Stateless Model Checking with Data-Race Preemption Points

Ben Blum, Garth Gibson

Carnegie Mellon University
Pittsburgh, PA, USA

2016, November 03

Background: Stateless Model Checking

Example

Initially `x = 0;`

Thread 1	**Thread 2**
`mutex_lock(m);`	`atomic_xadd(&x, 1);`
`x++;`	`yield();`
`mutex_unlock(m);`	`atomic_xadd(&x, 1);`
`assert(x >= 1);`	`assert(x >= 2);`

Example

Initially `x = 0;`

Thread 1	**Thread 2**
`mutex_lock(m);`	
`int tmp = x;`	
	`atomic_xadd(&x, 1);`
	`yield();`
	`atomic_xadd(&x, 1);`
`x = tmp + 1;`	
	`assert(x >= 2);`
`mutex_unlock(m);`	
`assert(x >= 1);`	

Stateless Model Checking

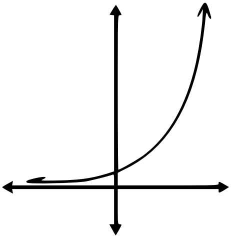
Stateless Model Checking *[Godefroid '97]*

- Dynamic concurrency testing technique
- Test framework controls thread scheduling
- Each test iteration, test a different interleaving
- Goal: Exhaustively check all possible program behaviours

Stateless Model Checking

Stateless Model Checking *[Godefroid '97]*

- Dynamic concurrency testing technique
- Test framework controls thread scheduling
- Each test iteration, test a different interleaving
- Goal: Exhaustively check all possible program behaviours



Stateless Model Checking

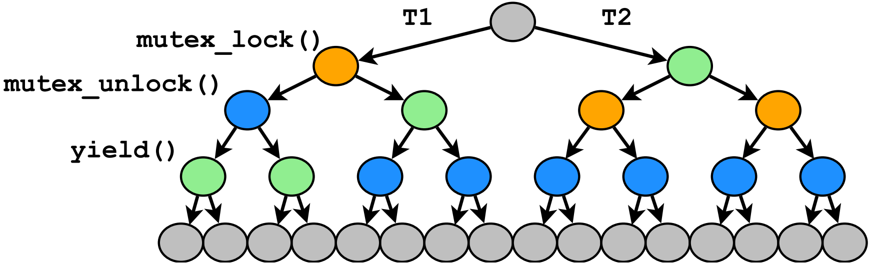
Stateless Model Checking *[Godefroid '97]*

- Dynamic concurrency testing technique
- Test framework controls thread scheduling
- Each test iteration, test a different interleaving
- Goal: Exhaustively check all possible program behaviours

Wait... *all possible* thread interleavings?

- Total verification may be feasible for "small" tests
- "Large" tests may have exponentially many possible schedules
- Completion depends on size of *state space*...

State Space Example



Possible interleavings represented as a tree

- Node: Intermediate execution state
- Edge: State transition from executing a thread

Preemption Points

The burning question: "Which **preemption points** (PPs) are important?"

State space of interleavings is *parameterized* by PPs.

- Too few PPs: Many bugs will go undetected
- Preempt everywhere: Completion is often infeasible

Prior model checkers hard-code a fixed set of PPs, committing to one state space in advance.

Coping techniques

Dynamic Partial Order Reduction (DPOR) *[Flanagan '05]*

- Identifies independent (commuting) transitions to prune

Iterative Context Bounding (ICB) *[Musuvathi '08]*

- Heuristically orders search to prioritize fewer preemptions

State space estimation *[Simsa '12]*

- Guesses completion time in advance based on existing progress

Challenge

Given fixed PPs, resulting state space can be inappropriate

- Choosing PPs statically makes tests infeasible, insufficient, or both!
- Even small code changes can have drastic impact

Problem: Make trade-off decision automatically.

Background: Data-Race Analysis

Data-Race Analysis

Data race: 2 threads access the same memory, and...

- At least one access is a write
- Threads do not hold the same lock
- No *Happens-Before* relation between threads

Data-Race Analysis

Data race: 2 threads access the same memory, and...

- At least one access is a write
- Threads do not hold the same lock
- No *Happens-Before* relation between threads

Variants of Happens-Before (HB)

- **Pure HB:** Any synchronization events *[Lamport '78]*
- **Limited HB:** Blocking synchronization (e.g. `cond_wait()`) enforces one ordering *[O'Callahan '03]*

Happens-Before Example

Thread 1	**Thread 2**
`x++;`	
`mutex_lock(m);`	
`// unrelated`	
`mutex_unlock(m);`	
	`mutex_lock(m);`
	`// unrelated`
	`mutex_unlock(m);`
	`x++;`

No race under Pure HB; *true potential race* under Limited HB.

Happens-Before Example

Thread 1	Thread 2
`x++;`	
`mutex_lock(m);`	
`y = true;`	
`mutex_unlock(m);`	
	`mutex_lock(m);`
	`bool tmp = y;`
	`mutex_unlock(m);`
	`if (tmp) x++;`

No race under Pure HB; *false positive* under Limited HB.

Races, Bugs, What's the Difference?

Not all data races lead to failures.

- C++ spec: All data races are undefined behaviour.
- Many prior tools: User attention is valuable, report only *failing* races. *[Engler '03, Kasicki '12]*

Stateless model checkers search for concrete, observable failures

- Assertion failures, memory errors, infinite loops

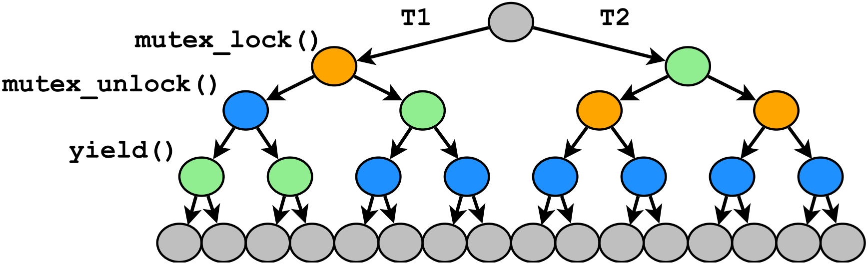
Problem: Classify data races as *failing* or *benign*.

Example (again)

Initially `x = 0;`

Thread 1	**Thread 2**
`mutex_lock(m);`	
`int tmp = x;`	
	`atomic_xadd(&x, 1);`
	`yield();`
	`atomic_xadd(&x, 1);`
`x = tmp + 1;`	
	`assert(x >= 2);`
`mutex_unlock(m);`	
`assert(x >= 1);`	

State Space (again)



None of these branches contain the necessary preemption!

A **data-race preemption point** is required to find the bug.

Iterative Deepening and Quicksand

Contribution

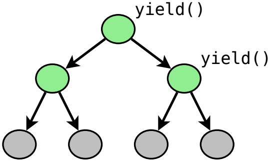
Algorithm: Iterative Deepening

- Seed model checker with synchronization API PPs;
- Dynamically detect new data-race candidates;
- Add data-race PPs as discovered;
- Iteratively advance to state spaces with new PPs;
- Prioritize them with dynamic state space estimates;
- ...until specified CPU budget is exhausted.

Iterative Deepening

"Minimal" state space: mandatory thread switches only

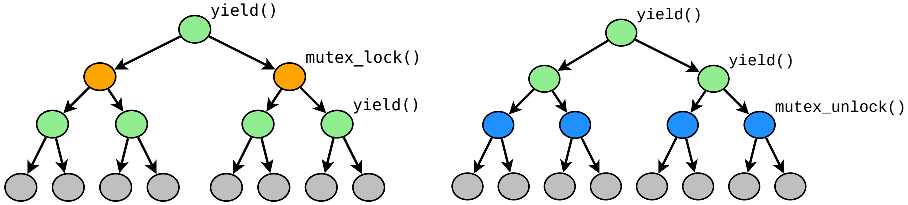
- `yield()`, `cond_wait()`, etc.



Iterative Deepening

Different PPs can produce state spaces of different sizes.

Testing them in parallel hedges our bets.

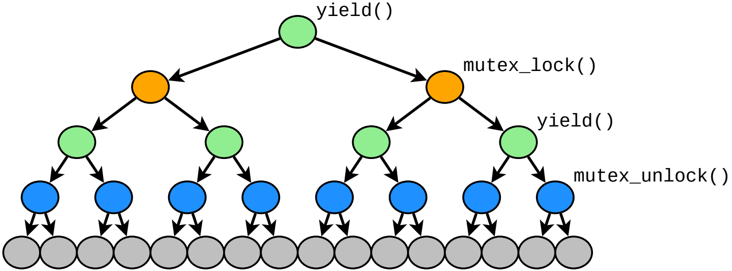


Iterative Deepening

If time allows, we combine PPs to produce larger tests.

All PPs enabled = "maximal" state space

- Prior work tools explore this state space only.



Implementation

Landslide *[Blum '12]*, our simulator-based model checker

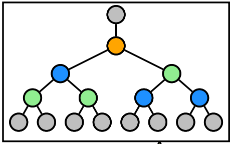
- Targets Pebbles thread libraries (CMU) and Pintos kernels (Berkeley, U. of Chicago, etc.)
- Wind River Simics provides instruction/memory-level tracing
- Features DPOR, estimation, ICB, and data-race detection

Quicksand, our Iterative Deepening implementation

- Manages queue of *jobs* with different PP combinations
- A separate Landslide instance tests each job
- Prioritizes jobs based on state space estimates

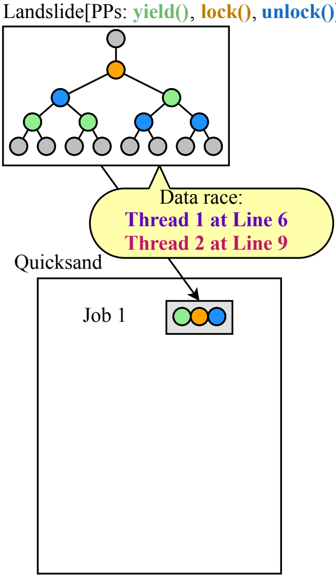
System Architecture

Landslide[PPs: **yield()**, **lock()**, **unlock()**]

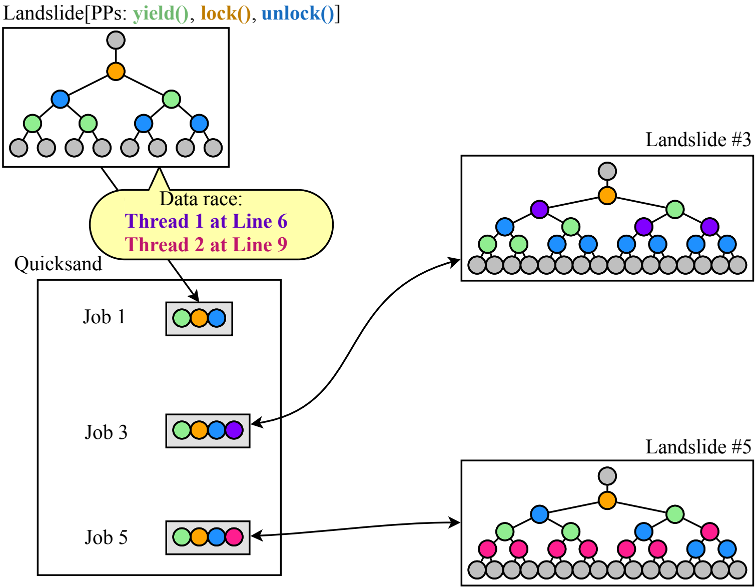


Data race:
Thread 1 at Line 6
Thread 2 at Line 9

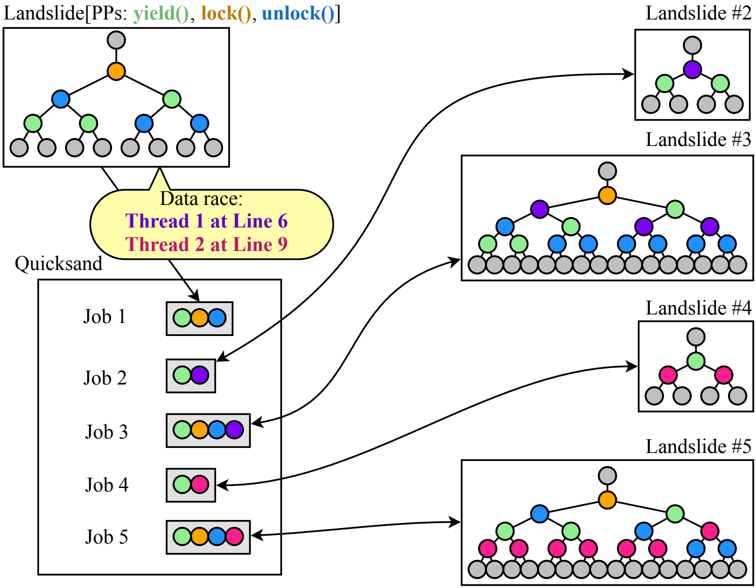
System Architecture



System Architecture

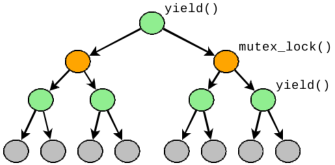


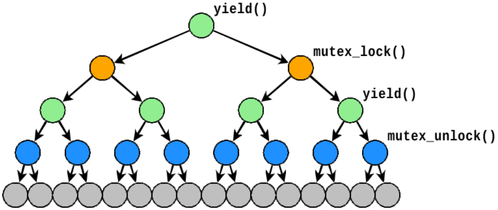
System Architecture



Iterative Deepening Reductions

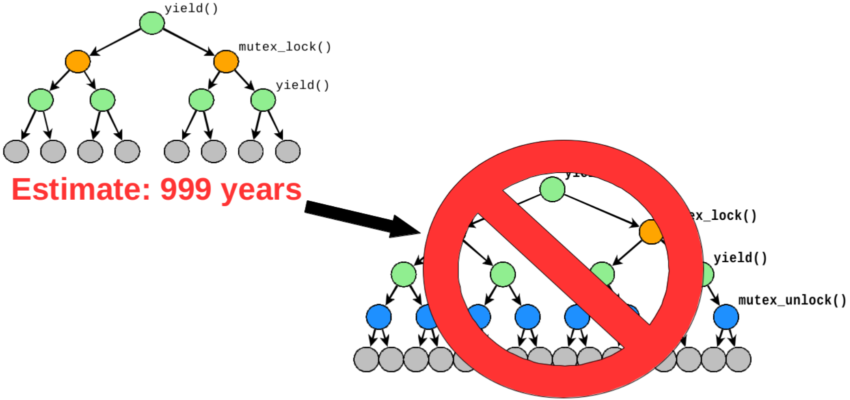
If one job is a subset of another, testing either might let us skip the other.





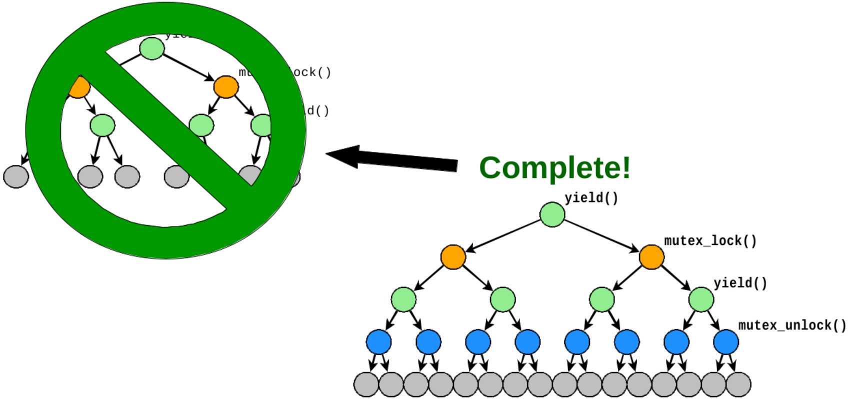
Iterative Deepening Reductions

If one job is a subset of another, testing either might let us skip the other.



Iterative Deepening Reductions

If one job is a subset of another, testing either might let us skip the other.



Total Verification

Convergence theorem: Testing maximal state space, after detecting all data races, $\equiv$ preempting everywhere!

- Insight: $\forall$ arbitrary buggy interleaving, $\exists$ equivalent interleaving using only sync/data-race PPs.

Hence, Iterative Deepening provides "best of both worlds":

- Provides total safety guarantee for small tests
- Finds bugs quickly for large tests when completion is infeasible

Evaluation

Evaluation Questions

Comparing to single-state-space (SSS) testing

- Do we find more bugs given fixed CPU budgets?
- Do we provide more total verifications?
- ...whether or not SSS chooses PPs on synchronization only, or "everywhere"?

Experimental Setup

Testing OS projects from CMU, Berkeley, and U. Chicago

- CMU: 79 “Pebbles” thread libraries ($\times$ 6 test cases each)
- Berkeley & U. Chicago: 78 “Pintos” kernels ($\times$ 3 test cases each)
- 629 unique tests in total

Experimental Setup

Testing OS projects from CMU, Berkeley, and U. Chicago

- CMU: 79 “Pebbles” thread libraries ($\times$ 6 test cases each)
- Berkeley & U. Chicago: 78 “Pintos” kernels ($\times$ 3 test cases each)
- 629 unique tests in total

Experimental trials:

- Quicksand with Limited HB (10 CPUs $\times$ 1 hour)
- Quicksand with Pure HB
- SSS Landslide with ICB, mutex/yield PPs only (1 CPU $\times$ 10 hours)
- SSS Landslide with ICB and “Preempt Everywhere” strategy

Experimental Setup

Testing OS projects from CMU, Berkeley, and U. Chicago

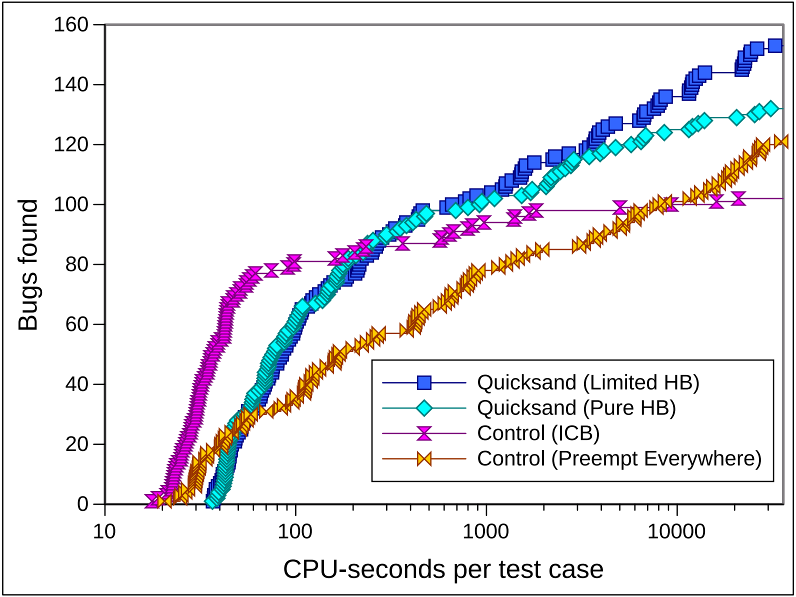
- CMU: 79 “Pebbles” thread libraries ($\times$ 6 test cases each)
- Berkeley & U. Chicago: 78 “Pintos” kernels ($\times$ 3 test cases each)
- 629 unique tests in total

Experimental trials:

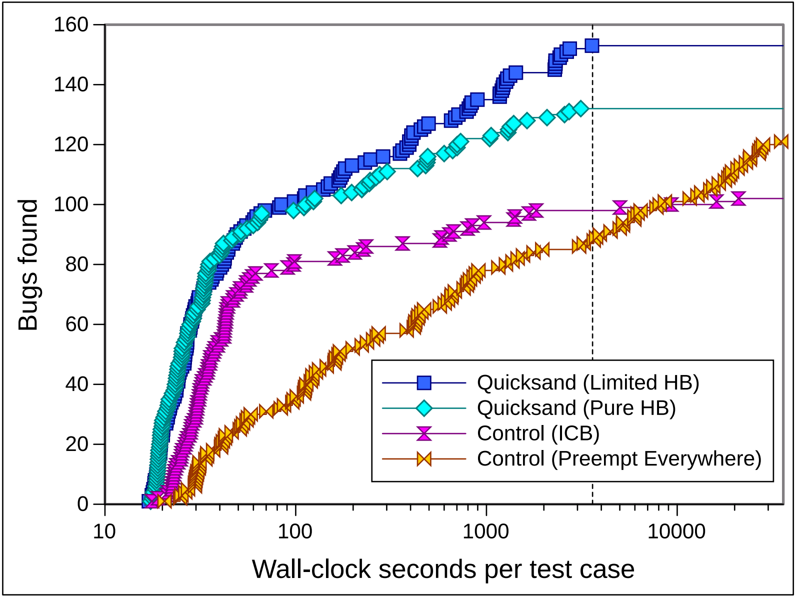
- Quicksand with Limited HB (10 CPUs $\times$ 1 hour)
- Quicksand with Pure HB
- SSS Landslide with ICB, mutex/yield PPs only (1 CPU $\times$ 10 hours)
- SSS Landslide with ICB and “Preempt Everywhere” strategy

629 tests $\times$ 10 CPU-hours $\times$ 4 trials $\approx$ 1000 CPU-days

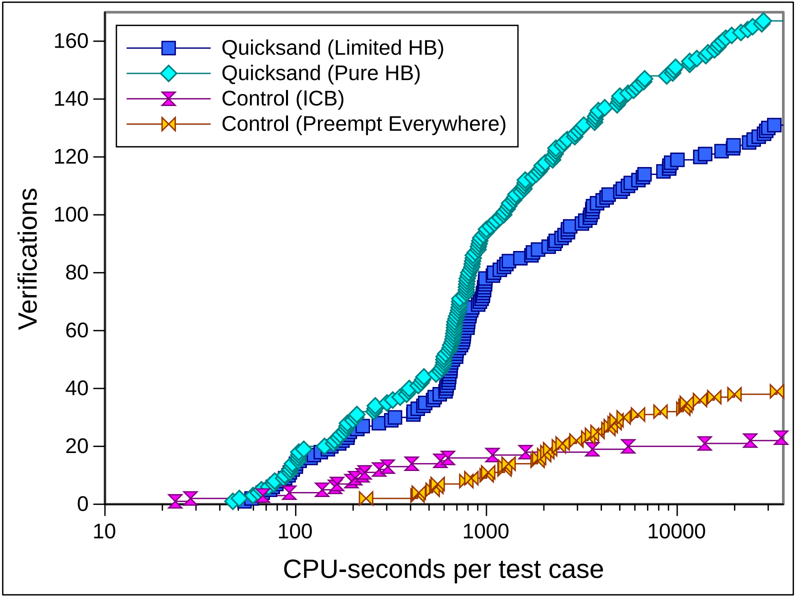
Results (Bug-finding, CPU time)



Results (Bug-finding, wall-clock time)



Results (Verification, CPU time)



Takeaway

Quicksand outperforms SSS after 10 CPU-hours in all cases.

- 108%-125% as many bugs found
- 336%-428% as many verifications provided

“Preempt Everywhere” finds bugs best for SSS, but overhead of PPs significantly impacts completion time.

Pure HB provides more verifications; when possible;
Limited HB finds more bugs, at expense of completion time.

Thanks & Questions

Bonus Slides

More Statistics about our Test Suite

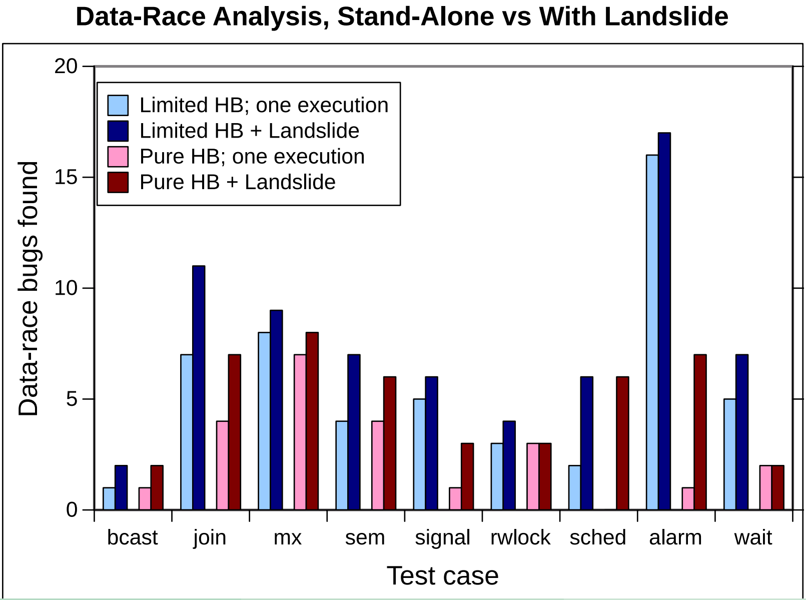
Bugs by required number of preemptions to expose:

Bound	SSS (sync PPs only)	SSS (preempt everywhere)
0	2	1
1	82	86
2	16	32
3	2	3
4+	0	0
Total	102	122

Bugs by type (Pure HB trial):

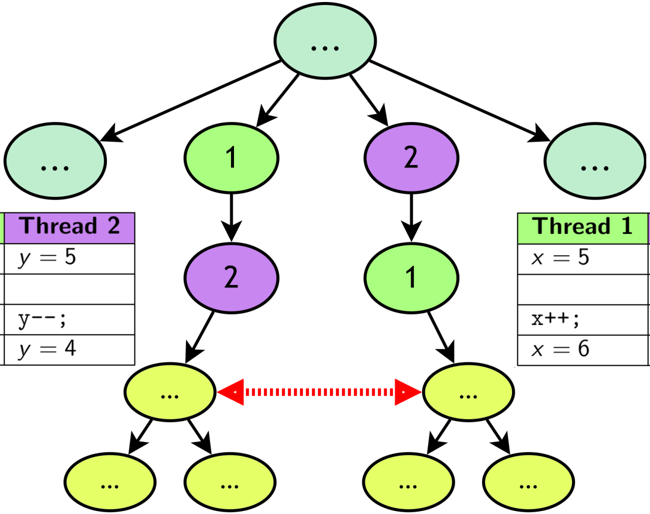
- 56 deadlocks
- 49 heap errors (47 use-after-free)
- 35 assertion failures
- 31 page fault crashes
- 1 infinite loop
- 1 recursive mutex lock

Results ("Nondeterministic" Data Races)



DPOR Example

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



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

References

- **[Lamport '78]**: Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 1978.
- **[Godefroid '97]**: Patrice Godefroid. VeriSoft. A tool for the automatic analysis of concurrent reactive software. CAV 1997.
- **[Engler '03]**: Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. SOSP 2003.
- **[O'Callahan '03]**: Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. PPoPP '03.
- **[Flanagan '05]**: Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. POPL 2005.
- **[Musuvathi '08]**: Madanlal Musuvathi et al. Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI 2008.
- **[Simsa '12]**: Jiri Simsa. Runtime Estimation and Resource Allocation for Concurrency Testing. CMU-PDL-12-113. December 2012.
- **[Blum '12]**: Ben Blum. Landslide: Systematic Dynamic Race Detection in Kernel Space. CMU-CS-12-118. May 2012.
- **[Kasicki '12]**: Baris Kasicki. Data races vs. data race bugs: telling the difference with Portend. ASPLOS '12.
- **[Huang '15]**: Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. PLDI '15.