Finding and Reproducing Heisenbugs in Concurrent Programs

Authors: Madan Musuvathi, Shaz Qadeer, Tom Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu

Presented by: Dingbao Xie

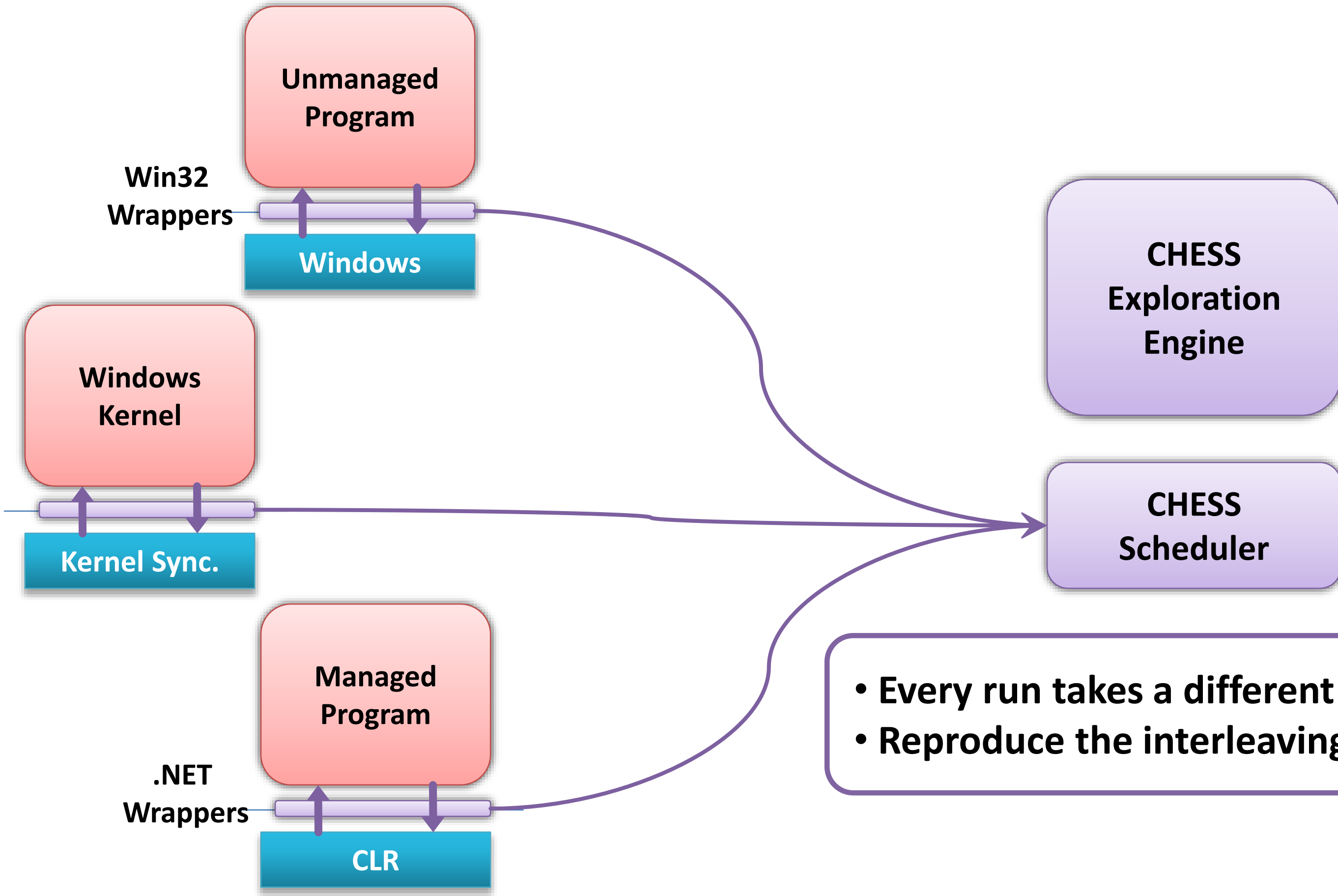
The Heisenbug problem

- Concurrent executions are highly nondeterminisitic
- Rare thread interleavings result in Heisenbugs
  - Difficult to find, reproduce, and debug
- Observing the bug can “fix” it
  - Likelihood of interleavings changes, say, when you add printfs
- A huge productivity problem
  - Developers and testers can spend weeks chasing a single Heisenbug

CHESS in a nutshell

- CHESS is a user-mode scheduler
- Controls all scheduling nondeterminism
  - Replace the OS scheduler
- Guarantees:
  - Every program run takes a different thread interleaving
  - Reproduce the interleaving for every run

CHESS architecture



- Every run takes a different interleaving
- Reproduce the interleaving for every run

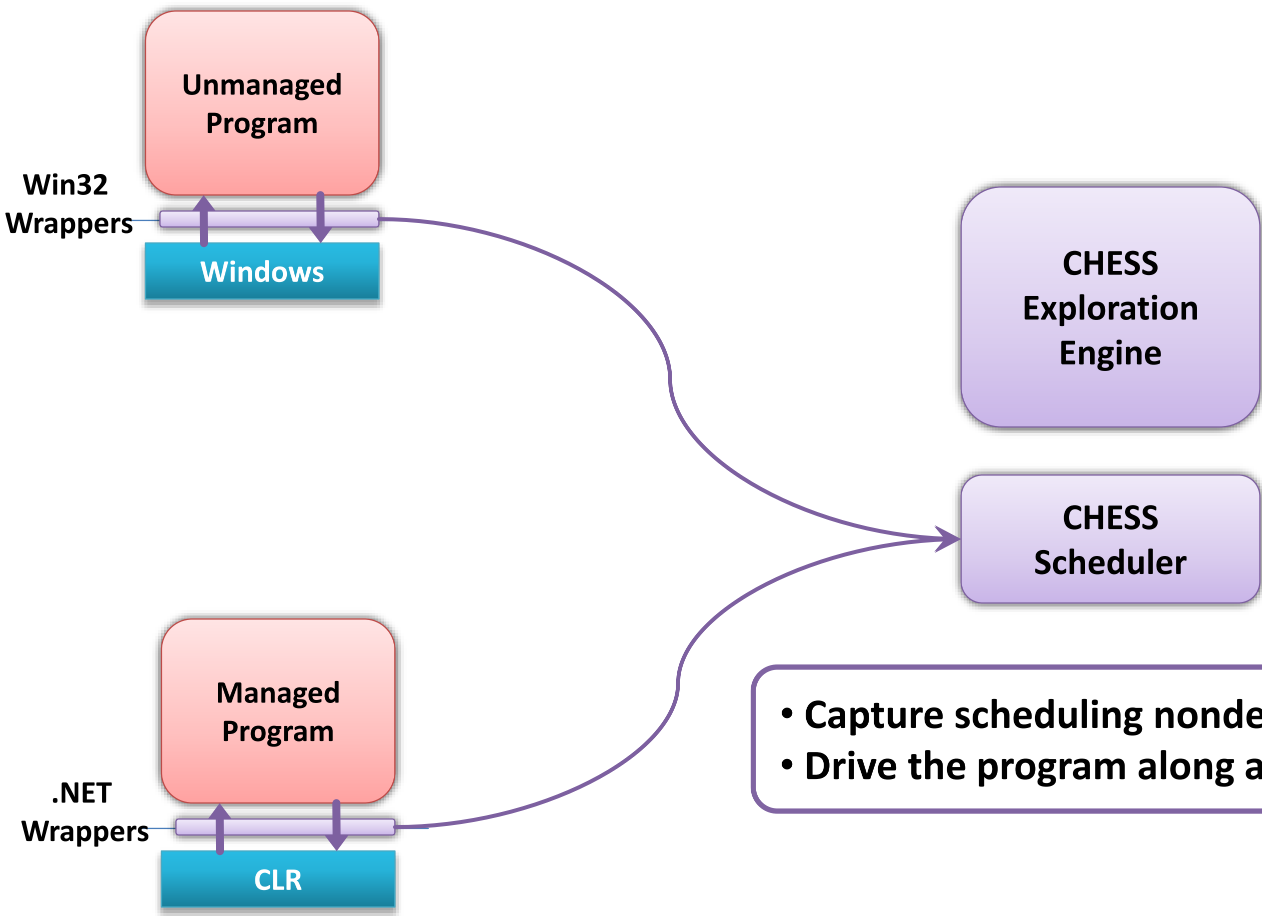
High level goals

- Scale to large programs
- Any error found by CHESS is possible in the wild
  - CHESS does not introduce any new behaviors
- Any error found in the wild can be found by CHESS
  - Need to capture all sources of nondeterminism
  - Exhaustively explore the nondeterminism (state explosion)
    - e.g. Enumerate all thread interleavings
  - Hard to achieve
    - Practical goal: beat stress

Errors that CHESS can find

- Assertions in the code
- Any dynamic monitor that you run
  - Memory leaks, double-free detector, …
- Deadlocks
  - Program enters a state where no thread is enabled
- Livelocks
  - Program runs for a long time without making progress

CHESS architecture



- **Capture scheduling nondeterminism**
- **Drive the program along an interleaving of choice**

Concurrent Executions are Nondeterministic

Thread 1

```
x = 1;
y = 1;
```

Thread 2

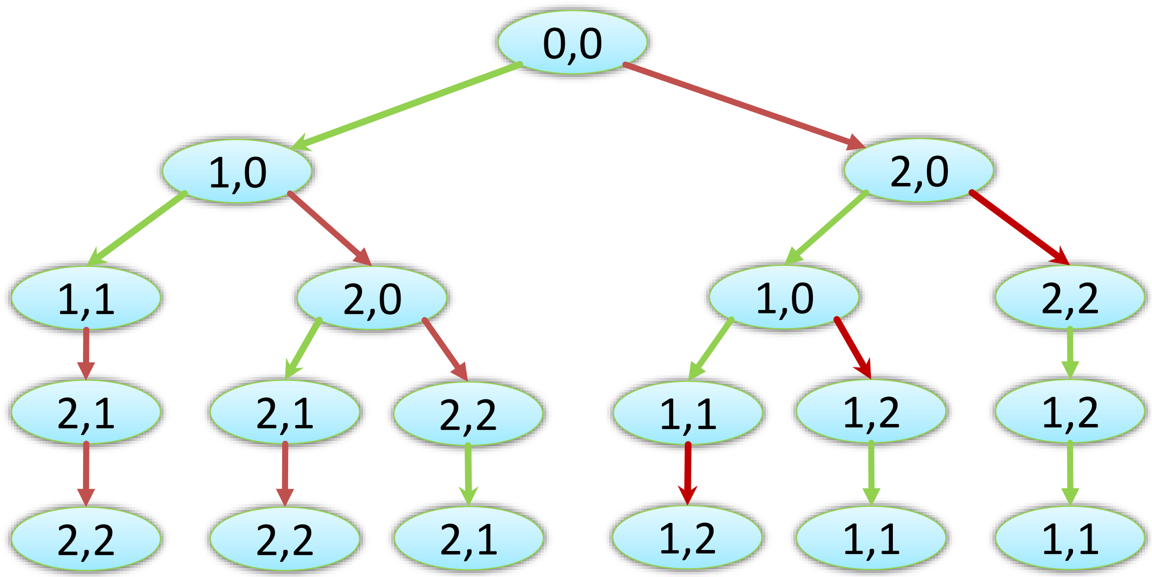
```
x = 2;
y = 2;
```

```
x = 1;
```

```
y = 1;
```

```
x = 2;
```

```
y = 2;
```



Sources of Nondeterminism
1. Scheduling Nondeterminism

- Interleaving nondeterminism
  - Threads can race to access shared variables or monitors
  - OS can preempt threads at arbitrary points
- Timing nondeterminism
  - Timers can fire in different orders
  - Sleeping threads wake up at an arbitrary time in the future
  - Asynchronous calls to the file system complete at an arbitrary time in the future

Sources of Nondeterminism
1. Scheduling Nondeterminism

- Interleaving nondeterminism
  - Threads can race to access shared variables or monitors
  - OS can preempt threads at arbitrary points
- Timing nondeterminism
  - Timers can fire in different orders
  - Sleeping threads wake up at an arbitrary time in the future
  - Asynchronous calls to the file system complete at an arbitrary time in the future
- CHESS captures and explores this nondeterminism

Sources of Nondeterminism
2. Input nondeterminism

- User Inputs
  - User can provide different inputs
  - The program can receive network packets with different contents
- Nondeterministic system calls
  - Calls to gettimeofday(), random()
  - ReadFile can either finish synchronously or asynchronously

Sources of Nondeterminism
2. Input nondeterminism

- User Inputs
  - User can provide different inputs
  - The program can receive network packets with different contents
  - CHESS relies on the user to provide a scenario
- Nondeterministic system calls
  - Calls to gettimeofday(), random()
  - ReadFile can either finish synchronously or asynchronously
  - CHESS provides wrappers for such system calls

Running Example

Thread 1

```
Lock (l);
bal += x;
Unlock(l);
```

Thread 2

```
Lock (l);
t = bal;
Unlock(l);

Lock (l);
bal = t - y;
Unlock(l);
```

Introduce Schedule() points

Thread 1

```
Schedule();
Lock (l);
bal += x;
Schedule();
Unlock(l);
```

Thread 2

```
Schedule();
Lock (l);
t = bal;
Schedule();
Unlock(l);

Schedule();
Lock (l);
bal = t - y;
Schedule();
Unlock(l);
```

- Instrument calls to the CHESS scheduler
- Each call is a potential preemption point

Emulate execution on a uniprocessor

Thread 1 | **Thread 2**

Thread 2:

```
Schedule();
Lock (l);
t = bal;
Schedule();
Unlock(l);
```

Thread 1:

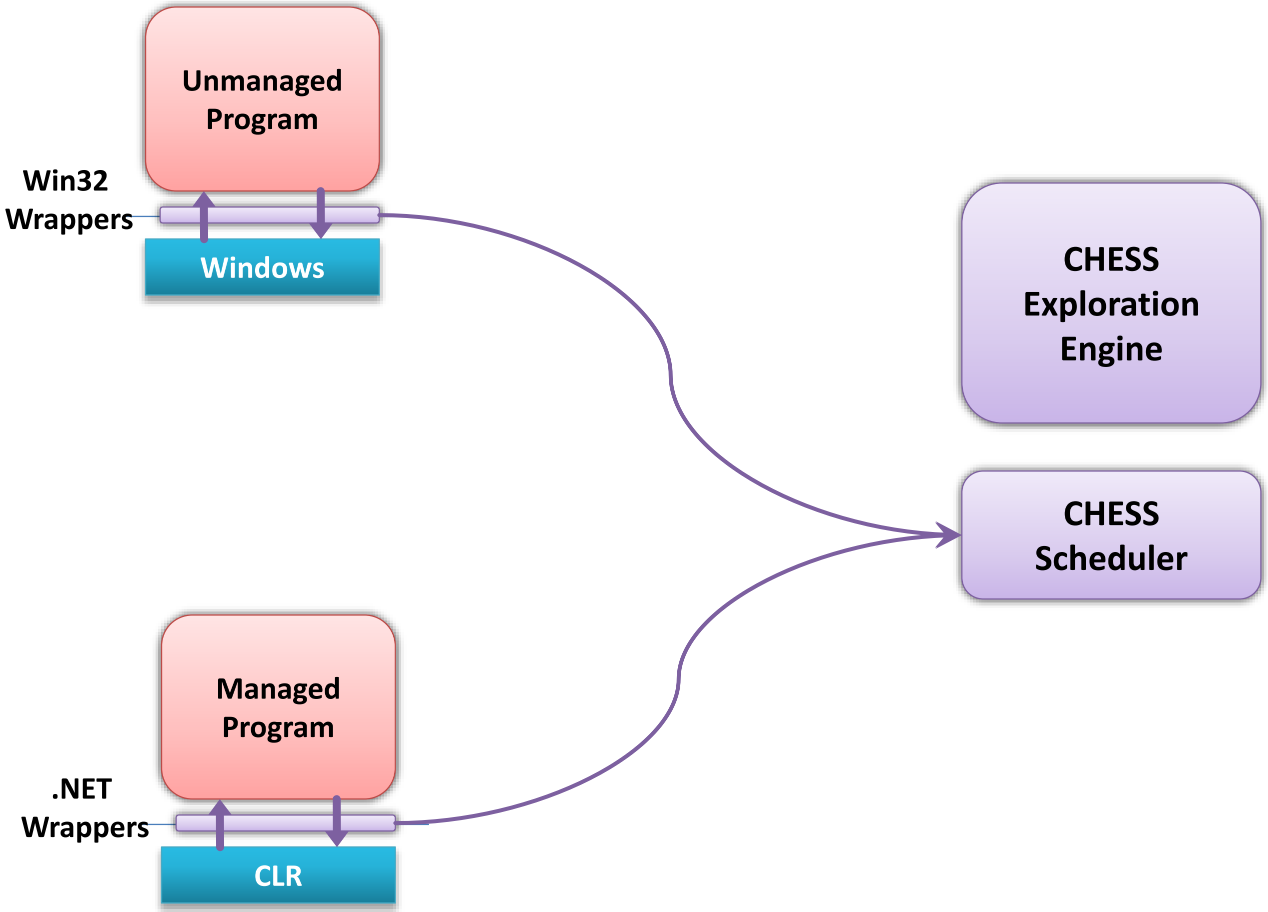
```
Schedule();
Lock (l);
bal += x;
Schedule();
Unlock(l);
```

Thread 2:

```
Schedule();
Lock (l);
bal = t - y;
Schedule();
Unlock(l);
```

- Enable only one thread at a time
- Linearizes a partial-order into a total-order
- Controls the order of data-races

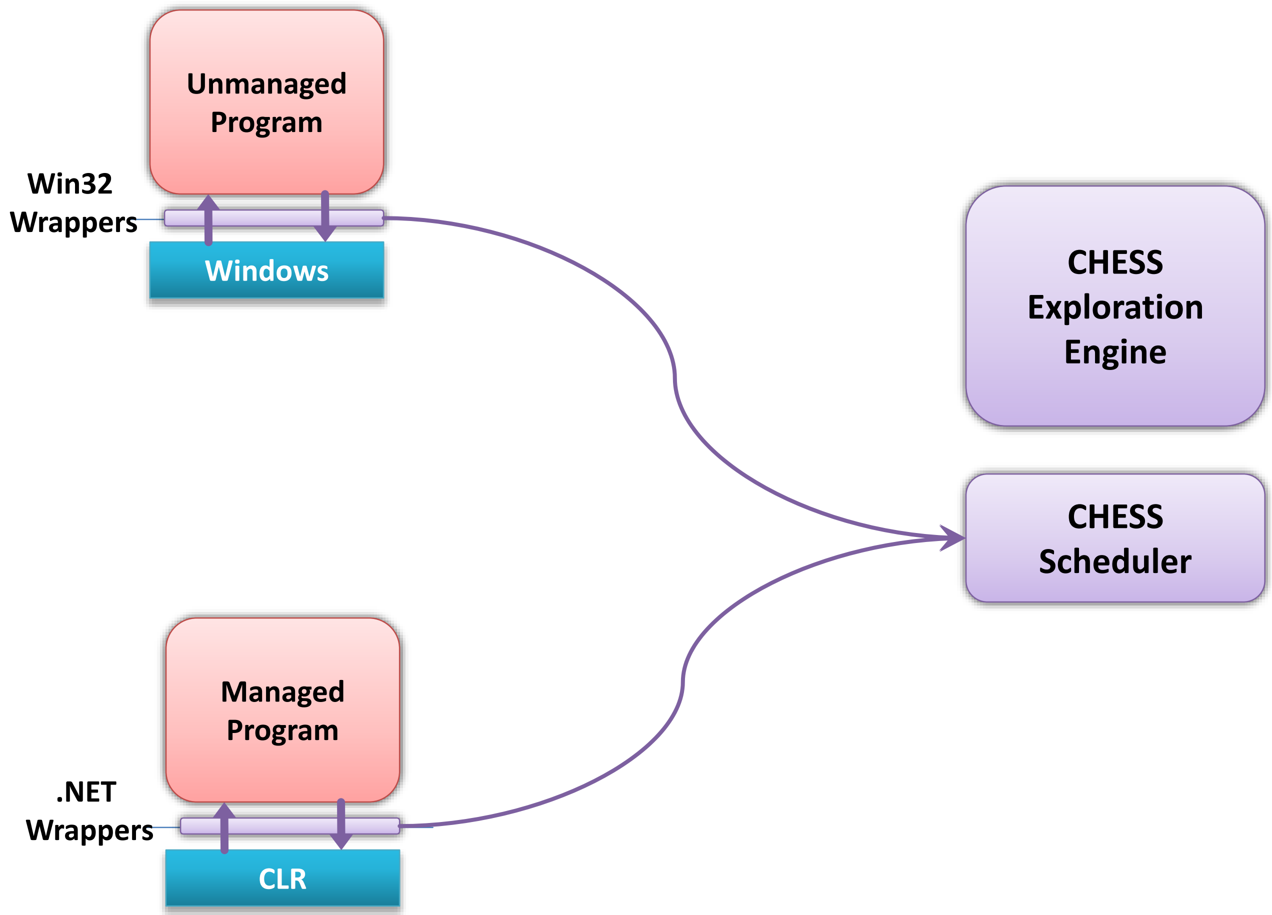
CHESS architecture



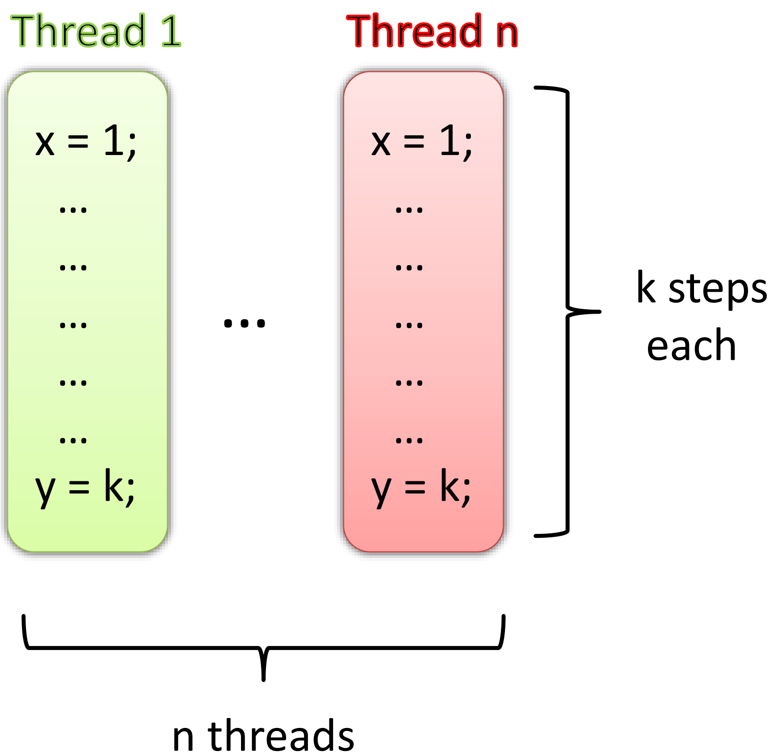
CHESS wrappers

- Translate Win32/.NET synchronizations
- Into CHESS scheduler abstractions
  - Tasks : schedulable entities
    - Threads, threadpool work items, async. callbacks, timer functions
  - SyncVars : resources used by tasks
    - Generate happens-before edges during execution
- Executable specification for complex APIs
  - Most time consuming and error-prone part of CHESS
- Enables CHESS to handle multiple platforms

CHESS architecture



State space explosion

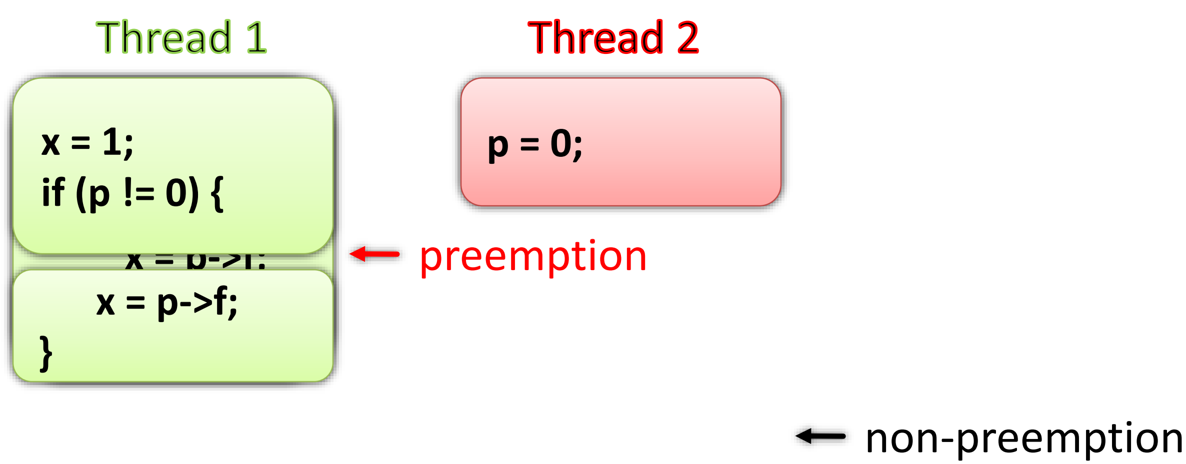


- Number of executions $= O(n^{nk})$
- Exponential in both n and k
  - Typically: $n < 10$ $k > 100$
- Limits scalability to large programs

Goal: Scale CHESS to large programs (large k)

Preemption bounding

- CHESS, by default, is a non-preemptive, starvation-free scheduler
  - Execute huge chunks of code atomically
- Systematically insert a small number preemptions
  - Preemptions are context switches forced by the scheduler
    - e.g. Time-slice expiration
  - Non-preemptions – a thread voluntarily yields
    - e.g. Blocking on an unavailable lock, thread end

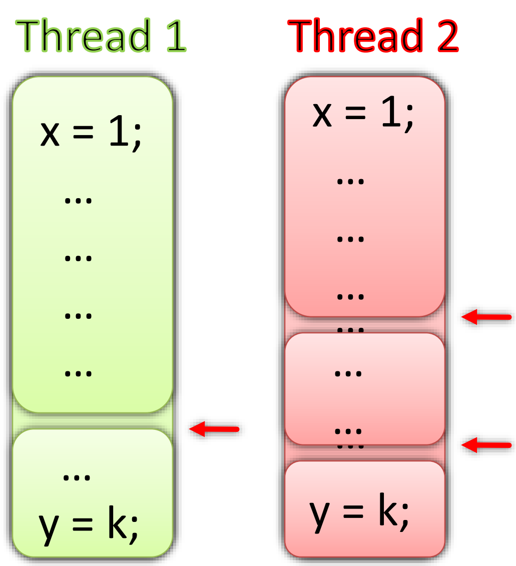


Polynomial state space

- Terminating program with fixed inputs and deterministic threads
  - n threads, k steps each, c preemptions
- Number of executions $\leq {}_{nk}C_c \,.\, (n+c)!$

$$= O(\,(n^2k)^c .\, n!\,)$$

Exponential in n and c, but not in k



- Choose c preemption points
- Permute n+c atomic blocks

Advantages of preemption bounding

- Most errors are caused by few (<2) preemptions
- Generates an easy to understand error trace
  - Preemption points almost always point to the root-cause of the bug
- Leads to good heuristics
  - Insert more preemptions in code that needs to be tested
  - Avoid preemptions in libraries
  - Insert preemptions in recently modified code
- A good coverage guarantee to the user
  - When CHESS finishes exploration with 2 preemptions, any remaining bug requires 3 preemptions or more

Does CHESS scale?

- The scheduler definitely does
  - Can attach to programs like Singularity, IE, Windows Graphics framework
  - Found and reproduced (unkown) bugs in all of them
- The exploration engine? yes.
  - Preemption bounding with heuristics does a good job
    - CHESS has reproduced any Heisenbug reported to us so far
    - Can also be because of “low hanging fruits”
  - Better heuristics, reduction strategies, and massive parallelization will help

Characteristics of input programs to CHESS

Programs	LOC	max Threads	max Synch.	max Preemp.
PLINQ	23,750	8	23,930	2
CDS	6,243	3	143	2
STM	20,176	2	75	4
TPL	24,134	8	31,200	2
ConcRT	16,494	4	486	3
CCR	9,305	3	226	2
Dryad	18,093	25	4,892	2
Singularity	174,601	14	167,924	1

Bugs found with CHESS

Programs	Total	Failure / Bug		
		Unk/Unk	Kn/Unk	Kn/Kn
PLINQ	1		1	
CDS	1		1	
STM	2			2
TPL	9	9		
ConcRT	4	4		
CCR	2	1	1	
Dryad	7	7		
Singularity	1		1	
Total	27	21	4	2

Conclusion

- CHESS is a tool for
  - Systematically enumerating thread interleavings
  - Reliably reproducing concurrent executions
- Coverage of Win32 and .NET API
  - Isolates the search & monitor algorithms from their complexity
- CHESS is extensible
  - Monitors for analyzing concurrent executions
  - Future: Strategies for exploring the state space