Landslide: A New Race-Finding Tool for 15-410

more clever than “agility_drill” since 2011.

Ben Blum (bblum@andrew.cmu.edu)

Carnegie Mellon University - 15-410

2018, October 28
Introduction

Outline

Theory: Seeing concurrency bugs in a new way
- Case study (example)
- Tabular execution traces
- The execution tree

Research Technique: “Systematic testing”
- Preemption points
- Challenges and feasibility

Tool: Landslide
- How it works
- Automatically choosing preemption points
- How to use it
Case Study

Consumer thread

```c
mutex_lock(mx);
if (!work_exists())
    cond_wait(cvar, mx);
work = dequeue();
mutex_unlock(mx);
access(work->data);
```

Producer thread

```c
mutex_lock(mx);
enqueue(work);
signal(cvar);
mutex_unlock(mx);
```
Case Study

Consumer thread

mutex_lock(mx);
if (!work_exists())
    cond_wait(cvar, mx);
work = dequeue();
mutex_unlock(mx);
access(work->data);

Producer thread

mutex_lock(mx);
enqueue(work);
signal(cvar);
mutex_unlock(mx);

▶ See *Paradise Lost* lecture!
▶ if vs while: Two consumers can race to make one fail.
### Thread Interleavings ("good" case)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(mx); if (!work_exists()) wait(cvar, mx);</td>
<td>lock(mx); if (!work_exists()) wait(cvar, mx);</td>
<td>lock(mx);</td>
</tr>
<tr>
<td>work = dequeue(); unlock(mx); access(work-&gt;data);</td>
<td>lock(mx); enqueue(work); signal(cvar); unlock(mx);</td>
<td></td>
</tr>
</tbody>
</table>

---

Ben Blum (CMU 15-410)
## Thread Interleavings (different “good” case)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(mx); if (!work_exists()) wait(cvar, mx); work = dequeue(); unlock(mx); access(work-&gt;data);</td>
<td>lock(mx); enqueue(work); signal(cvar); unlock(mx);</td>
<td>lock(mx); if (!work_exists()) wait(cvar, mx);</td>
</tr>
</tbody>
</table>
# Thread Interleavings (race condition)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lock(mx);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (!work_exists())</td>
<td>lock(mx);</td>
</tr>
<tr>
<td></td>
<td>wait(cvar, mx);</td>
<td>enqueue(work);</td>
</tr>
<tr>
<td></td>
<td>work = dequeue();</td>
<td>signal(cvar);</td>
</tr>
<tr>
<td></td>
<td>unlock(mx);</td>
<td>unlock(mx);</td>
</tr>
<tr>
<td></td>
<td>// SIGSEGV 😞</td>
<td>work = dequeue();</td>
</tr>
</tbody>
</table>
Testing

How can programmers be confident in the correctness of their code?

▶ Unit tests
  ▶ good for basic functionality, bad for concurrency
▶ Stress tests
  ▶ state of the art in 15-410
▶ Theorem proving
  ▶ heavy burden on the programmers
▶ Releasing to paying customers and worrying about correctness later

Motivation: Can we do better than stress testing?
Testing Mechanisms

**Stress testing**: largetest, mandelbrot 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 when failure occurs

What if...

- Make educated guesses about when to preempt
- Preempt enough times to run *every single* interleaving
- Overlook fewer bugs!
A different way of looking at race conditions...
Systematic Testing

Execution Tree

```
work_exists?
  cond_wait()
work_exists?
  cond_wait()
  cond_signal()
  enqueue()
  dequeue()
  work != NULL
  (no bug)
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(mx);</td>
<td>lock(mx);</td>
<td>lock(mx);</td>
</tr>
<tr>
<td>if (!work_exists())</td>
<td>wait(cvar, mx);</td>
<td>if (!work_exists())</td>
</tr>
<tr>
<td>work = dequeue();</td>
<td>signal(cvar);</td>
<td>wait(cvar, mx);</td>
</tr>
<tr>
<td>unlock(mx);</td>
<td>unlock(mx);</td>
<td>unlock(mx);</td>
</tr>
<tr>
<td>access(work-&gt;data);</td>
<td></td>
<td>access(work-&gt;data);</td>
</tr>
</tbody>
</table>
```
Systematic Testing

Execution Tree

```
work_exists?
cond_wait()

work_exists?
cond_wait()

cond_signal()

enqueue()

dequeue()

work != NULL
(no bug)
```

```
Thread 1
lock(mx);
if (!work_exists())
  wait(cvar, mx);

Thread 2
lock(mx);
if (!work_exists())
  wait(cvar, mx);

 bathrooms

Thread 3
lock(mx);
if (!work_exists())
  wait(cvar, mx);

work = dequeue();
unlock(mx);
access(work->data);
```
Systematic Testing

Execution Tree

```
work_exists?
  => cond_wait()

work_exists?
  => cond_wait()

cond_wait()
  => work_exists?

cond_signal()
  => enqueue()

**cond_wait()**

**cond_wait()**

**cond_wait()**

**cond_wait()**

**cond_signal()**

**enqueue()**

work = dequeue();
unlock(mx);
access(work->data);

work_exists?
  => cond_wait()

work != NULL
(no bug)

work != NULL
(no bug)

**cond_signal()**

**enqueue()**

Work exists?
  => cond_wait()

Thread 1

lock(mx);
if (!work_exists())
  wait(cvar, mx);

Thread 2

lock(mx);
enqueue(work);
signal(cvar);
unlock(mx);

Thread 3

lock(mx);
if (!work_exists())
  wait(cvar, mx);

Ben Blum (CMU 15-410)

Landslide

work != NULL
(no bug)

work != NULL
(no bug)```
Systematic Testing

Execution Tree

Thread 1 | Thread 2 | Thread 3
---|---|---
lock(mx);
if (!work_exists())
  wait(cvar, mx);
| lock(mx);
enqueue(work);
signal(cvar);
unlock(mx);
| lock(mx);
work = dequeue();
unlock(mx);

work = dequeue();
unlock(mx);
// SIGSEGV 😞

work != NULL (no bug)
work != NULL (no bug)
work == NULL Segfault!

work_exists?
cond_wait()
dequeue()
cond_wait()
Goal: Force the system to execute every possible interleaving.

- On 1st execution, schedule threads arbitrarily until program ends.
  - This represents one branch of the tree.
- At end of each branch, rewind system and restart test.
- Artificially preempt to interleave threads differently.
- Intuitively: Generate many “tabular execution traces”.

Okay, wait a sec...
Goal: Force the system to execute every possible interleaving.

- On 1st execution, schedule threads arbitrarily until program ends.
  - This represents one branch of the tree.
- At end of each branch, rewind system and restart test.
- Artificially preempt to interleave threads differently.
- Intuitively: Generate many “tabular execution traces”.

Okay, wait a sec...
Systematic Testing - The Big Picture

Goal: Force the system to execute every possible interleaving.

▶ On 1st execution, schedule threads arbitrarily until program ends.
  ▶ This represents one branch of the tree.
▶ At end of each branch, rewind system and restart test.
▶ Artificially preempt to interleave threads differently.
▶ Intuitively: Generate many “tabular execution traces”.

Okay, wait a sec...

▶ How can you possibly execute every possible interleaving?
▶ How did you know to draw that tree’s branches where they matter?
Preemption Point Example (remember this?)

```java
boolean want[2] = { false, false };

1  want[i] = true;                        (preemption point A)

2  while (want[j])
    // ...critical section...
    continue;                              (preemption point B)

3  continue;                              (preemption point C)

4  // ...critical section...

5  want[i] = false;
```

Some preemption points will expose bugs.
Some preemption points don’t matter.
Preemption Point Example (remember this?)

```java
boolean want[2] = { false, false };

1  want[i] = true;                             (preemption point A)
2  while (want[j])
3      continue;

4  // ...critical section...

5  want[i] = false;
```

Here, only preemption point A will cause a progress failure. All other interleavings are benign.
Preemption Points

Preemption points (PPs) are code locations where being preempted may cause different behaviour.

- IOW, somewhere that interesting interleavings can happen around.

Systematic tests are parameterized by the set of PPs.

- $n$ PPs and $k$ threads $\Rightarrow$ state space size is $O(k^n)$.
- Need to choose PPs very carefully for test to be effective.
  - “Effective” = both comprehensive and feasible.
Preemption Points

What does “all possible interleavings” actually mean?

One extreme: Preempt 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 preemption point
- Good news: Test will finish quickly.
- Bad news: Only one execution was checked for bugginess.
  - No alternative interleavings explored.
  - Makes “no race found” a weak claim.
Preemption Points

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

When are preemptions dangerous?

- Threads becoming runnable (`thr_create()`, `cond_signal()`, etc.)
  - Preemptions may cause it to run before we’re ready
- Synchronization primitives (`mutex_lock()`/`unlock()`, etc.)
  - If buggy or used improperly...
- Unprotected shared memory accesses ("data races")
  - May result in data structure corruption
  - More on this later...
Landslide

Landslide
About The Project

About me: Final year graduate student, advised by Garth Gibson

- TAed 15-410 for 3 semesters during undergrad
- Landslide’s publication history
  - Master’s thesis
  - Conference paper (OOPSLA 2016)
About The Project

About me: Final year graduate student, advised by Garth Gibson

- TAed 15-410 for 3 semesters during undergrad
- Landslide’s publication history
  - Master’s thesis
  - Conference paper (OOPSLA 2016)

About Landslide

- Simulator* module, which traces:
  - Every instruction executed
  - Every memory access read/written
  - *Used to be Simics, now Bochs (open-source, 3x faster)

- Landslide shows how your Pebbles programs may not be stable.
Big Picture: Execution Tree Exploration

Backtracking

- Each time test completes, identify a PP to replay differently
- Reset machine state and start over
- Replay test from the beginning, with a different interleaving
Big Picture: Execution Tree Exploration

Backtracking
- Each time test completes, identify a PP to replay differently
- Reset machine state and start over
- Replay test from the beginning, with a different interleaving

Controlling scheduling decisions
- Tool must control all sources of nondeterminism
- In 15-410, just timer and keyboard interrupts
- Landslide repeatedly fires timer ticks until desired thread is run.
Landslide & You

P2 (thread library)

system calls

Pebbles (reference kernel)

hardware drivers

Simulator (hardware emulation)
Landslide & You

- Simulator (hardware emulation)
  - Pebbles (reference kernel)
  - P2 (thread library)
  - Landslide

- Landslide manages multiple executions
- Pebbles examines memory, reads/writes
- P2 schedules threads
- Simulator injects timer, interrupts
- System calls

Ben Blum (CMU 15-410)
Identifying Bugs

Landslide can *definitely discover*:

- Assertion failures
- Segfaults
- Deadlock
- Use-after-free / double-free

Landslide can *reasonably suspect*:

- Infinite loop (halting problem)
- Data race bugs
What is a Data Race?

A **data race** is a pair of memory accesses between two threads, where:

- At least one of the accesses is a write
- The threads are not holding the same mutex
- The threads can be reordered (e.g., no `cond_signal()` in between)
What is a Data Race?

A **data race** is a pair of memory accesses between two threads, where:

- At least one of the accesses is a write
- The threads are not holding the same mutex
- The threads can be reordered (e.g., no `cond_signal()` in between)

Data races are *not necessarily* bugs, just highly suspicious!

- Bakery alg: Is `number[i]=max(number[0],number[1])+1` bad?
- What about unprotected `next_thread_id++`?
- “If threads interleaved the wrong way here, it *might* crash later.”
  - Hmmm...
Choosing the Right Preemption Points

How can we address exponential state space explosion?
Choosing the Right Preemption Points

How can we address exponential state space explosion?

State of the art tools choose a fixed set of preemption points.

- E.g., “all thread API calls” or “all kernel mutex locks/unlocks”
- Depending on length of test, completion time is unpredictable.
- More often, a subset is better in terms of time/coverage.
Choosing the Right Preemption Points

How can we address exponential state space explosion?

State of the art tools choose a fixed set of preemption points.
- E.g., “all thread API calls” or “all kernel mutex locks/unlocks”
- Depending on length of test, completion time is unpredictable.
- More often, a subset is better in terms of time/coverage.

Current systematic testing model is not user-friendly.
- Tool: “I want to use these PPs, but can’t predict completion time.”
- User: “I have 16 CPUs and 24 hours to test my program.”

*Stress testing allows user to choose total run time – can we offer this too?*
Iterative Deepening in Landslide

Landslide automatically iterates through different configurations of PPs.

- Manages work queue of jobs with different PPs
- Each job represents a new state space for Landslide to explore
- Prioritizes jobs based on estimated completion time

Repeat state space explorations, adding preemption points, until time is exhausted.

*Only required argument is CPU budget*
Iterative Deepening

Minimal state space includes only “mandatory” context switches

- e.g., yield(), cond_wait().
Iterative Deepening

Adding different PPs can produce state spaces of different sizes; Landslide tries them in parallel.
Iterative Deepening

If time allows, Landslide will combine PPs into larger, more comprehensive state spaces.
Demo
Test Suite

6 of the official P2 tests are “Landslide-friendly”:

- thr_exit_join
- paraguay
- rwlock_downgrade_read_test
- broadcast_test
- paradise_lost
- mutex_test
What makes a Landslide-friendly test

“Why not largetest, juggle, cyclone, agility_drill...?”

Sample code from cyclone:

```c
for (i = 0; i < MAX_MISBEHAVE /* 64 */; i++) {
    misbehave(i);
    tid = thr_create(child_fn, i);
    thr_join(tid, &status);
}
```
What makes a Landslide-friendly test

“Why not large test, juggle, cyclone, agility_drill...?”

Sample code from cyclone:

```c
for (i = 0; i < MAX_MISBEHAVE /* 64 */; i++) {
    misbehave(i);
    tid = thr_create(child_fn, i);
    thr_join(tid, &status);
}
```

Stress tests expose various interleavings using big loops/many threads;

Landslide finds many interleavings itself; even if the test case has no loops.

How long do you think Landslide would take to test cyclone..?
Previous Semesters

S’15–S’18: 103 groups signed up to use Landslide; 75 found bugs

115 deterministic bugs (e.g. swexn, initialization)
  ▶ 87 (76%) of these were fixed before submitting P2
    ▶ (as verified by running Landslide again – not a guarantee!)

122 distinct non-deterministic bugs
  ▶ 80 (66%) of these were fixed before submitting P2
    ▶ (verified by running Landslide again, same)

75 groups in total found any bugs
  ▶ 58 (77%) groups fixed at least 1 bug
  ▶ 35 (47%) groups fixed all found bugs
  ▶ Most ambitious group: 11 bugs found & fixed!
Survey Results

F’17, S’18: Asked students what they thought of it all
▶ (also ran similar study at Penn State & U. of Chicago)

Easier to diagnose than with stress testing?

<table>
<thead>
<tr>
<th></th>
<th>UC</th>
<th>PSU</th>
<th>CMU</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>N</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>SA</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>not sure/depends</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Survey Results

F’17, S’18: Asked students what they thought of it all
▶ (also ran similar study at Penn State U. & U. of Chicago)
Survey Results

F’17, S’18: Asked students what they thought of it all
▶ (also ran similar study at Penn State U. & U. of Chicago)

![Recommend to a friend? chart]

- UC
- PSU
- CMU

# students

<table>
<thead>
<tr>
<th></th>
<th>SD</th>
<th>D</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Try Landslide on your P2!

Bare minimum effort: No more than 1 hour

- Clone a github URL, run setup script, run tests, answer survey
- Landslide will automatically report test results (as described below)

Full study plan: 4-8 hours of active attention

- (Estimated, including time to diagnose and fix bugs)
- However, many tests should run passively overnight – start soon!
Try Landslide on your P2!

Bare minimum effort: No more than 1 hour
- Clone a github URL, run setup script, run tests, answer survey
- Landslide will automatically report test results (as described below)

Full study plan: 4-8 hours of active attention
- (Estimated, including time to diagnose and fix bugs)
- However, many tests should run passively overnight – start soon!

Prerequisites
- You *must* pass the P2 hurdle before using Landslide.
  - startle, agility_drill, cyclone, join_specific_test, thr_exit_join
- Recommended to attempt several stress tests, e.g.:
  - juggle 4 3 2 0, multitest, racer, paraguay
Try Landslide on your P2!

Interested?

To participate...

▶ Meet prerequisites of passing P2 tests
▶ Read & follow user guide instructions
  ▶ (this will also be emailed to you)
▶ Optional “Landslide clinic” for in-person tech (or moral) support
  ▶ Next week, room and time TBD
Questions?
Coping with State Space Explosion

Serious problem: State spaces grow exponentially

- With $p$ preemption points and $k$ runnable threads, size $p^k$.
- Threatens our ability to explore everything.
- Fortunately, some sequences result in identical states.

Partial Order Reduction identifies and skips “equivalent” interleavings.

- After each execution, compare memory reads/writes of each thread.
- Find when reordering threads couldn’t possibly change behaviour.
- Example follows...
State Space Reduction

**Thread 1** | **Thread 2**
---|---
x = 5 | y = 5
x++; | y--;
x = 6 | y = 4

**Thread 1** | **Thread 2**
---|---
x = 5 | y = 5
x++; | y--;
x = 6 | y = 4
State Space Reduction

Thread 1 | Thread 2
---|---
x = 5 | y = 5
x++; | y--;
x = 6 | y = 4

Thread 1 | Thread 2
---|---
x = 5 | y = 5
x++; | y--;
x = 6 | y = 4
State Space Reduction

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 5$</td>
<td>$y = 5$</td>
</tr>
<tr>
<td>$x++;$</td>
<td>$y--;$</td>
</tr>
<tr>
<td>$x = 6$</td>
<td>$y = 4$</td>
</tr>
</tbody>
</table>

Avoided exploring a subtree!
More Survey Results

(To dispel any accusations that I might have omitted survey questions with less good results)

How many bugs did Landslide help you find?

- UC
- PSU
- CMU

# students

0  2  4  6  8  10

# bugs found

0  1  2  3  4  5  6
More Survey Results

(To dispel any accusations that I might have omitted survey questions with less good results)
More Survey Results

(To dispel any accusations that I might have omitted survey questions with less good results)
More Survey Results

(To dispel any accusations that I might have omitted survey questions with less good results)
More Survey Results

(To dispel any accusations that I might have omitted survey questions with less good results)

![Debugging output easy to understand?](chart.png)
More Survey Results

(To dispel any accusations that I might have omitted survey questions with less good results)
More Survey Results

(To dispel any accusations that I might have omitted survey questions with less good results)