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

more clever than "`mandelbrot`" *since 2011.*

Ben Blum (`bblum@andrew.cmu.edu`)

Carnegie Mellon University - 15-410

2015, February 11

Outline

Theory: Seeing race conditions 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
- User study (that's you!)

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)

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

Thread Interleavings (different "good" case)

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

Thread Interleavings (race condition)

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 ☹`		

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

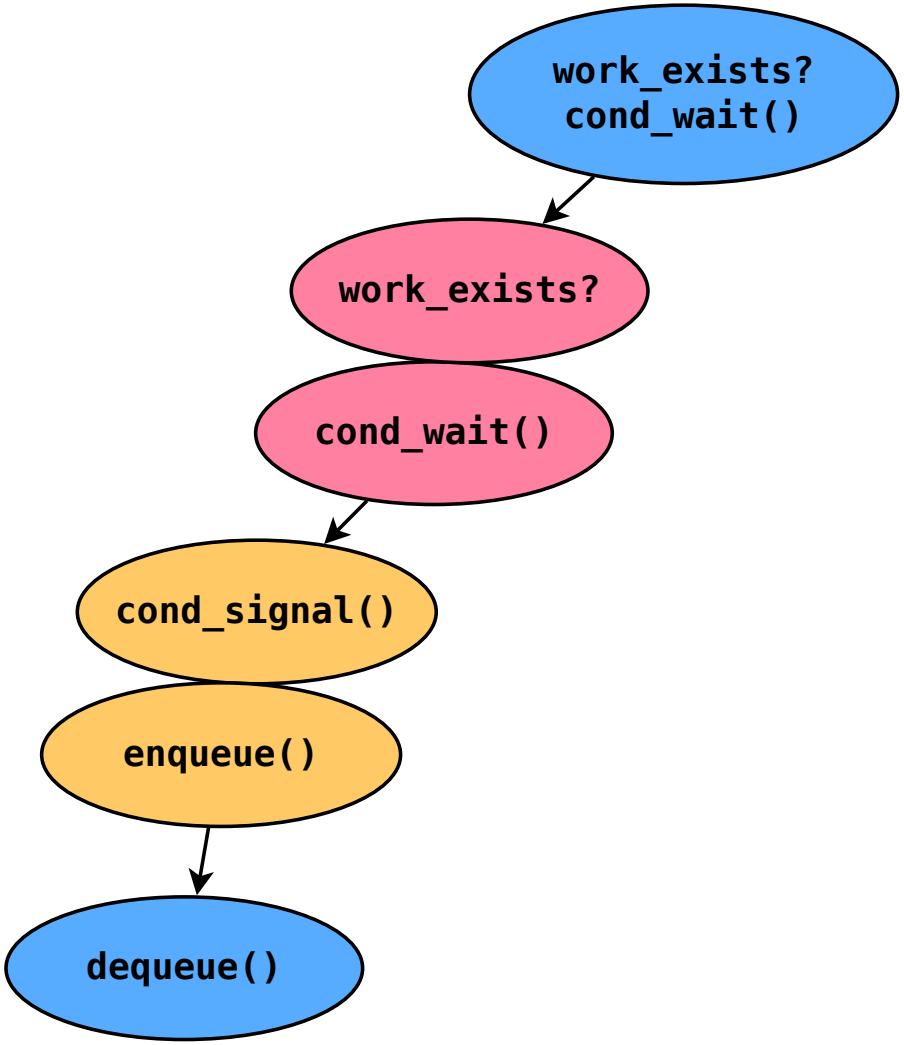
- Attempting to exercise as many interleavings as practical
- Exposes race conditions at random
  - "If a preemption occurs at just the right time. . . "
- Cryptic panic messages when failure occurs

What if. . .

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

A different way of looking at race conditions. . .

Execution Tree

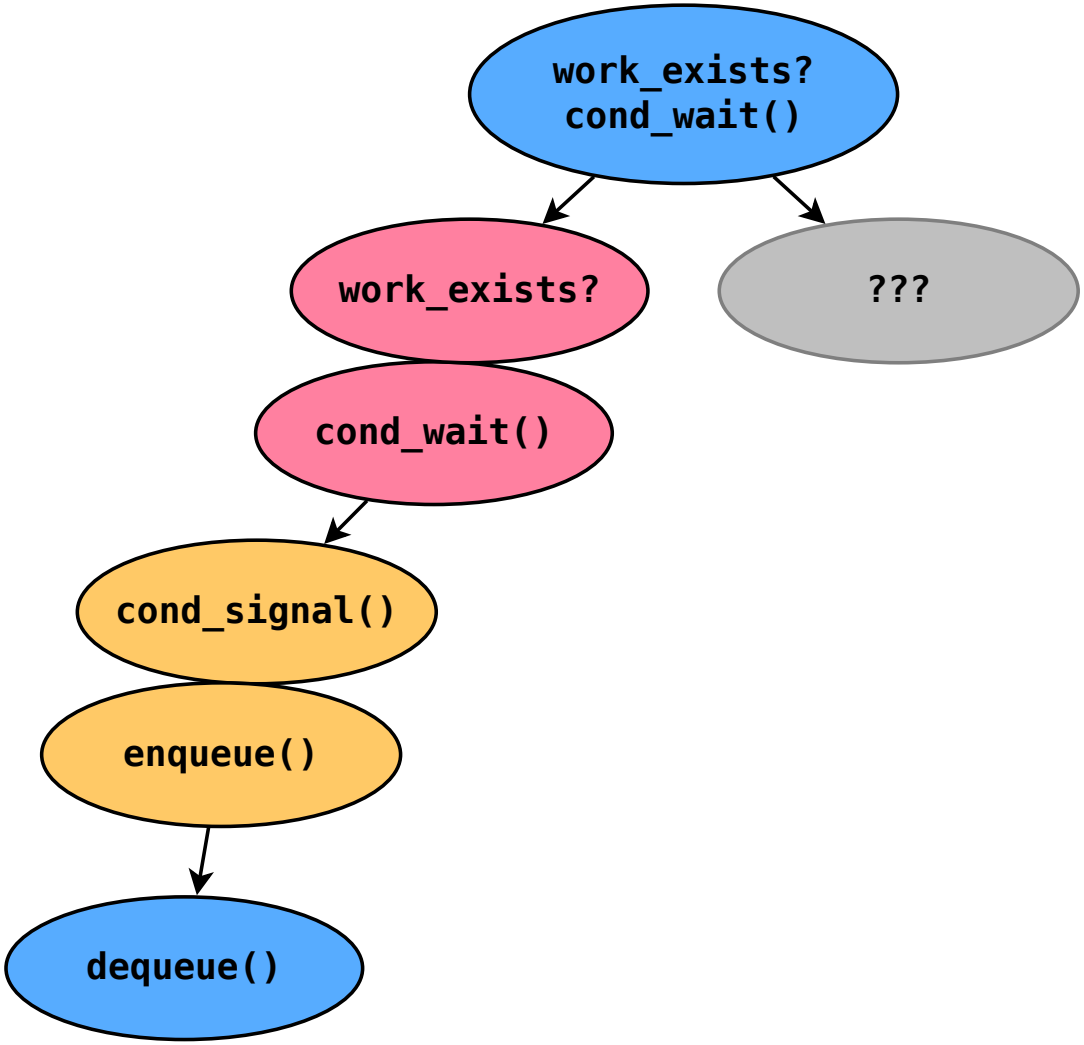


Thread 1	Thread 2	Thread 3

`lock(mx);`
`if (!work_exists())`
`  wait(cvar, mx);`
 	
`work = dequeue();`
`unlock(mx);`
`access(work->data);`
	
	`lock(mx);`
	`enqueue(work);`
	`signal(cvar);`
	`unlock(mx);`
		
		`lock(mx);`
		`if (!work_exists())`
		`  wait(cvar, mx);`

work != NULL
(no bug)

Execution Tree



Thread 1	Thread 2	Thread 3

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

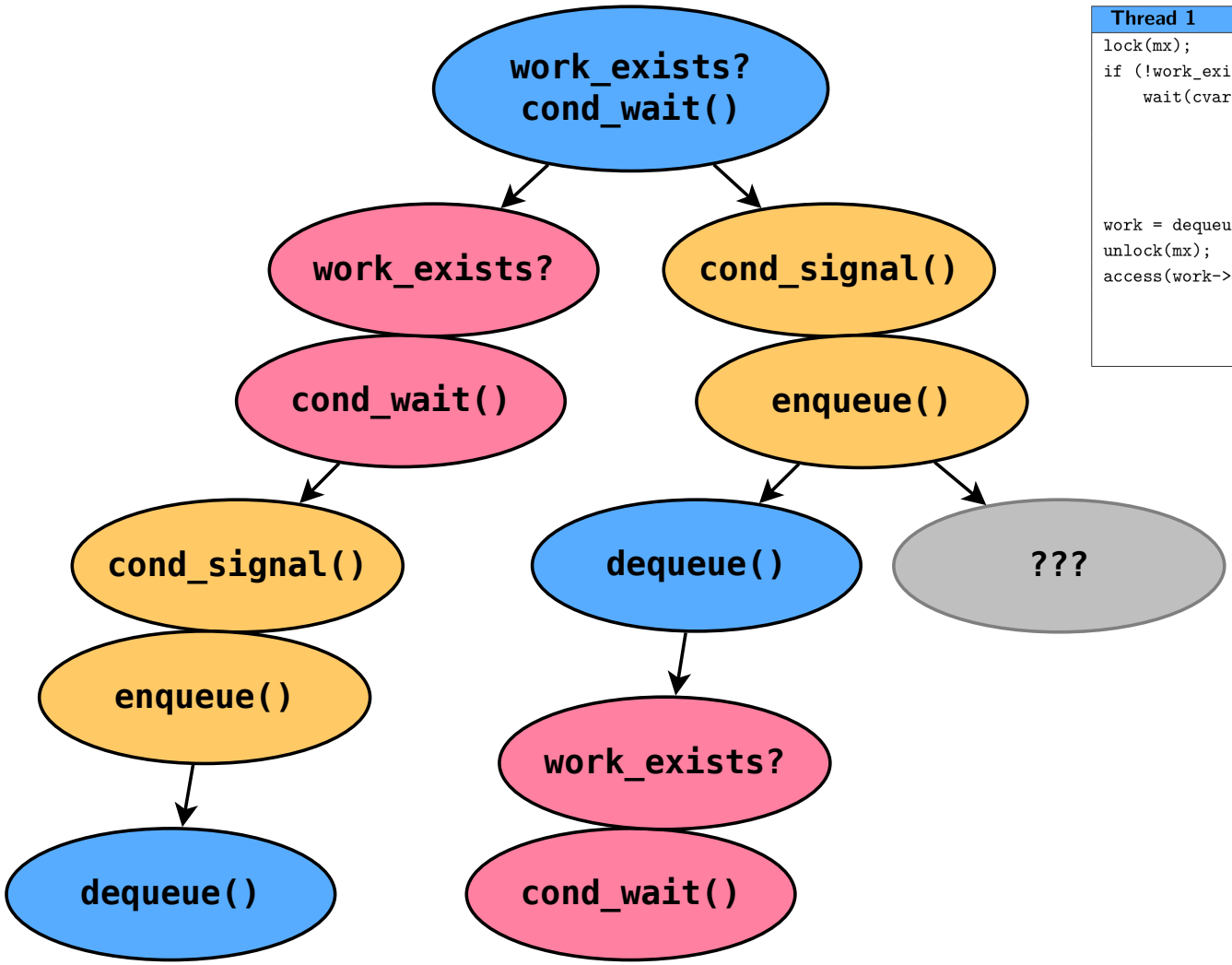
		`lock(mx);`
		`if (!work_exists())`
		`  wait(cvar, mx);`

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

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

work != NULL
(no bug)

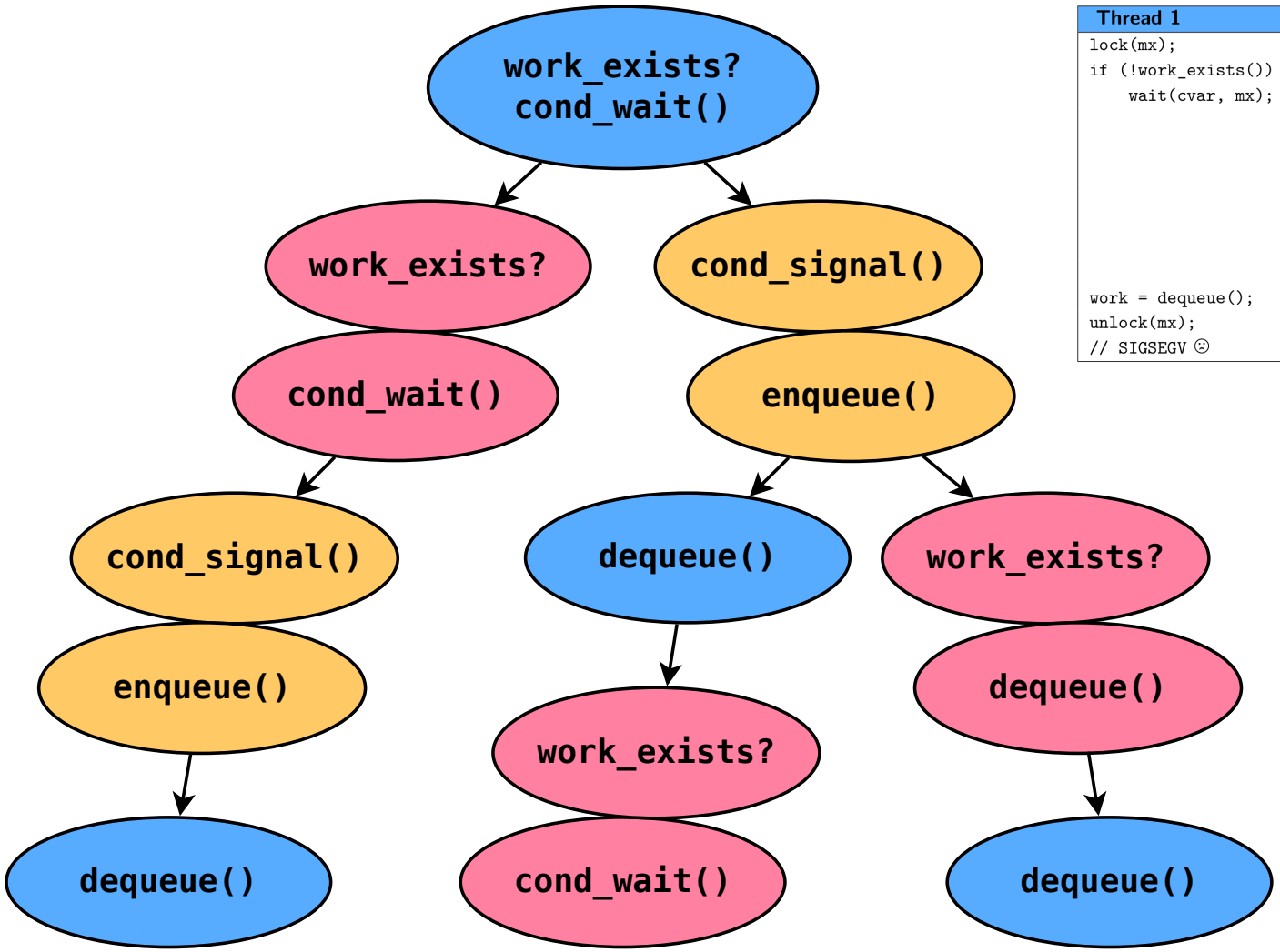
Execution Tree



Thread 1	Thread 2	Thread 3

lock(mx);
if (!work_exists())
  wait(cvar, mx);
 	
work = dequeue();
unlock(mx);
access(work->data);
	
	lock(mx);
	enqueue(work);
	signal(cvar);
	unlock(mx);
		
		lock(mx);
		if (!work_exists())
		  wait(cvar, mx);

Execution Tree



Thread 1	Thread 2	Thread 3

`lock(mx);`
`if (!work_exists())`
`  wait(cvar, mx);`
 	
`work = dequeue();`
`unlock(mx);`
`// SIGSEGV ☹`
	
	`lock(mx);`
	`enqueue(work);`
	`signal(cvar);`
	`unlock(mx);`
		
		`lock(mx);`
		`work = dequeue();`
		`unlock(mx);`

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 add preemptions to produce different thread interleavings.
- Intuitively: Generate many "tabular execution traces".

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 add preemptions to produce different thread interleavings.
- 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 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.

- If there are n PPs and k threads, state space size is n^k.
- Need to choose the set of 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.

Is there a **"sweet spot"**?

Preemption Points

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

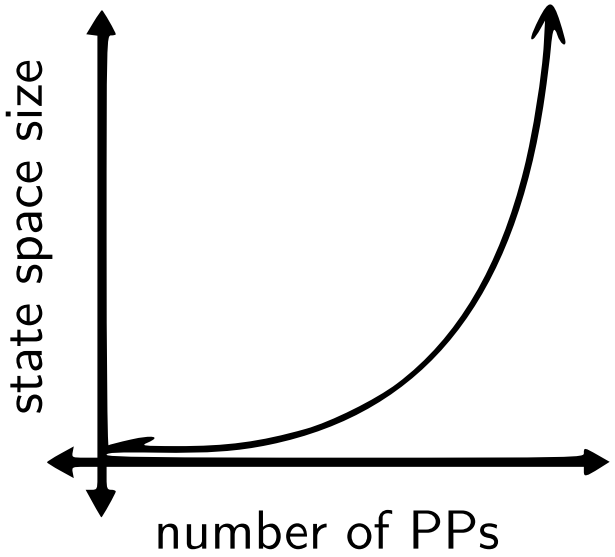
When do we fear being preempted?

- 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

Challenges

Parameters of systematic tests must be kept small.

- Test program length / number of threads
- Number of preemption points used



Landslide

About The Project

About me: 4th year graduate student, advised by Garth Gibson

- TAed 15-410 for 3 semesters during undergrad
- 1st graduate year was 5th year MS, rest in Ph.D. program
  - `http://www.contrib.andrew.cmu.edu/~bblum/thesis.pdf`

About The Project

About me: 4th year graduate student, advised by Garth Gibson

- TAed 15-410 for 3 semesters during undergrad
- 1st graduate year was 5th year MS, rest in Ph.D. program
  - `http://www.contrib.andrew.cmu.edu/~bblum/thesis.pdf`

About Landslide

- Simics module, which traces:
  - Every instruction executed
  - Every memory access read/written
- Originally supported only P3s; can now test P2s fully-automated
- *Landslide* shows how your *Pebbles* programs may not be stable.

Big Picture: Execution Tree Exploration

Backtracking

- At end of each branch, identify a PP to replay differently
- Reset machine state and start over
- Implemented using Simics bookmarks
  - `set-bookmark` and `skip-to`
- Replay test from the beginning, with a different interleaving

Big Picture: Execution Tree Exploration

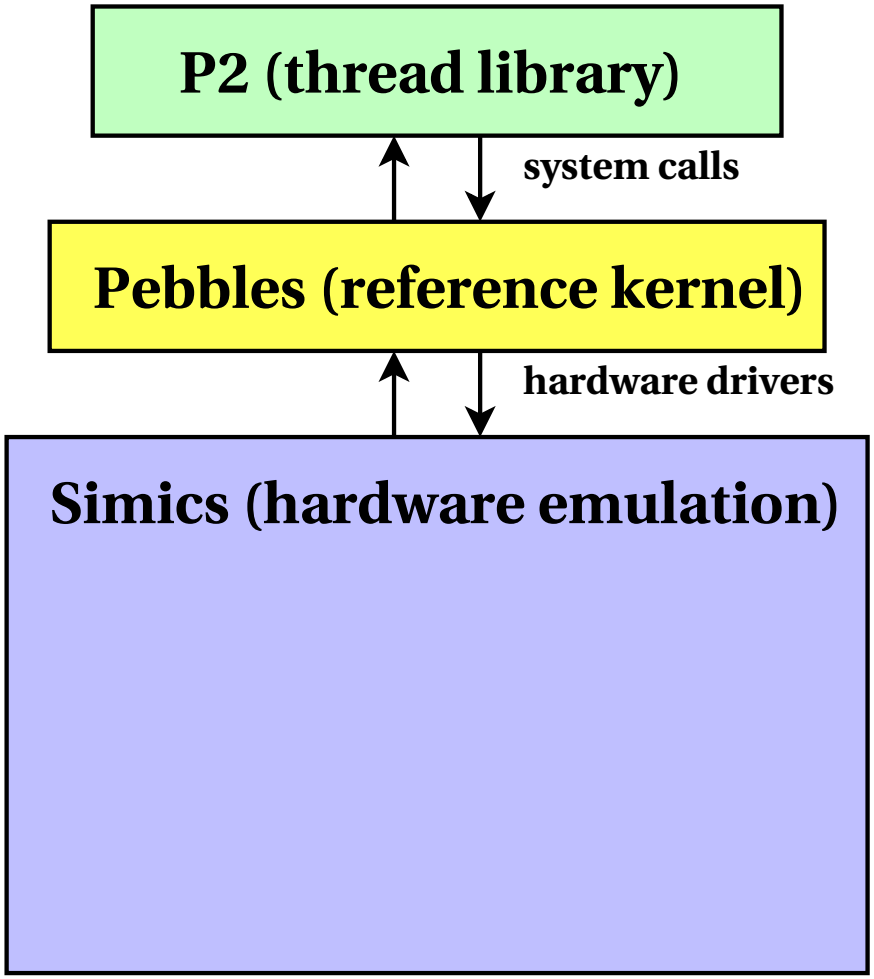
Backtracking

- At end of each branch, identify a PP to replay differently
- Reset machine state and start over
- Implemented using Simics bookmarks
  - `set-bookmark` and `skip-to`
- Replay test from the beginning, with a different interleaving

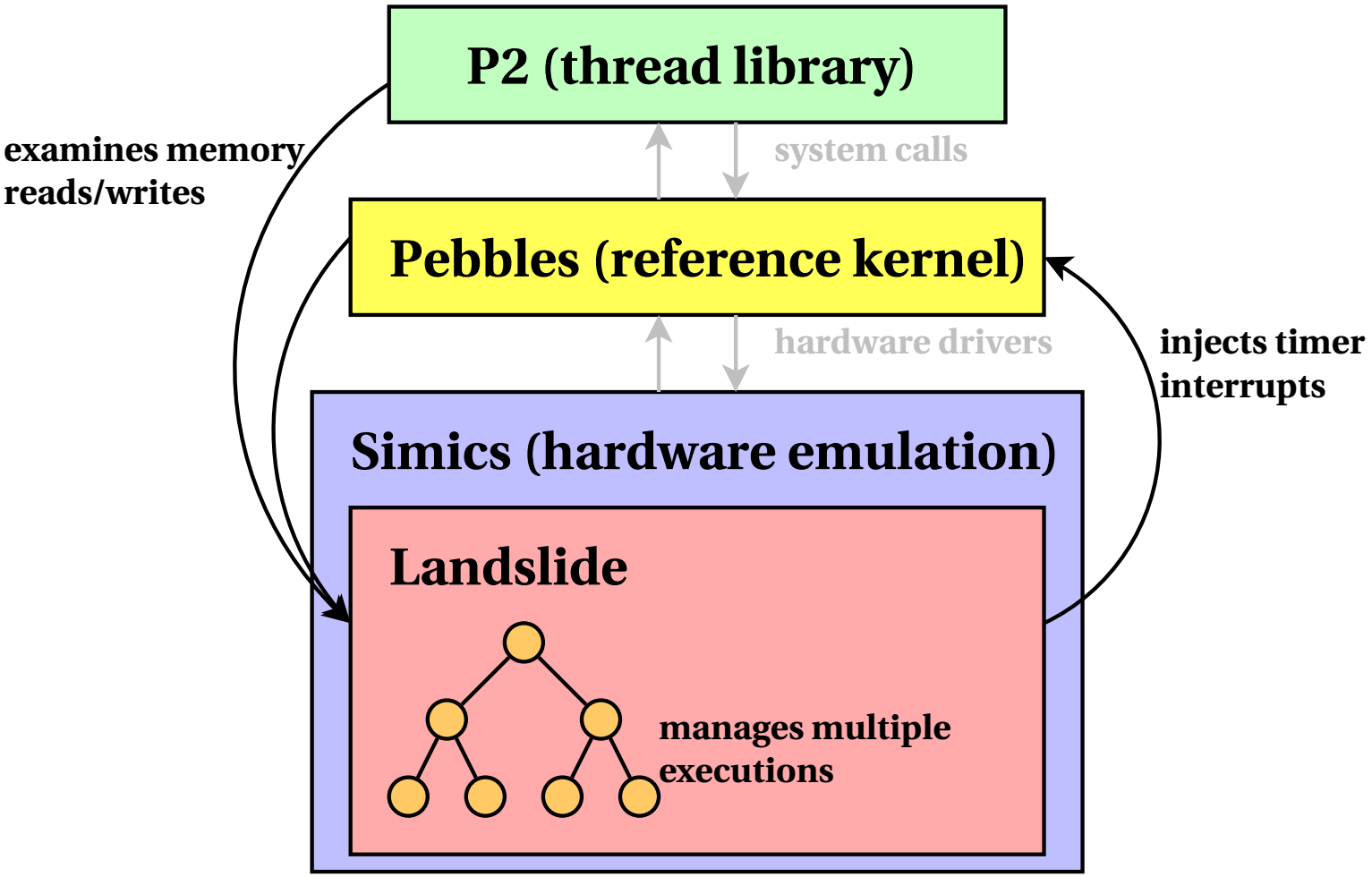
Controlling scheduling decisions

- 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



Landslide & You



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 algorithm: Is `number[i]=max(number[0],number[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 hard-code a fixed set of preemption points.

- E.g., "all thread library 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 hard-code a fixed set of preemption points.

- E.g., "all thread library 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 of Preemption Points

Goal: Run the best tests for a given CPU budget.

- Technique: "Iterative Deepening"

Based on experience from past 15-410 student volunteers

- Students worked best with an iterative process
- "Start small, then add more preemption points as time allows"
- Landslide now automates this process

Named after analogous technique in chess AI.

- Chess search is DFS limited by max number of moves (ply).
- Chess AIs repeat DFS, increasing ply, until timeout.

Iterative Deepening in Landslide

Landslide automatically iterates through different configurations of PPs.

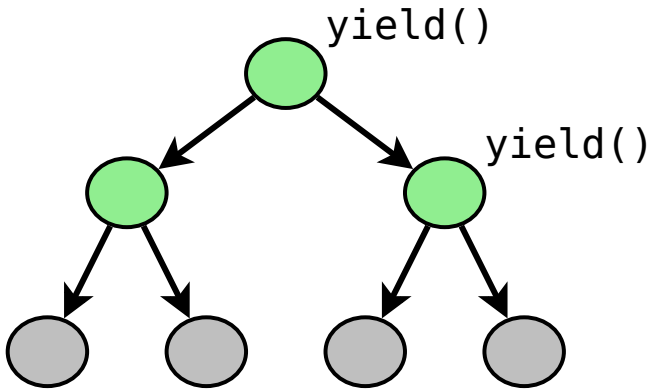
- Manages work queue of jobs with different PPs
- Each job represents a new state space for Landslide to explore
- Prioritizes which jobs are important / likely to finish
  - Based on nature of PPs (data races? mutexes?)
  - Based on estimated completion time
- Repeat state space explorations, adding preemption points, until time is exhausted.

Only required argument is CPU budget

Iterative Deepening

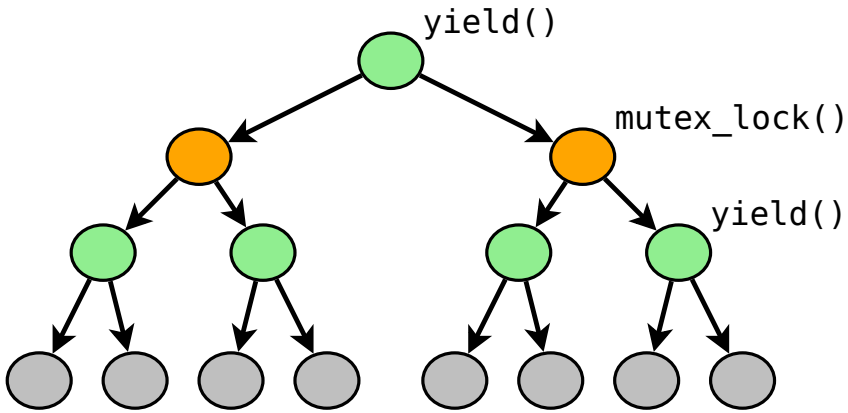
Minimal state space includes only "mandatory" context switches

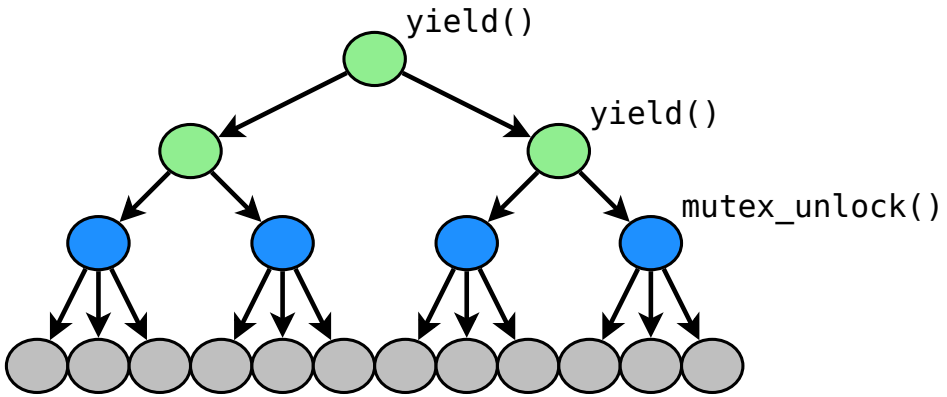
- e.g., `yield()`, `cond_wait()`.



Iterative Deepening

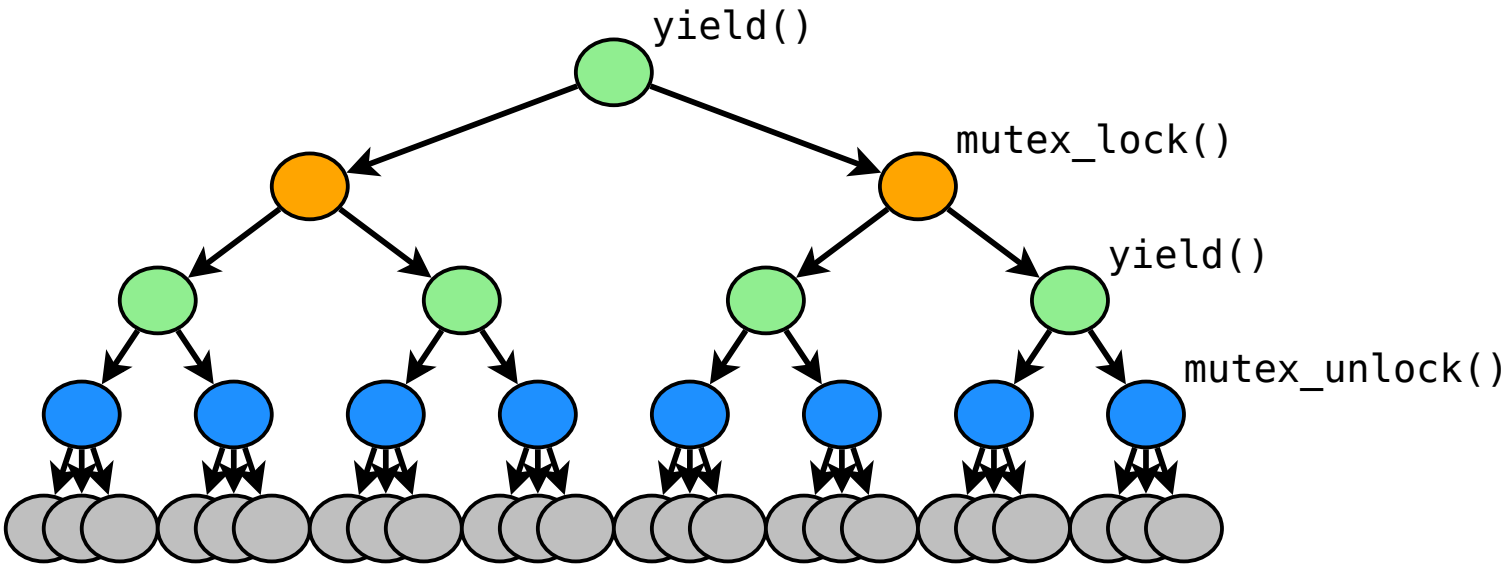
Adding different PPs can produce state spaces of different sizes; Landslide tries them in parallel.





Iterative Deepening

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



Demo

Preliminary Evaluation

We tested 22 random submitted P2s from Spring 2014.

- Test cases:
  - `thr_exit_join`
  - `paraguay`
  - `rwlock_downgrade_read_test`
  - `broadcast_test` (new)
- Each test given CPU budget of 11 cores $\times$ 10 minutes.

Preliminary Results

88 total tests were run (9,680 total CPU-minutes)

Found 7 **deterministic** initialization bugs (e.g. use-after-free)

56 tests ran to completion without finding bugs:

- 47 exhausted the CPU budget.
- 9 completed all state spaces before time ran out.

Between 20 and 33 **non-deterministic** bugs were found:

- 20 deadlock races among 12 P2s
- 10 use-after-free races among 6 P2s
- 3 misc (assert fail, segfault) among 2 P2s

User Study

Try Landslide on your P2!

- Bare minimum effort: No more than 1 hour
  - Clone a github URL, run setup script, run, answer survey questions
  - Landslide will automatically report test results
- 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!

User Study

Try Landslide on your P2!

- Bare minimum effort: No more than 1 hour
  - Clone a github URL, run setup script, run, answer survey questions
  - Landslide will automatically report test results
- 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`
- Must have *attempted* several stress tests
  - `juggle 4 3 2 0`, `multitest`, `racer` (15 min), `paraguay`

User Study - Additional Information

Human Subjects Research

- CMU IRB has approved this study
- Landslide will collect results while you use it
  - Record commands issued, take snapshots of your P2 code
  - All data will be anonymized before publication
- No coercion: ***There is no penalty for not participating.***
  - I am not on course staff, cannot influence your grade
  - Course staff will not have access to study data during semester

User Study - Additional Information

Human Subjects Research

- CMU IRB has approved this study
- Landslide will collect results while you use it
  - Record commands issued, take snapshots of your P2 code
  - All data will be anonymized before publication
- No coercion: ***There is no penalty for not participating.***
  - I am not on course staff, cannot influence your grade
  - Course staff will not have access to study data during semester

Risks & Benefits

- Benefit: Landslide may help you find/fix bugs, improving your grade!
- Risk: Landslide may find no bugs and be a waste of your time.
- Benefit: You might learn something...

User Study - How to Participate

Interested?

To participate...

- Review this lecture and study information sheet
- Meet prerequisites of passing P2 tests
- Email me (`bblum@cs.cmu.edu`) for instructions, survey
- Use Landslide in addition to stress tests until P2 is due!

Watch your email for more details!

Questions?

More Preliminary Results

Is dynamically adding "data race" PPs effective?

- 3 among reported bugs (9-15%) required data races to expose.

Is "iterative deepening" better than state-of-the-art?

- Control experiment: Just 1 state space, same CPU time
  - PPs used: `mutex_lock()`, `mutex_unlock()`
  - 110 minutes on 1 CPU
- Of 33 total bug reports, control failed to find 10 (30%).
  - 3 required data race PPs to expose
  - 1 ran out of time
  - 6 obscured by different bug in same state space

Coping with State Space Explosion

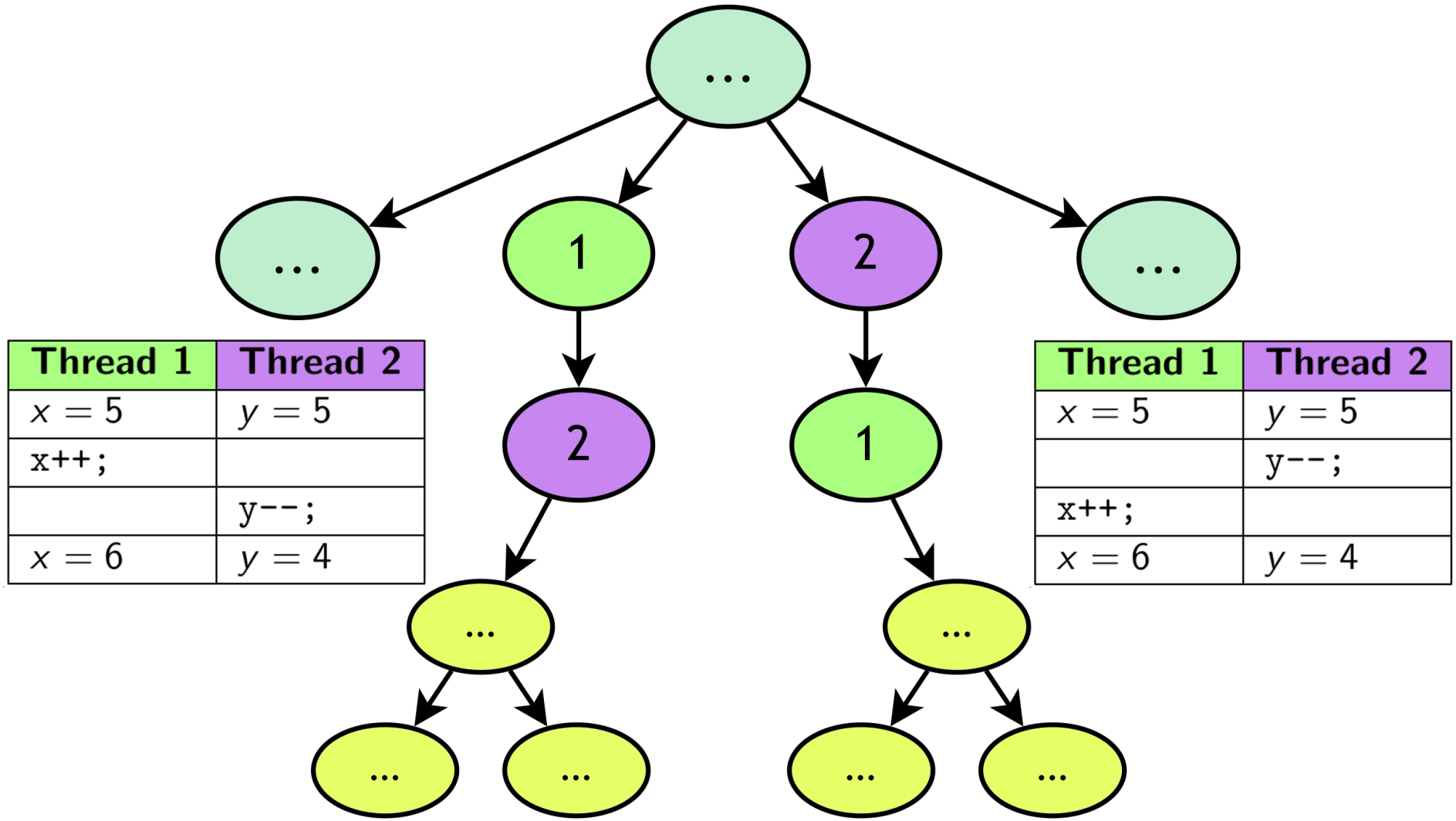
Serious problem: State spaces grow exponentially

- 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...

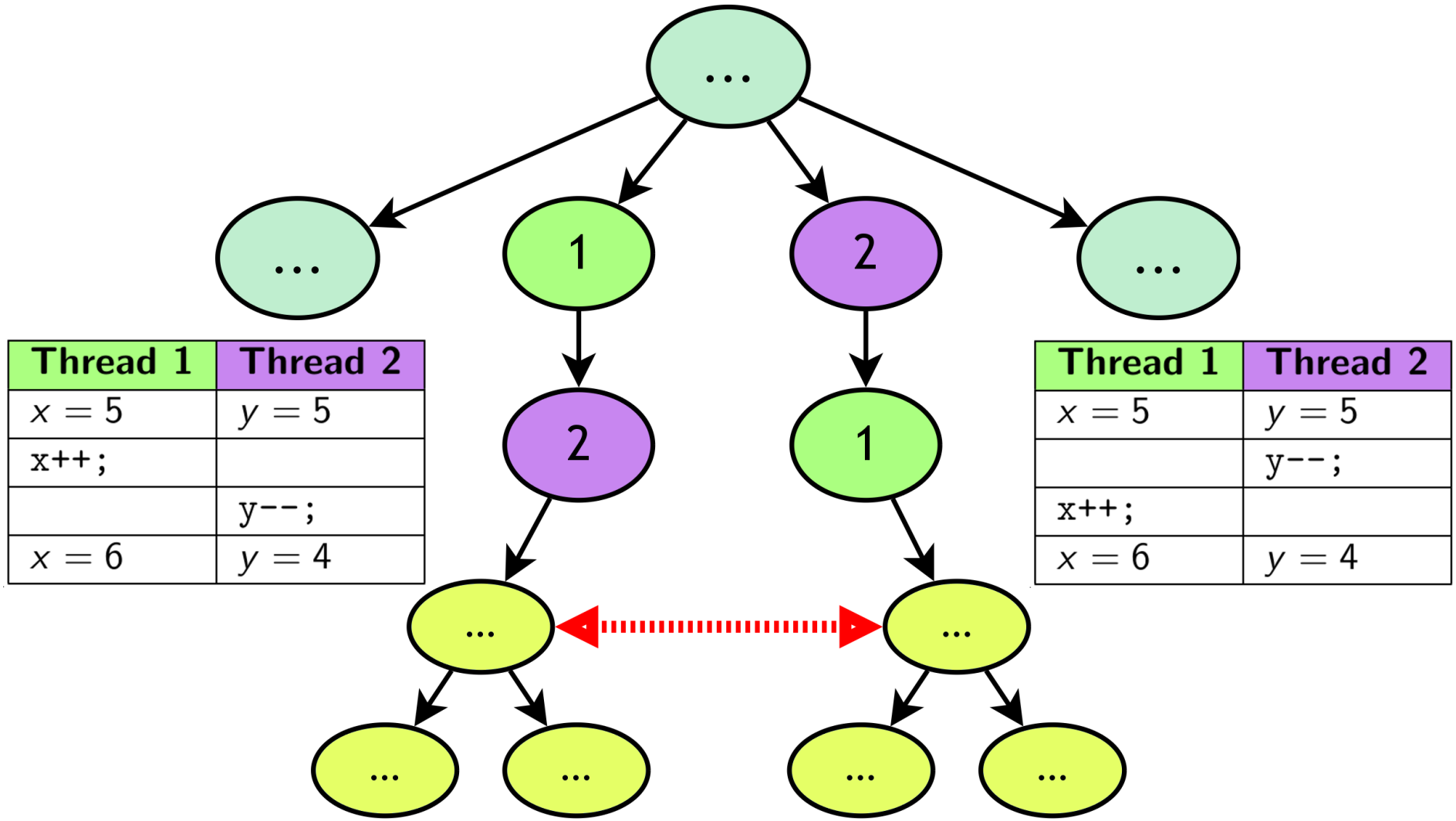
State Space Reduction



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

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

State Space Reduction

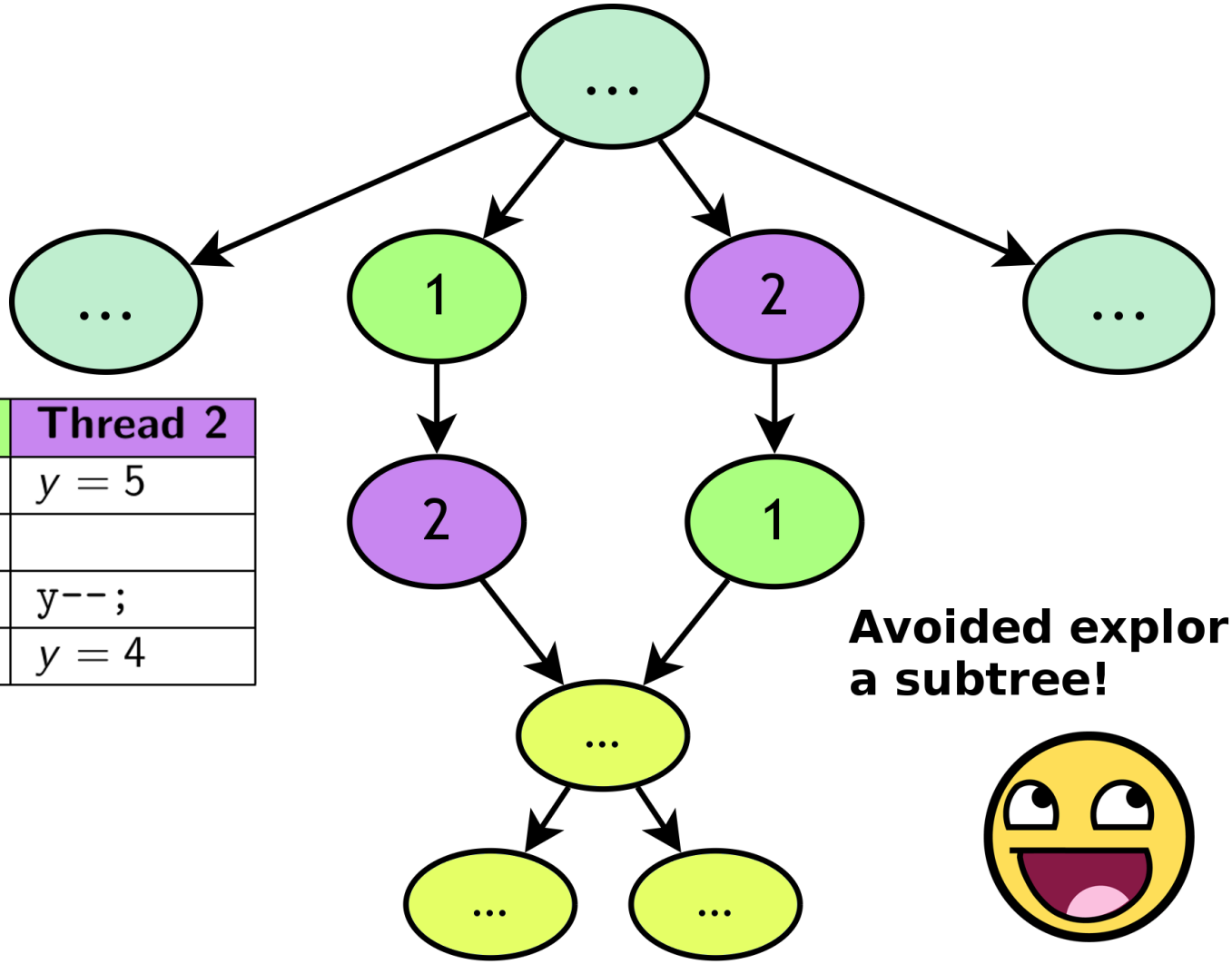


Thread 1	Thread 2
$x = 5$	$y = 5$
x++;	
	y--;
$x = 6$	$y = 4$

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

State Space Reduction

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



Avoided exploring a subtree!