOR analyzes the memory accesses to identify a set of candidate branches to explore next."

During the backtracking stage, the explorer applies a state-space reduction technique called *Dynamic Partial Order Reduction* (DPOR). Briefly, DPOR analyzes the memory accesses in a just-finished execution to identify a set of candidate branches to explore next. These branches represent reorderings of state transitions that conflicted with each other, with reorderings of independent transitions pruned out. For example, Figure 3 depicts a subset of a possible execution tree in which the highlighted transitions of threads 1 and 2 are independent from each other (that is, if they were reordered, the resulting kernel state would be identical.)

Bug Detection Techniques

During the test case's execution along each thread interleaving, Landslide applies several bug-detection predicates to the kernel's state, some accurate and some heuristic-based.

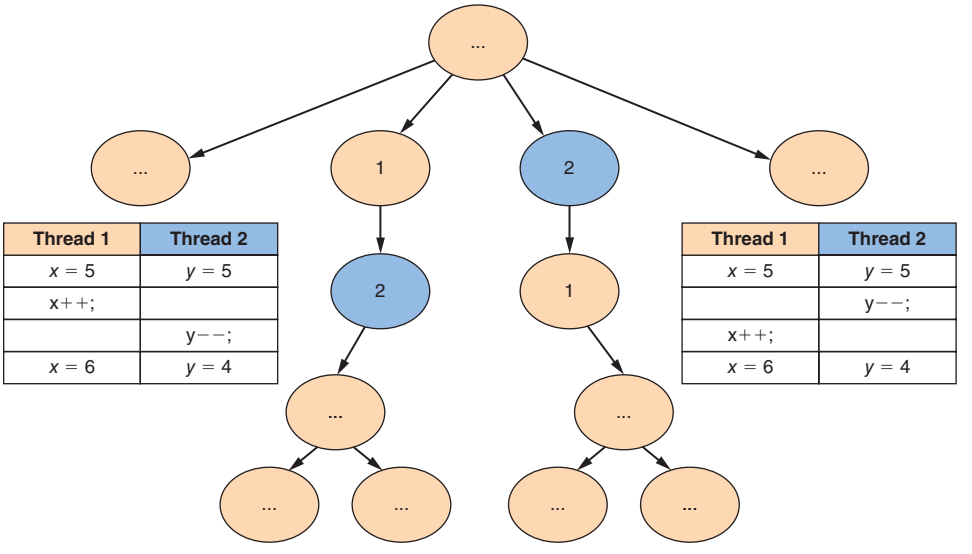

Figure 3: An example part of an execution tree that could be pruned using DPOR. The highlighted transitions of threads 1 and 2 are independent, meaning that to achieve full coverage, Landslide needs to explore only one of the two subtrees.
(Source: Landslide: Systematic dynamic race detection in kernel space, 2011.[3])

Landslide's "definite" bug-detection techniques include identifying kernel panics, use-after-free bugs (making use of the heap access tracking), and deadlocks (making use of mutex and scheduler instrumentation).

Additionally, Landslide can heuristically detect infinite loops by comparing the current execution of the test case against previous executions under different thread interleavings. If the current execution has lasted a certain proportion longer than the average of all previous executions, as visualized in Figure 4, Landslide assumes the deviation represents a nondeterministic infinite loop.

"Landslide can heuristically detect infinite loops by comparing the current execution of the test case against previous executions."

Use of Simics Features

This section discusses how Landslide and Simics fit together, and highlights some Simics features that Landslide makes heavy use of to enable systematic testing.

Landslide is implemented as a "trace" module, which means that Simics calls into it once per instruction and once per memory access, supplying information about the instruction or access about to be performed. Landslide uses this information to update its internal state machine to track the kernel's progress, by reading the values at memory locations, comparing the current instruction against certain known execution points in the kernel, and so on.

Landslide's control over the system consists of two parts. Together, these parts enable it to steer the kernel through the different branches of the execution tree, testing for bugs in each branch until the tree is exhausted.

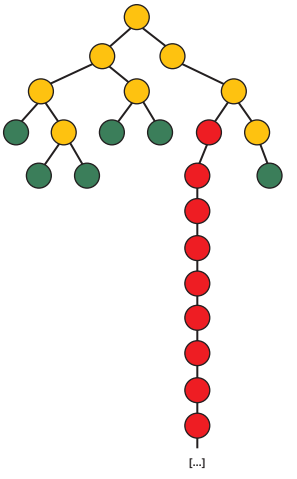

Figure 4: An example decision tree containing a nondeterministic infinite loop. If Landslide explores the highlighted branch after testing sufficiently many of the terminating branches, it assumes the kernel is stuck in an infinite loop and will report a bug. (Source: Landslide: Systematic dynamic race detection in kernel space, 2011.[3])

"If that thread is not the desired one, Landslide repeats the process, injecting more timer interrupts until the desired thread begins running."

The first part is causing a timer interrupt to occur at a given point during the kernel's execution. Landslide achieves this by manipulating the CPU's pending interrupt vector. When Landslide wishes to cause a particular thread to preempt another thread at a given decision point, it injects a timer interrupt before the pending instruction. In response, the kernel triggers a context-switch to the next thread on its scheduler run-queue. If that thread is not the desired one, Landslide repeats the process, injecting more timer interrupts until the desired thread begins running.

The second part of Landslide's control is backtracking. At the end of each branch of the decision tree, if Landslide wishes to explore a different interleaving at a particular decision point, it must reset the system state to the past state at that point. Fortunately, Simics provides a facility for reverse-execution in the form of the *set-bookmark BOOKMARK-NAME* and *skip-to BOOKMARK-NAME* commands. At each decision point during execution, Landslide uses *set-bookmark* to ask Simics to set a bookmark. Then, when the current execution of the test case completes, Landslide uses *skip-to* to reverse-execute to the bookmark associated with the desired decision point, at which point exploration resumes. Because Landslide places itself outside the scope of Simics' reverse execution system, although the entire simulated machine state is reset to the earlier point, Landslide's memory of the entire state space tree is persistent.

User Interface

Instrumenting and testing a kernel with Landslide involves three stages of effort. These are required annotations, configuring decision points for a more efficient search, and interpreting the resulting traces Landslide emits when it finds a bug. This section gives a brief overview of each.

Required Annotations

Users annotate their kernels to inform Landslide of certain important concurrency events during execution. We provide a set of annotation functions, named with the prefix *tell_landslide*, for this purpose. The annotations denote when a thread runs *fork()*, *sleep()*, or *vanish()*, when a thread is added to or removed from the run-queue, and when a thread becomes blocked on a mutex. The annotation is placed just before the actual action being annotated. Code 2 shows an annotated sample of the code from the example in the "Systematic Testing" section.

```
1  void add_to_runqueue(thread_t *child) {
2          tell_landslide_thread_runnable(child->tid);
3          // ... more implementation follows ...
4  }
5  int thread_fork() {
6          thread_t *child = construct_new_thread();
7          tell_landslide_forking(child->tid);
8          add_to_runqueue(child);
9          return child->tid;
10 }
```

Code 2. The same example *thread_fork()* implementation, now with annotations for use with Landslide.
Source: Landslide: Systematic dynamic race detection in kernel space.[3]

There is also a configuration file, *config.landslide*, in which the student must specify constant information such as the function names of the timer handler and context switcher, which threads exist when the kernel boots, and which user-space test program Landslide should invoke.

Finally, there are two short (nominally two-line) functions used within Landslide itself that the user must implement. These are predicates on the kernel's scheduler state and express potentially nontrivial conditions: whether the current thread is runnable but not on the run-queue, and whether preemption is disabled while interrupts are on. This logic executes within Landslide, inside of Simics, rather than as part of the simulated kernel's execution.

Configuring Decision Points

If Landslide uses only decision points that it automatically identifies on voluntary reschedules, the resulting interleavings will be coarse-grained and likely to overlook bugs. We provide an extra annotation for students to add more decision points for a finer-grained search, called *tell_landslide_decide()*. We recommend inserting it into concurrency primitives, such as at the start of *mutex_lock()* and at the end of *mutex_unlock()*.

However, this strategy may cause Landslide to identify decision points in unrelated parts of the kernel, such as when accessing mutexes in unrelated and/or already-trusted system calls. We provide interface options in *config.landslide* for the student to view currently identified decision points and to selectively eliminate them. For example, if a student were testing thread death and reaping, they might want decision points to appear in *wait()* and *vanish()* but not if unrelated virtual memory operations are also in progress. Accordingly, they could write *within_function wait vanish* and *without_function destroy_address_space*. The *within_function* directive requires that at least one of the specified functions shall be on the call stack when decision points are identified, and *without_function* requires the opposite.

Decision Traces

When Landslide identifies a bug, it outputs a *decision trace*. This trace reports what kind of bug was detected, and also reports each decision point in the current interleaving: which thread was running, a trace of its stack when it was switched away from, and the thread that Landslide caused to preempt it. With this trace, the user can better understand the concurrent execution that exposed the bug. In Code 3 we show an example decision trace, which depicts a sequence of thread interleavings that can expose the bug in the example from the Systematic Testing section.

"With this trace, the user can better understand the concurrent execution that exposed the bug."

```
USE AFTER FREE: read from 0x15a8f0 at IP 0x104209
Block 0x15a8f0 was allocated by thread 3 at (...)
   and freed by thread 4 at (...)
Decision trace follows:
1:  switched from thread 3 -> thread 4 at:
   0x105a10 in context_switch,
   0x1041f4 in thread_fork,
   0x10362b in thread_fork_wrapper
2:  switched from thread 4 -> thread 3 at:
   0x105a10 in context_switch,
   0x104681 in yield,
   0x104570 in exit,
   0x103708 in exit_wrapper
Current thread 3 at:
   0x104209 in thread_fork,
   0x10362b in thread_fork_wrapper
Total decision points 24, total backtracks 5
```

Code 3. An example decision trace that Landslide would emit when it finds a bug. This particular decision trace represents the example use-after-free bug in `thread_fork()` presented earlier.
Source: Landslide: Systematic dynamic race detection in kernel space.[3]

Results

We evaluated Landslide in two ways: first, by instrumenting two prior-semester student kernels to measure the exploration time needed to find different races,

and second, by meeting with current-semester student volunteers, before they submitted their kernel for grading, to see if they could find bugs on their own with Landslide. (The volunteers were chosen from students with free time, and were therefore not chosen at random.)

In the first phase, we instrumented one kernel written by a teaching assistant in a previous year and also one student kernel later graded by that TA. We configured Landslide to search for five complicated well-known race conditions. In addition to finding all five races, Landslide also found a sixth previously unknown race in the TA's own kernel. Using additional decision points only on calls to *mutex_lock()*, Landslide found each of the six bugs in 11 to 57 seconds on a 2.6 GHz Intel® Xeon® server, executing between 1 and 377 distinct interleavings per bug.

"In addition to finding all five races, Landslide also found a sixth previously-unknown race in the TA's own kernel."

In the user-study phase, we found that students spent on average 119 minutes (60 to 158) on the required instrumentation, and a further 36 minutes (10 to 60) refining Landslide's search. Of the four groups who finished the required instrumentation, all four found previously unknown bugs in their kernels: two races and two deterministic errors. These bugs manifested as infinite loops, a kernel panic, and a use-after-free. Despite wishing the instrumentation were easier, the students reported that they found working with Landslide rewarding.

"All four groups found previously unknown bugs in their kernels: two races and two deterministic errors."

Future Work

There are several promising future work directions for Landslide that we would like to explore. These include incorporating new testing techniques, such as parallelized search, state space estimation, and new state space reduction techniques. They also include extending Landslide to support more complicated kernel features, such as symmetric multiprocessing and device driver nondeterminism.

Other Testing Techniques

The most notable bug-detection predicate that Landslide does not yet incorporate is data race detection.[2][7] A data race is defined as a pair of memory accesses done by two distinct threads on the same address, at least one of which is a write, where there is no synchronization or dependency between the two threads at the time of either access. Many tools already exist for identifying data races, but we anticipate that searching for them with Landslide could additionally help guide Landslide's search towards thread interleavings more likely to have bugs based on such data races.

Ongoing research exists in several other techniques for coping with the exponential nature of the state spaces associated with systematic testing. Among these are parallelized dynamic partial order reduction[8] and dynamic interface reduction[9].

Extending Landslide's Concurrency Model

Landslide's present incarnation makes several limiting assumptions about

the concurrency model of the kernel under test. Chief among these are the assumptions that the kernel schedules threads only on one processor at a time, and that the timer interrupt is the kernel's only source of nondeterminism.

We anticipate revising the concurrency model to incorporate SMP scheduling would be a relatively minor change, as the overall structure of the state space tree remains the same, though some context switches would instead be cross-CPU switches. Unlike all context switches in the current uniprocessor model, such context switches would not necessarily involve executing any scheduler code. Incorporating device driver nondeterminism, however, will be more of a challenge, as in addition to context-switching to an arbitrary thread at any decision point, nondeterminism can also arise from either taking interrupts to receive input from a device or from context switching to a device driver's dedicated handler thread.

Lifting these limitations would be a significant step towards making Landslide applicable to real-world kernels such as Linux. Overall, we are optimistic for the future of systematic testing for concurrency bugs, and we hope to see sophisticated bug-finding tools along these lines in due time.

References

[1] Patrice Godefroid. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. In Proceedings of the 9th International Conference on Computer Aided Verification, CAV '97, pages 476–479, London, UK, 1997. Springer-Verlag.

[2] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In Proceedings of the Workshop on Binary Instrumentation and Applications, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[3] Ben Blum. Landslide: Systematic dynamic race detection in kernel space. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 2012. CMU-CS-12-118.

[4] David A. Eckhardt, "Pebbles kernel specification," 2012. [Online]. Available: http://www.cs.cmu.edu/~410/p2/kspec.pdf.

[5] Randy Bryant and David O'Hallaron, "Introducing computer systems from a programmer's perspective," in Proceedings of the 32nd Technical Symposium on Computer Science Education (SIGCSE). Charlotte, NC: ACM, Feb. 2001.

[6] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '05). USENIX Association, Berkeley, CA, USA, 41-41.

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

[8] Jiri Simsa, Randy Bryant, Garth Gibson, Jason Hickey. Scalable dynamic partial order reduction. Third International Conference on Runtime Verification (RV2012), 25–28 September 2012, Istanbul, Turkey.

[9] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical software model checking via dynamic interface reduction. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11). ACM, New York, NY, USA, 265–278.

Author Biographies

Ben Blum is a PhD candidate in the Computer Science Department at Carnegie Mellon University. He first implemented Landslide as the research topic for his Master's degree at CMU and is continuing the work during his PhD studies. Ben has additionally served as a teaching assistant for 15-410 for three semesters. His web site is at http://www.cs.cmu.edu/~bblum.

David A. Eckhardt is an associate teaching professor of computer science at Carnegie Mellon University. He joined the faculty after completing his MS and PhD at Carnegie Mellon and BS in Computer Science at The Pennsylvania State University. Dave received an Intel Foundation Graduate Fellowship and was co-inventor of a patent with Intel Senior Fellow Kevin Kahn. He has taught Operating Systems at CMU continuously since 2003 and has supervised student projects based on the Linux, FreeBSD, Haiku, OS X, and Plan 9 operating systems. Dave's research interests include operating systems, wireless networks, and high-performance networking. His web site is at http://www.cs.cmu.edu/~davide/.

Garth Gibson is a professor of computer science at Carnegie Mellon University, the cofounder and chief scientist at Panasas Inc., and a Fellow of the ACM. He has an M.S. and Ph.D. from the University of California at Berkeley and a BMath from the University of Waterloo in Canada. Garth's research is centered on reliable scalable storage systems for parallel and cloud computing and he has had his hands in the creation of the RAID taxonomy and the IETF NFS v4.1 parallel NFS extensions. Garth is also an investigator in the Intel Science and Technology Center for Cloud Computing. His web site is http://www.cs.cmu.edu/~garth/.