

Systematic Dynamic Race Condition Detection

more clever than “slaughter cho2” since 2011.

Ben Blum (bblum@andrew.cmu.edu)

Carnegie Mellon University - 15-410

2012, April 6

Outline

Theory: Seeing race conditions in a new way

- ▶ Case study (example)
- ▶ The execution tree
- ▶ Decision points

Technique: Systematic testing

- ▶ Requirements
- ▶ Challenges and feasibility

Tool: Landslide

- ▶ How it works
- ▶ The user-tool relationship
- ▶ User study (that's you!)

Case Study

```
int thread_fork()
{
    thread_t *child = spawn_new_thread();
    add_to_runqueue(child);
    return child->tid;
}
```

Decision Points (“good” case)

Thread 1	Thread 2
<code>spawn_new_thread</code>	
<code>add_to_runqueue</code>	
<code>return child->tid</code>	
	<code>vanish</code>
	(TCB gets freed)

(new thread)

(voluntary reschedule)

Decision Points (race condition)

Thread 1	Thread 2
<code>spawn_new_thread</code>	
<code>add_to_runqueue</code>	
	<code>vanish</code>
	(TCB gets freed)
<code>return child->tid</code>	

(new thread + preempted)

(voluntary reschedule)

(bad!)

Testing Mechanisms

Stress testing: slaughter cho2 and friends

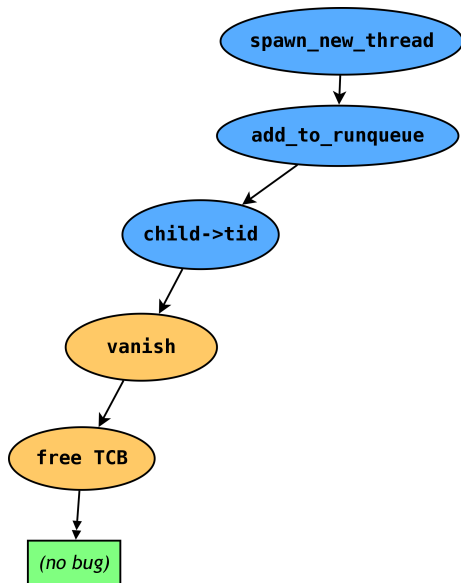
- ▶ Attempting to exercise as many interleavings as practical
- ▶ Exposes race conditions at random
 - ▶ “If a preemption occurs at just the right time...”
- ▶ Cryptic panic messages or machine reboots

What if...

- ▶ Make educated guesses about when to preempt
- ▶ Preempt enough times to run *every single* interleaving
- ▶ Tell the story of what *actually happened*.
- ▶ Overlook fewer bugs!

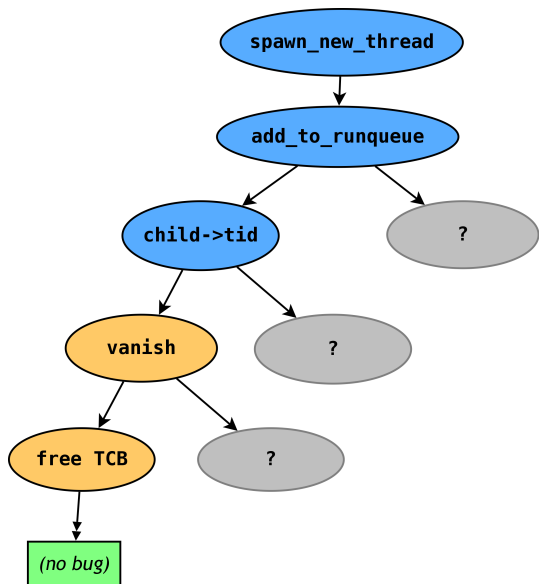
A different way of looking at race conditions. . .

Execution Tree



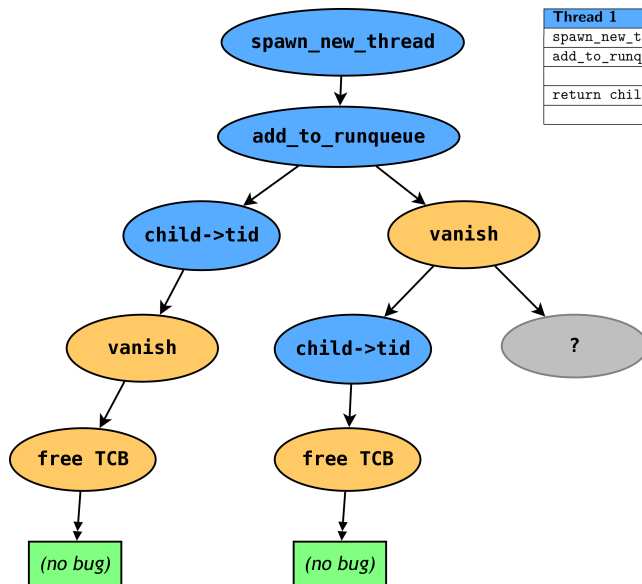
Thread 1	Thread 2
spawn_new_thread	
add_to_runqueue	
return child->tid	
	vanish
	(TCB gets freed)

Execution Tree



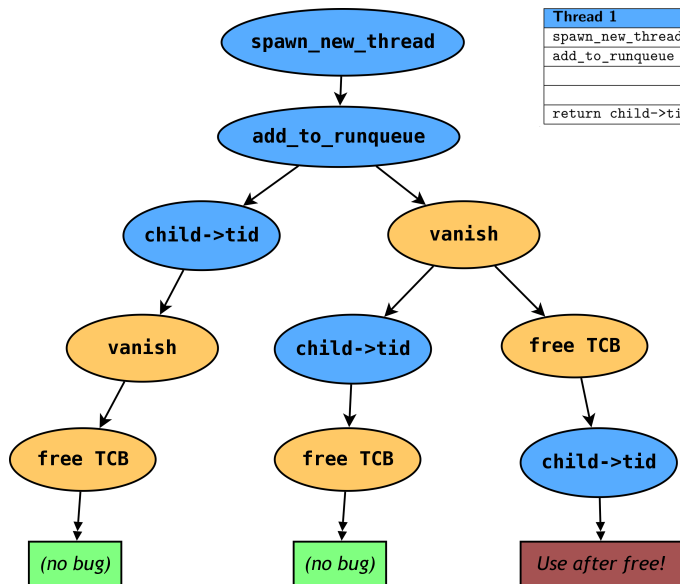
Thread 1	Thread 2
spawn_new_thread	
add_to_runqueue	
return child->tid	
	vanish
	(TCB gets freed)

Execution Tree



Thread 1	Thread 2
spawn_new_thread	
add_to_runqueue	
	vanish
return child->tid	
	(TCB gets freed)

Execution Tree



Thread 1	Thread 2
spawn_new_thread	
add_to_runqueue	
	vanish (TCB gets freed)
return child->tid	



Decision Points

A **decision point** is...

A code location where being preempted causes different behaviour.

- ▶ Intuitively: Somewhere that interesting interleavings can happen around.

Examples:

- ▶ A new thread becomes runnable
- ▶ Voluntary reschedule (e.g. `yield`, `cond_wait`)
- ▶ Synchronization primitives

Systematic Testing

Systematic Testing

Systematic testing is:

- ▶ Systematically enumerating different interleavings
 - ▶ Intuitively: Generate many “tabular execution traces”
- ▶ Exploring all branches in these trees
 - ▶ (by controlling scheduling decisions at decision points)
- ▶ In practice: Depth-first search of branches

Execution Tree Exploration

Important point: When does a branch end?

- ▶ All threads run to completion, or
- ▶ A bug is detected

Backtracking:

- ▶ Identify a decision point to choose differently
- ▶ Reset machine state and start over
- ▶ Replay test from the beginning, with different choices

More on Decision Points

Important point: What does “all possible interleavings” mean?

One extreme: Decide at every instruction

- ▶ Good news: Will find every possible race condition.
- ▶ Bad news: Runtime of test will be impossibly large.

Other extreme: Nothing is a decision point

- ▶ Good news: Test will finish quickly.
- ▶ Bad news: Only one execution was checked for bugginess.
- ▶ Bad news: No alternative interleavings explored.
 - ▶ Makes “no race found” a weak claim.

More on Decision Points

Sweet spot: Insert a thread switch everywhere it “might matter”.

When do we fear being preempted?

- ▶ New threads becoming runnable (`fork`, `cond_signal`, etc)
 - ▶ Preemptions may cause it to run before we're ready
- ▶ Synchronization primitives (`mutex_lock`, etc)
 - ▶ If used improperly...
- ▶ Unprotected shared memory accesses
 - ▶ May result in data structure corruption

Finding the sweet spot is a joint effort between programmer and tool.
(More on this later.)

Controlling Scheduling Decisions

Control over sources of nondeterminism

- ▶ Device interrupts/input
 - ▶ Disk drivers: when disk reads finish
 - ▶ Ethernet drivers: when packets arrive
- ▶ To control thread switches in a 410 kernel, vary when clock ticks happen.

Memory Interposition

In order to find use-after-free, need to know:

- ▶ When objects are `free()`d
- ▶ When threads access shared memory in the heap

Solution: Keep track of all memory events

- ▶ All calls to `malloc/free`
- ▶ All shared memory reads/writes

State Space Explosion

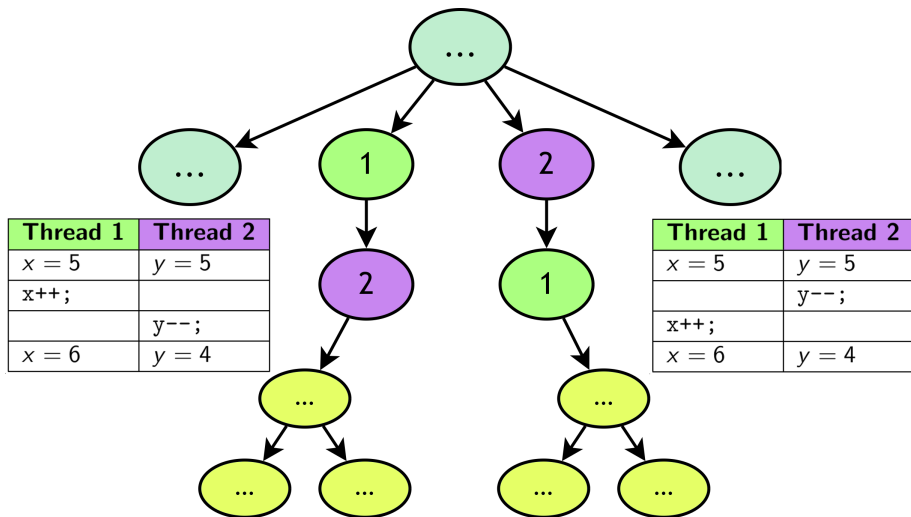
State spaces grow exponentially

- ▶ With d decision points, k runnable threads, size d^k .
- ▶ Threatens our ability to explore everything.
- ▶ Fortunately, some sequences result in identical states.

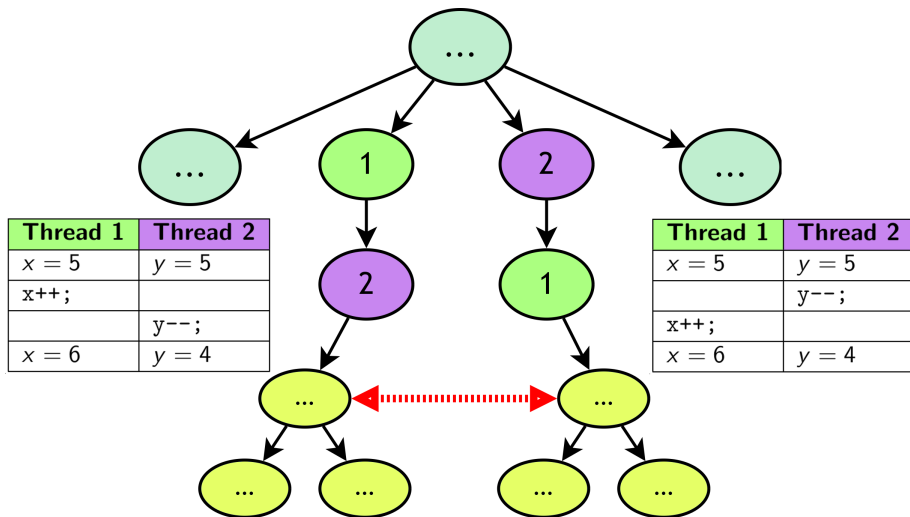
Partial Order Reduction can help.

- ▶ Complicated algorithm; ask me later for details.
- ▶ Intuitive explanation follows.

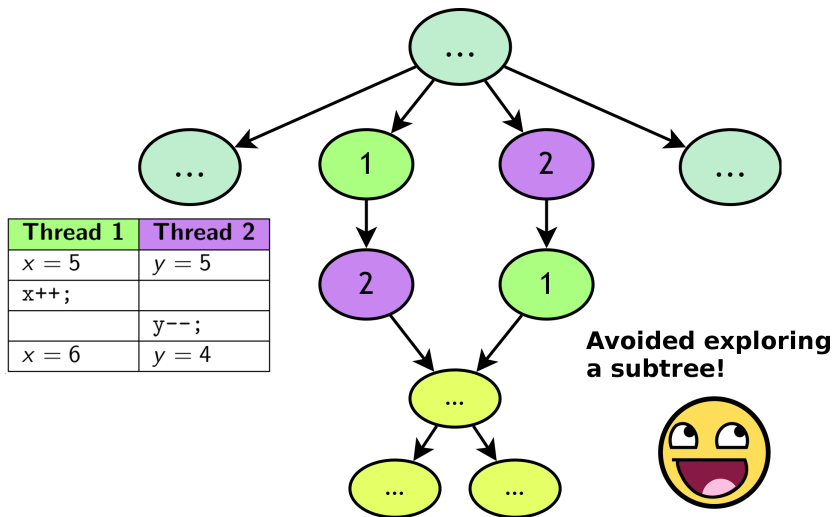
State Space Explosion



State Space Explosion



State Space Explosion



Landslide

About The Project

5th year MS since June 2011

Working with Garth Gibson, Jiri Simsa

Landslide: Shows that your Pebbles are not as stable as you thought.

Landslide in Simics

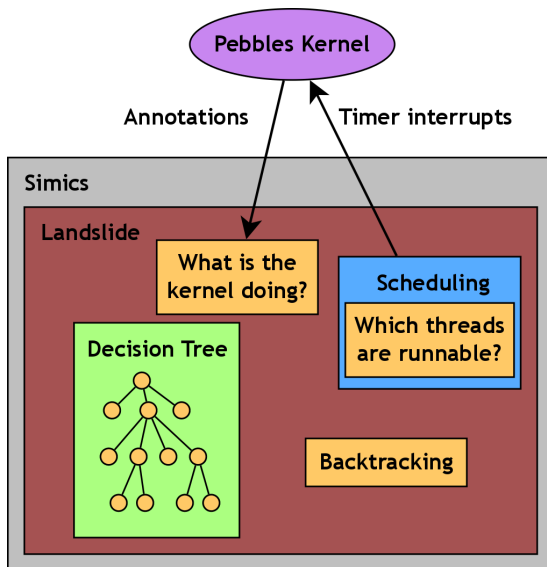
As a Simics module, Landslide knows:

- ▶ Every instruction the kernel executes
- ▶ Every memory address the kernel reads/writes

Artificially causes timer interrupts

Checkpointing/backtracking via Simics bookmarks

Anatomy



Identifying Bugs

Landslide can *definitely discover*:

- ▶ Kernel panics
- ▶ Deadlock
- ▶ Use-after-free / double-free

Landslide can *reasonably suspect*:

- ▶ Memory leak
- ▶ Progress sense (halting problem)

Using Landslide

In Which Ben Offers Help

This is something you can try!

Mutual benefit

- ▶ Landslide may help you find bugs
- ▶ You may help Ben evaluate his thesis project

Keeping It Real

Finding race conditions is hard for humans.

It is hard for computer programs too.

Landslide is not an oracle.

Annotating Your Kernel

Step 1

```
int thread_fork()
{
    thread_t *child = spawn_new_thread();
    tell_landslide_forking();
    add_to_runqueue(child);
    tell_landslide_decide(); /* Interrupt me here! */
    return child->tid;
}
```

Your kernel needs to say when certain events happen:

- ▶ When do threads become runnable / descheduled?
- ▶ When does the scheduler switch threads?

Time estimate: 40 minutes

Configuring Landslide

Step 2

```
# What test case to run?  
TEST_CASE="double_thread_fork"  
  
# The names of some important functions  
TIMER_WRAPPER="timer_handler_wrapper"  
CONTEXT_SWITCH="sched_switch"  
  
# What functions to pay attention to?  
within_function "thread_fork"  
within_function "vanish"
```

Edit `config.landslide` with some details and tweaks

Fill in two implementation-dependent C functions in Landslide (≤ 10 lines)

Time estimate: 60 minutes

Configuring Decision Points

Landslide automatically identifies a minimal set of decision points.

- ▶ It might find bugs.
- ▶ It might overlook more fine-grained interleavings.

With help from you, it could find more.

- ▶ Optional annotation: `tell_landslide_decide()`
- ▶ Hints to where a context switch should be forced.
- ▶ Inside every call to `mutex_lock...`

Quick Demo

In Which Ben Offers Help - Warning

If you are already struggling, this will not “save” you.

- ▶ False-negatives: not guaranteed to find races at all
- ▶ Research-quality: possibly difficult to integrate with your kernel
- ▶ Finishing the kernel project is more important.

You should:

- ▶ Be expecting an A or B
- ▶ Be able to turn in without late days (if you really had to)
- ▶ Be looking for. . .
 - ▶ That “one pesky race”
 - ▶ A race that stress-testing missed
 - ▶ Or just familiarity with a new technique

In Which Ben Offers Help

Your kernel

- ▶ Must load the shell and run programs
- ▶ `fork`, `exec`, `vanish`, `wait`, `readline`
- ▶ Must never spin-wait (see hurdle form!)
- ▶ Should `assert()` important invariants
 - ▶ Think of `panic()` as `tell_landslide_bug()`

In Which Ben Offers Help

User study next week, starting Monday

Expect to spend:

- ▶ Up to 4 hours, just to try it out.
- ▶ 6-8 hours, if you find a bug and track it down.
- ▶ More, for multiple bugs or the truly dedicated. . .

Give feedback (intuitive? frustrating? found bugs?)

Watch 410.announce for details!

Questions?