Landslide:
Systematic Dynamic Race Detection in Kernel-space
(User Guide)

Benjamin Blum (**bblum**)

April 11, 2012

1 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 is difficult to reproduce them reliably enough to collect enough information to help debug. In kernel-space, race condition debugging becomes even more difficult, as many aspects of the concurrency implementation itself are part of the system being tested, and may themselves have race-inducing flaws.

Landslide[1] is an effort to make easier the process of debugging kernel-space races. It is geared towards kernels that meet the Pebbles specification, which students in Operating System Design and Implementation (15-410) at CMU implement, and implemented as a module for Simics, the x86 simulator that students use to run their Pebbles kernels. During execution of a kernel, Landslide records important actions performed by the kernel, attempts to decide at which points in the kernel's execution a preemption will be most likely to expose a bug, and then exercises all possible interleavings of kernel threads around such points. When Landslide finds a bug (determined with a set of various checks and heuristics), it stops execution and prints information about the sequence of interleavings that caused the bug to show up.

With Landslide, we see testing a kernel as a process of manipulating test parameters in two ways: first, in the choice of test case (the userspace program that exercises a specific set of system calls), and second, in the configuration of Landslide in regard to which parts of the kernel are "interesting" in the behaviour of the test case and which are irrelevant. Searching for and understanding race conditions exposed by a given test becomes a joint effort between the programmer and Landslide, combining the programmer's specific knowledge about the design of the kernel and Landslide's ability to explore many interleavings efficiently.

2 How To Use This Document

If you want to start using Landslide as soon as possible, skip to appendix A and follow the step-by-step instructions, referring back to other sections for explanation as referenced.

Try not to get overwhelmed by the details here - you don't need to understand all the text in this document to use Landslide. Think of it like a reference manual.

The slides for the lecture which accompanies this user study are available at
`http://www.cs.cmu.edu/~410/landslide-lecture.pdf`.

[1] Landslide *(n)* - A phenomenon which demonstrates that Pebbles are not as stable as you might think.

3 Landslide's View of the World

3.1 On Simics

Landslide runs as a Simics module. When running the kernel, Simics calls into Landslide once every time the kernel executes an instruction or performs a memory read or write. Landslide uses this information to determine "what the kernel is doing" at each instruction, and triggers timer interrupts in certain patterns to cause a currently-running kernel thread to get preempted by another one.

Landslide also makes use of Simics "bookmarks" to checkpoint and restore the execution state at certain points in the execution of the test. This enables it to try multiple possibilities in terms of which kernel threads run when.

Once Simics is run with a custom `config.simics`, the above is all done automatically.

3.2 Components of Landslide

Between getting called at every instruction and deciding when to trigger timer interrupts or checkpoint or restore execution state, Landslide needs to model the kernel's behaviour and the concurrency properties of the test case. Several different components help achieve this.

3.2.1 Kernel Instrumentation

Landslide needs to know when the kernel performs certain types of actions. The user provides this information in several different ways, as discussed more in section 6.

The build script, before building Landslide itself, turns this information into a header file which contains `#define`s for addresses of functions within the kernel. Internally, Landslide compares `%eip` against these addresses to know when the kernel runs what function.

Because every kernel is slightly different, the communication between Landslide and your kernel will be unique, so you will need to spend some thought on these. More on this later.

3.2.2 Scheduling

From several of the hooks, Landslide is able to maintain a queue of threads that mirrors the kernel's runqueue, and hence know which threads are runnable at any given point during the execution. (See section 5.1.)

At any point during the test (provided interrupts are on and preemption is enabled), Landslide may choose to preempt the currently-running thread and run somebody else. This is done by triggering successive timer interrupts to induce context switches until the desired thread is switched to.

3.2.3 Decision Points

A perfectly accurate race detection tool would need to consider every single instruction[2] as a potential point at which a preemption might expose a race condition. This would make enumerating all interleavings impossibly expensive, however, so a feasible race detection tool needs to have intelligent heuristics to judge which instructions are more likely to be places where preemptions will trigger race conditions. At these "decision points", then, Landslide will choose whether to continue running the current thread or to preempt and run a different one.

Good decision points are most likely to be voluntary reschedules (i.e., calls to `yield` or similar - Landslide provides these by default), calls to synchronization primitives, accesses to shared data structures, and so on.

You do not need to configure the set of decision points in order to use Landslide, but doing so is certainly more likely than using the default set. More on this in section 8.

[2] actually, just every single access to shared memory while interrupts are on

3.2.4 The Decision Tree

After one execution of the test case, Landslide analyses each decision point that was encountered, and chooses from one of them a thread that was runnable but different from the one that was run. It then uses Simics bookmarks to backtrack to that decision point, and causes timer interrupts in a pattern that causes the kernel to schedule the newly chosen thread, executes the test with this new interleaving, and repeats. In this way we see that the many possible interleavings, as defined by the set of decision points, form a tree, which represents all possible ways the threads can run concurrently.

Modelling the decisions and interleavings as a tree presents the "mysterious nondeterminism" of the race condition in a new way: If a test case can race, then the resulting bug will appear in some, but not all, of the branches of the tree. Therefore, "systematic dynamic race detection" means to explore all branches of the tree and test for the bug in each one.

You might imagine that if Landslide had to try each runnable thread at each decision point, exponential explosion would make most trees take forever to explore. Landslide uses an algorithm called *dynamic partial order reduction* to prune redundant branches out of the tree to speed up exploration. An intuitive explanation of the pruning: Imagine that at decision point A, threads 1 and 2 are runnable, and that they don't interfere with each other, so that running 1 followed by 2 yields the same state as running 2 followed by 1. Partial order reduction identifies redundancies resulting from such "independent" transitions, and eliminates them. You don't need to worry about partial order reduction itself, except to realise that using it means that adding extra decision points will increase the size of the tree, but usually not cause exponential explosion.

4 What to Expect

When working with Landslide, please remember to keep realistic expectations about the nature of the tool.

Landslide is not a stress tester or a fuzzer, like the `cho` family of test cases. Tests such as `cho` attempt to break the kernel by making many system calls in as many different calling patterns as possible, and hoping that some of them will fail. Landslide (when paired with a Landslide-friendly test case - see section 7) attempts to break the kernel by exploring all possible interleavings of kernel threads in one particular pattern of a few system calls. Systematic testing (Landslide) and stress testing (`cho`) are orthogonal testing strategies.[3]

Landslide is not a "race condition oracle". If you run Landslide with a certain test case and a certain set of decision points, and it finishes exploring the decision tree and found no bugs, it does not mean that the system calls that got executed are free of race conditions. It does, however, mean that the interactions caused by the test case's pattern of system calls do not race at the granularity of the specific decision points chosen.[4]

Landslide is a *framework* that enables you, the programmer, to explore properties of a concurrent test case that you previously could not. It relies on you to configure it accurately in accordance with your goals: whether you are seeking to find a race that you suspect exists, or seeking to show that your kernel doesn't race on a particular test, it is your responsibility to choose a set of decision points that is relevant, granular enough to yield a meaningful exploration, yet minimal enough that Landslide can finish in a reasonable amount of time. More on this in section 8.

One final thing: It may seem tempting at times to look for "simpler" ways of circumventing some of the more complicated parts of the instrumentation process. Please do not do this - it will make your instrumentation brittle, and quite likely to break in more mysterious ways. If you are confused about what some part of the process is asking you to do, do not guess randomly (it will lead to frustrating crashes!); ask Ben for help instead.

[3] Also note that it is completely infeasible to run `cho` under Landslide. Because the test case does so much stuff on its own, the decision tree will be enormous, and Landslide will not be able to make any meaningful progress trying to explore it.

[4] Note that if you chose every instruction to be a decision point, the granularity would be perfect, and then you could say there are no races in the specific test case. This would make the decision tree impossibly large, though.

5 Kernel Requirements

5.1 Scheduler Functionality

In order to cause desired preemptions, Landslide needs to assume that the core of your scheduler works. In short, this means that timer interrupts will trigger context switches, and that if a thread is "runnable" (see section 6.1.1), at any point where interrupts/preemption is enabled, a finite number of timer preemptions will eventually cause that thread to run.

For deadlock detection, if you have mutexes which loop around a call to `yield` (or similar), they must be annotated as described in section 6.1. If you have mutexes which explicitly deschedule blocking threads, no special annotations are needed.

It is not okay to spin-wait anywhere, whether in mutexes or otherwise, especially in `readline` or `sleep`. Relatedly, the kernel must never run the idle loop when not truly idle, because Landslide uses this as a way of telling when all threads are wedged and/or when the test is done running.

5.2 VM

Landslide does not have much to do with the VM, but your kernel must direct-map most of kernel memory (i.e. below `USER_MEM_START`), including the heap, globals, and kernel thread stacks. Landslide needs this to read values out of memory and to identify memory conflicts.

Your kernel should also use LMM (the `malloc` package in the base code) for dynamic allocation, with the default configuration that the base code provides.

5.3 System Calls

In order to run with Landslide, your kernel must be able to boot up to the shell prompt, receive keyboard input to make the shell start a test case, and return to the shell prompt after the test case finishes. This means you must have the following system calls implemented and working in at least a rudimentary way: `fork`, `exec`, `vanish`, `wait`, `readline`.

The following system calls are exercised in some, but not all, of the test cases we distribute, so will be helpful to have: `thread_fork`, `yield`.

Apart from the system calls which init and shell execute, the rest are irrelevant and do not need to be implemented to use Landslide on your kernel.

6 Instrumenting Your Kernel with Landslide

Landslide tries to be as design-agnostic as possible, after assuming that the kernel implements the Pebbles specification, but it still needs to know about the implementation of certain abstractions within the kernel.

6.1 Annotations

We provide a set of functions that your kernel needs to call at certain points during its execution to communicate what interesting events are happening.

6.1.1 Important Note about Runqueues

Landslide uses two of these annotations, `on_rq()` and `off_rq()`, for tracking the state of the scheduler's runqueue at each point during execution. Note that this refers to the *actual runqueue data structure*, which does not necessarily correspond to the abstract "set of all runnable threads", depending on whether or not your kernel keeps the current thread on the runqueue or off of it.

If your kernel does not keep the currently-running thread on the runqueue, you'll need to tell Landslide this separately, in `kern_current_extra_runnable` - see section 6.3.

6.1.2 Required Annotations

- `tell_landslide_thread_switch(int new_tid)` - Call this in your context switcher, when a new thread is switched to.
- `tell_landslide_sched_init_done()` - Call this when your scheduler is done being initialized. Until this is called, Landslide will ignore all other annotations. (This corresponds to `starting_threads` in section 6.2.)
- `tell_landslide_forking()` - Call this when a new thread is being forked. (Hint: call it twice, once in `fork` and once in `thread_fork`.)

  Call it "just before" the action which makes the new thread runnable - whether your kernel just adds it to the runqueue (in which case the next annotation called would be `thread_on_rq`) or begins running it immediately (in which case `thread_switch`); Landslide handles both cases.[5]
- `tell_landslide_vanishing()` - Call this when a thread is about to go away. Call it "just before" the relevant context switch; i.e., the one that should never return.
- `tell_landslide_sleeping()` - Call this when a thread is about to go to sleep (that's `sleep()` the system call). Call it "just before" the relevant context switch; i.e., the one that won't return for as long as the thread requested to sleep.
- `tell_landslide_thread_on_rq(int tid)` - Call this when a thread is about to be added to the runqueue.[6] (Make sure this call is done within whatever protection is also used for the runqueue modification itself.)
- `tell_landslide_thread_off_rq(int tid)` - Call this when a thread is about to be removed from the runqueue. (Same protection clause as above applies.)
- Mutex annotations - These are only necessary if your kernel uses mutexes that leave blocked threads on the runqueue, rather than explicitly descheduling them.
  - `tell_landslide_mutex_locking(void *mutex_addr)` - Call this when a thread is about to start locking a mutex. Landslide needs the mutex address for deadlock detection, to track which mutex each thread is blocked on.
  - `tell_landslide_mutex_locking_done()` - Call this when a thread finishes locking a mutex (and now owns it).
  - `tell_landslide_mutex_unlocking(void *mutex_addr)` - Call this when a thread is about to start unlocking a mutex.
  - `tell_landslide_mutex_unlocking_done()` - Call this when a thread finishes unlocking a mutex (and no longer owns it).
  - `tell_landslide_mutex_blocking(int owner_tid)` - Call this when a thread is contending on a locked mutex, and hence is no longer logically "runnable" yet still on the runqueue. Landslide needs to know who owns the mutex for deadlock detection.

[5] It may help to think of this as setting a "currently forking" state flag in the calling thread: when the next annotation is called, Landslide will see the flag is set and act accordingly. It is okay if interrupts are on between this call and the subsequent event.

[6] If there are multiple such places, which is likely if you use `variable_queue`, we recommend wrapping them in a helper function which contains the annotation.

6.1.3 Optional Annotations

- `tell_landslide_decide()` - Call this to define additional decision points. See section 8.
- `assert()` or `panic()` - You should be using these already, but they are extra useful in Landslide. The more important invariants for which you have `assert`s in your kernel, the more likely Landslide is to find bugs that would trigger them - otherwise, even if they do happen, Landslide might never know (just like in conventional stress testing). [7]

6.2 Config File

There is also a config file you need to fill out. Some of the fields are required information, and some are tweaks that change the way Landslide behaves. The specific meaning of each field is explained in comments in the file; here we describe their purpose.

6.2.1 Required Fields

- `KERNEL_IMG` - The build scripts use `objdump` on this file to extract code and data addresses.
- `KERNEL_SOURCE_DIR` - Used for file and line number information in stack traces.
- `TEST_CASE` - What userspace program should Landslide explore the state space of?
- `TIMER_WRAPPER` - Used for controlling timer interrupts and tracking whether a thread is handling an interrupt.
- `CONTEXT_SWITCH` - Used for detecting voluntary reschedules.
- `READLINE` - Used for detecting when the kernel is ready to run a test, and when the test finishes.
- `INIT_TID`, `SHELL_TID`, `IDLE_TID`, `FIRST_TID` - Used for initialization, and tracking of test state.
- `starting_threads` - This lets you tell Landslide what threads exist at the point where `tell_landslide_sched_init_done` will be called. You should probably use it for each of init and (if it exists in your kernel) idle (but probably not shell).

6.2.2 Optimizations

These configuration options help Landslide identify potentially-conflicting shared memory accesses that it should ignore.

The partial order reduction algorithm achieves state space pruning by identifying *independent* transitions between decision points - that is, they that have no conflicting memory accesses. Telling Landslide to ignore some common types of shared accesses results in more independence between transitions, and hence more reduction, and hence a faster-running test. Of course, ignoring a memory access means you likely won't find races involving it, so specify only those that you are confident about.

- `sched_func` - When functions that make up the kernel's scheduler are identified, Landslide knows to ignore shared memory accesses from them. Accesses to scheduler data structures happen during *every* transition, so ignoring these is necessary for achieving any reduction at all.
- `ignore_sym` - You may wish to ignore memory accesses from other global data structures as well. For example, if you have a mutex which is locked very frequently, but irrelevant to the test case you're actually running, ignoring that mutex results in an independence relation that better reflects what you actually care about.

[7] Relatedly, a useful trick is to use poison values for memory that should not be read from until it is written again, and to use canary members in data structures that should not get scribbled over. Functions that operate on such memory can assert that poison values don't appear and that canary values don't change.

- `within_function` - If you configure decision points to happen in common code paths, such as `mutex_lock`, it will likely generate more branching than you need. (For example, if you're testing `vanish`, there's no need to branch on a `mutex_lock` in `exec`.) This configuration option allows you to whitelist functions so that Landslide will only decide if the current thread is "within" one of those functions.
- `without_function` - Similar to above, but a blacklist. These two can be used together; later invocations take precedence over earlier ones.

6.2.3 Behaviour Tweaks

- `BUG_ON_THREADS_WEDGED` - Landslide's "all threads wedged" detection (i.e., nobody runnable, not necessarily mutex-based deadlock) is not perfect. The default is to let the kernel keep running, but if it seems like you have an all-threads-wedged bug, turn this on.
- `EXPLORE_BACKWARDS` - In which order to explore the branches? Backwards means with more preemptions first; forwards means tending to let the current thread keep running more often. Backwards tends to find bugs more quickly, but produces longer decision traces.
- `DECISION_INFO_ONLY` - This option makes Landslide stop after one branch, and output the list of decision points it identified using the current config. Useful if you want to see all decision points, and tweak settings to get rid of frivolous ones.
- `BREAK_ON_BUG` - If you want to see how long Landslide takes to explore, this should be off (i.e., Landslide will make simics quit when it's finished). If you want to use the simics debug prompt when a bug is found, turn this on.

6.3 In-Landslide Functions

There are two functions that you need to implement in Landslide itself, to express certain kernel designs that might be more complicated than a simple true/false condition or integer. They are in `work/modules/landslide/student.c`.

You may need to read some of your kernel's global variables (pointers or state flags) to implement these functions. To get access to these, use `extra_sym` in `config.landslide` (which is explained in the comments there).

- `bool kern_current_extra_runnable(conf_object_t *cpu)` - See section 6.1.1. This says whether the current thread is "logically runnable" despite not being on the runqueue itself (that is, `tell_landslide_on_rq` won't have been called for it).
- `bool kern_ready_for_timer_interrupt(conf_object_t *cpu)` - This function expresses whenever a clock interrupt will trigger a prompt context switch. Landslide will avoid trying to preempt your kernel whenever this is false.

The codebase also provides the following macros to help you implement these. The `cpu` argument to the above functions is an opaque object which these macros need.

- `GET_CPU_ATTR(cpu, name)` - returns the value of the cpu's specified register. (e.g., write `eax` for `name`).
- `READ_MEMORY(cpu, addr)` - retrieves the 4-byte value at the given address.
- `READ_BYTE(cpu, addr)` - retrieves the 1-byte value at the given address.
- `GET_ESP0(cpu)` - retrieves the value of the pseudo-register `%esp0`.

7 Userspace Test Cases

We ship Landslide with a suite of small test cases designed to expose several common races that students frequently encounter during 15-410 Project 3.[8]

- `vanish_vanish` - Tests when a parent and child process `vanish()` simultaneously. (This test is identical to `fork_test1` from the P3 hurdle suite.)
- `fork_wait` - Tests basic interaction of `fork()` and `wait()`. (This test is from the P3 hurdle suite.)
- `double_wait` - Tests interactions of multiple waiters on a single child. (new)
- `double_thread_fork` - Tests for interactions of multiple threads in one process vanishing. (new)[9]
- `yield_vanish` - Tests for interactions between `yield()` and `vanish()`. (new)

It may help your intuition to note why all of these test cases are "Landslide-friendly": they all perform very little work on a single run, enabling Landslide to completely explore the state-space. They also run several, but not too many, threads at once, producing potentially interesting interleavings.

To warm up, we recommend you test `double_thread_fork` (and edit your kernel explicitly to introduce the associated bug - see the footnote in the test's description above). After that, we recommend trying `vanish_vanish`, which is the most likely to expose many different types of bugs in many different kernel designs.

8 Configuring Decision Points

Getting a good set of decision points is important to being able both to explore reasonably quickly and to produce meaningful interleavings that are likely to find bugs. After you get Landslide working with the default set of decision points, it's time to add some more.

In no particular order, you should be thinking about the following things when building the set of decision points.

- Use `tell_landslide_decide()` to indicate choice points. Recommended places are at the start of `mutex_lock`, and if that doesn't find bugs, at the end of `mutex_unlock` as well. But use your own intuition, depending on what system call(s) you're testing.
- Use `sched_func` and `ignore_sym` to make Landslide ignore global memory accesses, to enable better pruning.
- Use `DECISION_INFO_ONLY` to examine the set of choice points, and figure out which ones you want to get rid of.
- Use `within_func` and `without_func` to make Landslide only pay attention to parts of your kernel that you care about.

Example tip: If you `tell_landslide_decide()` in `mutex_lock`, you could surround the call with an if statement that makes sure the mutex is not some global mutex you don't care about. You could also (or instead) make Landslide ignore it by using `without_func` on whatever function takes the mutex you don't care about.

[8] You may wonder why some of these are written in assembly. This is because using your userspace thread library for thread creation (a) is not guaranteed to work, (b) might forcibly serialise certain operations that would expose kernel bugs only if concurrent, and (c) necessitates many more system calls, which dramatically increases the size of the decision tree.

[9] This test tests for the use-after-free bug presented in lecture - if your `thread_fork` implementation has "`return child->tid;`" as its last line of code.

9 Interpreting Landslide's Results

Landslide prints a lot of information. Most of it is only useful for development, but you may find some of it interesting anyway.

The most important thing that Landslide prints is the decision trace, when it finds a bug. Landslide prints the following information:

- Before proclaiming "A bug was found!", there should be a message in bright red explaining what the bug is.
- The trace itself consists of a list of the decision points that were encountered in the buggy interleaving. For each decision point, Landslide says what thread used to be running, what thread it chose to run next, and the stack trace of the old thread at the point it was switched away from.
- At the end, there will be a stack trace for the current point in time, which will be where the bug was detected, and not a decision point itself.
- Also at the end there will be some statistics about the exploration: how many decision points were visited, how many backtracks were taken (i.e., how many interleavings were explored), the average number of decision points per branch, etc.

We ship Landslide with a somewhat high verbosity setting. If you want it to be quieter, you can edit it in `work/modules/landslide/common.h` - `CHOICE` is the recommended minimum level (`INFO` is highest, `DEV` is default).

10 Feedback

Ben needs feedback about your experience using Landslide to help him present his thesis. He wants to find answers to the following questions:

- What ways can Landslide be improved to make it a better race-finding tool? (This means in terms of the high-level process of finding useful decision points, less so in terms of nasty implementation details - the latter would be good for development and bug reports, but has little research value.)
- Does using Landslide make you think differently about race conditions, in a way that makes you better at debugging them, even at times when you're not using Landslide?
- Is Landslide effective at finding bugs, when properly configured? Also, is it effective at saying it found no bug when using a reasonably-thorough set of decision points?

Of course, you shouldn't try to answer those questions directly. But keep them in mind as the goal of the study, when answering the following questions.

10.1 While using Landslide

While you're using Landslide, please try to take 5 minutes approximately every half-hour to give short updates. For convenience, you can print out appendix B and fill it in as you go.

10.2 After using Landslide

After you've finished with Landslide, please write some conclusions about your experience:

- Were you able to get Landslide to work with your kernel (i.e., run through a minimal exploration tree completely)? If so, how long did it take, from sitting down to getting it to work? If not, did your kernel do something that was incompatible with Landslide, or were you unable to get the instrumentation right for some other reason?

- Did you find bugs while using Landslide, that you imagine would have been very hard to find otherwise? Did Landslide's output help you understand what caused them?
- Were you able to get Landslide to say "you survived!" with custom decision points? Did you think the set of decision points you used for this provide a strong guarantee about the absence of races?
- Describe your experience configuring the set of decision points. What was intuitive, obvious to do? Did you feel stuck at any point, not knowing where to go next?
- Was there anything you wanted to make Landslide do that it didn't support?

A Step-by-Step Setup

We recommend you work on the Andrew Linux machines, as they are supported by the 410 environment, and Landslide has only been tested with Simics installation on AFS.

As a way of keeping the attack plan steps small, we recommend not adding/configuring custom decision points until after getting Landslide working with the automatic minimal set of decision points (i.e., it explores the tree to completion).

- Get the repository: `http://www.cs.cmu.edu/~410/landslide.tar.bz2` [10]
- Read section 5 to make sure your kernel is Landslide-compatible. (If not, fix it and/or ask Ben for help.)
- Build your kernel for Landslide
  - Run `prepare-workspace.sh`
  - Copy your kernel sources/build directory into `pebsim/`.
  - Add `tell_landslide.c` and `tell_landslide.h` to your kernel, and add `tell_landslide.o` to your kernel's `config.mk`.
  - Copy in the test program(s) you want to run, from `tests/`, and add them to your kernel's `config.mk`. (section 7)
  - Add calls to the annotation functions in your kernel. (section 6.1)
  - Build your kernel, and put `bootfd.img` and `kernel` in `pebsim/`.
- Fill out `config.landslide` in `pebsim/`. (section 6.2)
- Implement the functions in `student.c` in `work/modules/landslide`. (section 6.3)
- In `pebsim/`, run `./landslide`. This will build everything (if necessary), and then run Landslide.
- Iterate reconfiguring Landslide until satisfied with the results. (section 8)
- Fill out feedback. (section 10)

[10] All directory paths henceforth are relative to the base directory of the repository.

B While-You-Work Feedback Form

Questions to be thinking about while you're working:

- How much time did you spend on each phase of the process? (annotating your kernel, doing `config.landslide`, writing `student.c`, fixing instrumentation problems/crashes, customizing decision points for more efficient search, or trying to understand/solve a race that was found)
- What problems were you fighting, getting things to work? (conceptual understanding, fixing instrumentation, finding good decision points)
- Did you have to change your kernel at all to make it compatible?
- Is there a feature you wish Landslide had to help you out, that it doesn't offer?
- Did you realize anything interesting or surprising about Landslide, or about your kernel?

Please fill out a row after every 30-60 minutes spent. Approximate time estimates are totally fine.

Time	Minutes spent doing...						Remarks (problems, realizations, features)
	annotate	config	student.c	fixing	customizing	found races	