CHESS: Systematic Stress Testing for Concurrent Software

Shaz Qadeer
Microsoft Research

Joint work with Madan Musuvathi

Concurrent software

- Operating systems, databases, device drivers
- Mail servers, web servers, browsers, GUIs, ...
- Games
- Applications
  - C/C++ with Pthreads, C#, Java

Concurrency is increasingly important

- Single-core performance has peaked
- Increased hardware performance will come from multiple cores
- To improve performance, software must harness the hardware parallelism

Concurrency is a problem

- Windows 2000 hot fixes
  - Concurrency errors most common defects among "detectable errors"
  - Incorrect synchronization and protocol errors most common defects among all coding errors
- Windows Server 2003 late cycle defects
  - Synchronization errors second in the list, next to buffer overruns

Security vulnerabilities involving race conditions

- Buffer overruns
- Phishing attacks

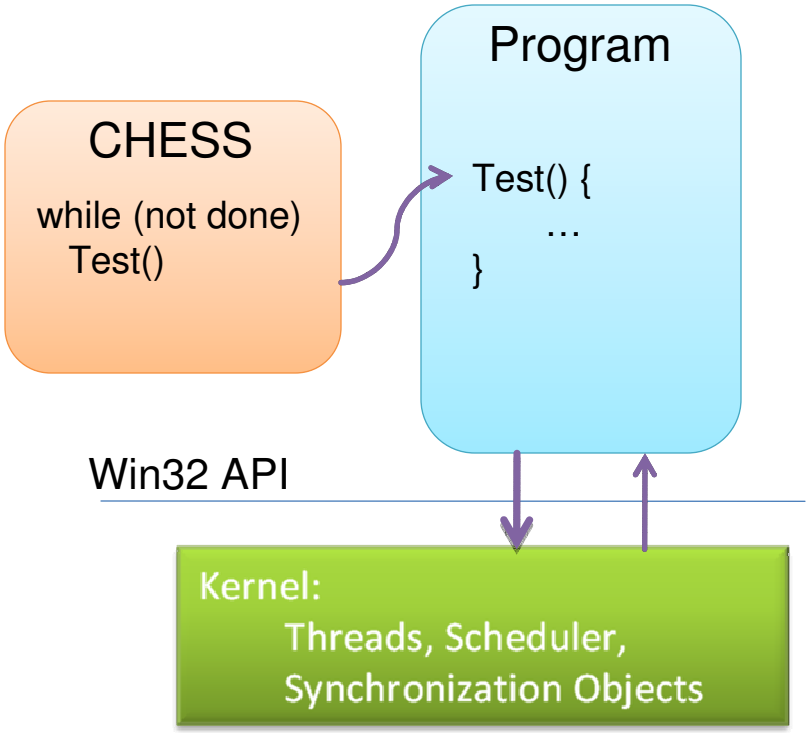
Testing concurrent programs is HARD

- Specific thread interleavings expose subtle errors
  - Testing often misses these errors
- Even when found, errors are hard to debug
  - No repeatable error trace
  - Source of the bug is far away from where it manifests

Current testing methodology

- Think of an interesting test scenario
  - e.g. queue a request while unloading the driver
- Heavily stress this scenario
  - Run thousands of threads for days
- Force scheduling variety
  - Instrument with sleep(), random(),…
- Tester has no control over scheduling nondeterminism

Systematic stress testing with CHESS



Tester provides a test scenario
CHESS runs the scenario in a loop

- Each run takes a different interleaving
- Each run is repeatable

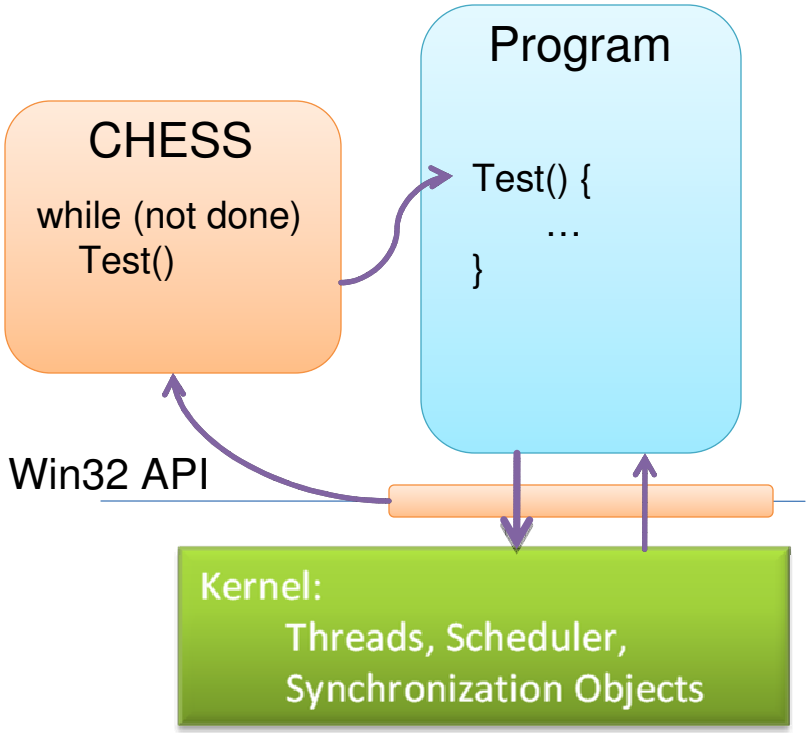
Conditions on Test()

- Test() should terminate in all interleavings
- Test() should be idempotent
  - Free all resources (handles, memory, …)
  - Reset program state
  - "Reset external world"

Key observation:

- Existing stress tests already have these properties
- Because they repeatedly run for ever

Minimize perturbation of system



Run the system as is

- On the actual OS, hardware
- Using system threads
- Using system synchronization objects

Advantages

- Avoid reporting false errors
- Easy to add to existing test frameworks
- Use existing debuggers

Case study: APE toolkit

- APE = Asynchronous Programming Environment
- Windows component written using the Win32 API
- ~10KLOC, highly concurrent
- CHESS revealed
  - Memory leak in test code
  - Data races
    - Double-checked locking
    - Concurrent read/write to different bit-fields of variable
    - …

Outline

- CHESS overview
- Systematic exploration of schedules
- Happens-before relation
- Partial-order reduction
- Iterative context bounding
- Related work

Interleaving granularity

- There is a potential context switch after every instruction
- It suffices to explore context switches only at accesses to synchronization objects
  - Justification later
- Instrument Win32 API calls that perform synchronization to give control to the model checker

Thread schedule = Finite sequence of thread identifiers

Systematic stress testing
=
Exhaustive exploration of thread schedules

in contrast with

Model checking
=
Exhaustive exploration of reachable states

Algorithm

1. Impose a total order on thread schedules
2. Execute them systematically in that order
3. Check each execution for assertion violations and deadlocks

Example:

- Lexicographical order on thread identifiers
- Extended to lexicographical order on executions
- Enumerated using depth first search

Problem:

How do we make the system explore a given schedule?

Scheduler nondeterminism

- Nondeterministic choices for the scheduler
  - May introduce a context switch at any point
  - On context switch, may pick any runnable thread
  - On resource release, may wake up any one of the waiting threads

Hijacking the scheduler (I)

- Ensure at most one thread is runnable
  - A semaphore sem(t) initialized to 0 for each thread t
  - When t starts, it blocks on P(sem(t))
  - Thread t executes V(sem(u)) and P(sem(t)) to resume u and suspend itself

Hijacking the scheduler (II)

- Ensure no thread ever blocks on a system resource
  - If system resource unavailable, schedule a different thread

EnterCriticalSection → MyEnterCriticalSection

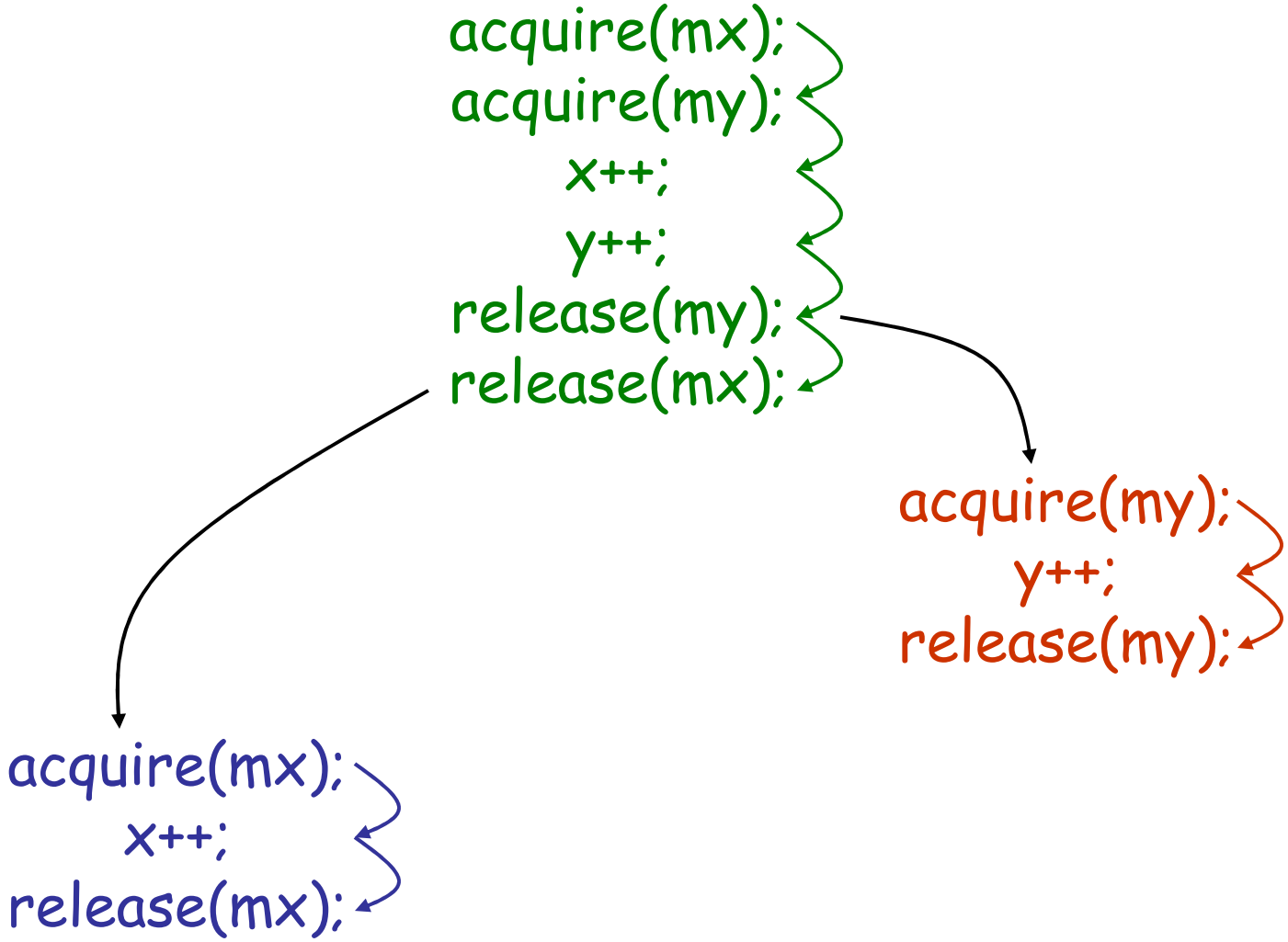
```
MyEnterCriticalSection() {
    while ( !TryEnterCriticalSection() ) {
        // Resource not available
        ScheduleNextThread();
    }
}
```

Why is it sufficient to explore context switches only at accesses to synchronization objects?

Introducing context switches

- Partition of program variables
  - Data variables
  - Synchronization variables
- Accesses to data variables are ordered by accesses to synchronization variables
  - Locks, fork-join, semaphores, events, atomic operations (increment, decrement, test-and-set)

Happens-before relation



Happens-before relation

- There is a data-race on x if there are two accesses to x such that
  - They are unrelated by the happens-before relation
  - At least one of those accesses is a write

No race

```
acquire(mx);
acquire(my);
    x++;
    y++;
release(my);
release(mx);
```

```
acquire(my);
    y++;
release(my);
```

```
acquire(mx);
    x++;
release(mx);
```

Race on x

```
acquire(mx);
acquire(my);
    x++;
    y++;
release(my);
release(mx);
```

```
acquire(my);
    y++;
release(my);
```

```
x++;
```

A data race usually indicates an error!

Introducing context switches

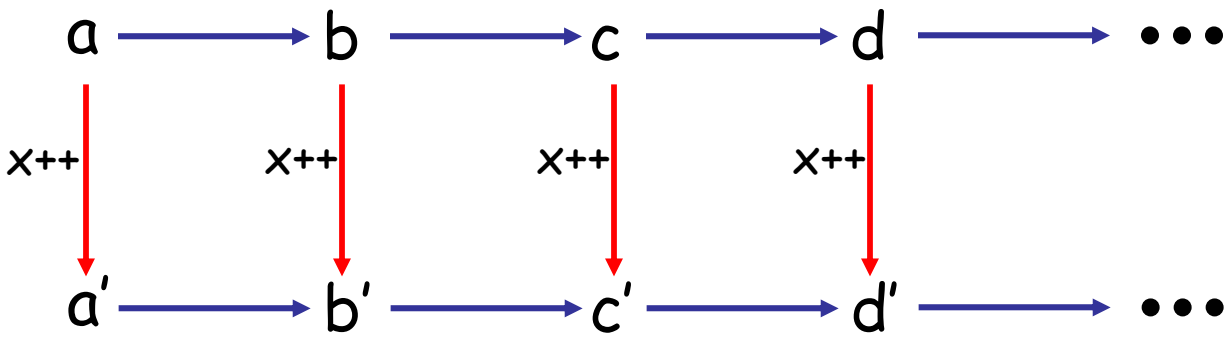
- Partition of program variables
  - Data variables
  - Synchronization variables
- Accesses to data variables are ordered by synchronization actions on synchronization variables
  - Locks, fork-join, semaphores, events, atomic operations (increment, decrement, test-and-set)
- Instrument data accesses and report data-races as errors

Algorithm

1. Impose a total order on thread schedules
2. Execute them systematically in that order
3. Check each execution for assertion violations and deadlocks
4. For each execution and data variable x, check that conflicting accesses to x are ordered by the happens-before relation

Theorem: Partial-order reduction

- Suppose CHESS terminates on a program without reporting a data-race or an error, then the program is error-free.



Multi-core executions

- CHESS systematically enumerates all sequentially consistent executions
- Any data-race free multi-core execution is equivalent to a sequentially consistent execution
- Therefore, CHESS will not miss errors in multi-core executions

Algorithm

1. Impose a total order on thread schedules
2. Execute them systematically in that order
3. Check each execution for assertion violations and deadlocks
4. For each execution and data variable x, check that conflicting accesses to x are ordered by the happens-before relation

Clock-vector algorithm

Initially:

$CV(l) = [0,\ldots,0]$ $CV(x) = [0,\ldots,0]$ $CV(t) = [0,\ldots,\overset{\text{t-th element}}{1},\ldots,0]$

Thread t performs:

release(l): $CV(t)[t] := CV(t)[t] + 1;\ CV(l) := CV(t)$

acquire(l): $CV(t) := \max(CV(t), CV(l))$

access(x):
```
if (CV(x) ≤ CV(t))
        CV(x) := CV(t)
else
        Report race on x
```

n = number of threads

Algorithm requires $\Theta(n)$ time per operation

Goldilocks algorithm

- Synchronization protocol associated with each potentially shared variable
  - thread-local variables
  - lock-protected variables
- Goldilocks handles the common case efficiently

Initially:

owner(x) =- 1 LS(x) = Locks LH(t) = { }

Thread t performs:

release(l): Remove l from LH(t)

acquire(l): Add l to LH(t)

```
access(x):  if (owner(x) = t)
                // No race on x
            else if (LS(x) ∩ LH(t) ≠ { })
                // No race on x
            else
                Report race on x
            owner(x) := t; LS(x) := LH(t)
```

Example 1

```
class Ref { int f; }
Ref g;

Lock m;
```

```
Ref o;
o := new Ref;
o.f := 1;
acquire(m);
g.f++;
release(m);
o.f++;
```

```
acquire(m);
g.f++;
release(m);
```

Example 2

```
class Ref { int f; }
Ref g;

Lock m;
```

```
Ref o;
o := new Ref;        // Ref object o1 allocated
o.f := 1;            Race reported on o1.f!
acquire(m);
g = o;
release(m);
```

```
acquire(m);
g.f++;
release(m);
```

Initially:

owner(x) =- 1 LS(x) = Locks LH(t) = { }

Thread t performs:

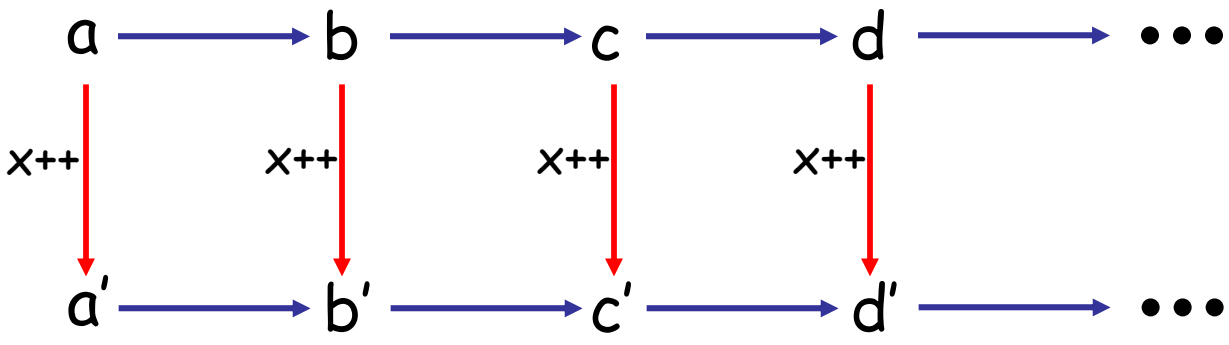
release(l): Remove l from LH(t)

acquire(l): Add l to LH(t)

```
access(x):  if (owner(x) = t)
                // No race on x
            else if (LS(x) ∩ LH(t) ≠ { })
                // No race on x
            else if (CheckOwnershipTransfer())
                // No race on x
            else
                Report race on x
            owner(x) := t; LS(x) := LH(t)
```

Partial-order reduction

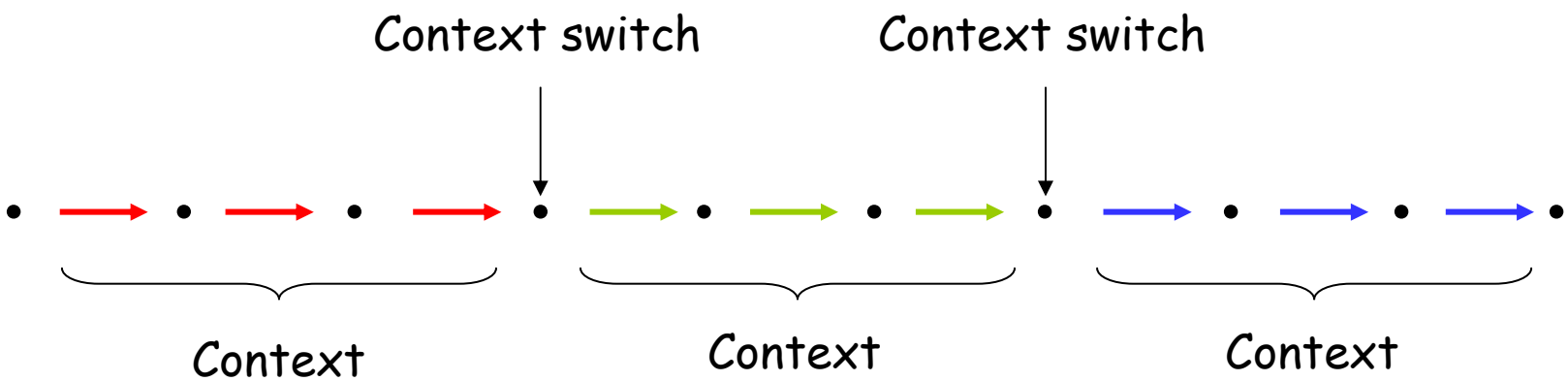
- Suppose CHESS terminates on a program without reporting a data-race or an error, then the program is error-free.



Prioritized search

- Partial-order reduction prunes away many redundant executions
- For realistic programs, there are still a huge number of executions
- Is there a way to prioritize search for erroneous executions?

Iterative context bounding



- Give priority to executions with fewer context switches
- Different from bounded-depth model checking
  - no bound on the computation within each context

Three reasons

- Finds smallest number of context switches to the error
  - Error traces produced are easy to understand
- In practice, many errors manifest themselves within small context bounds

KISS: a static analysis tool

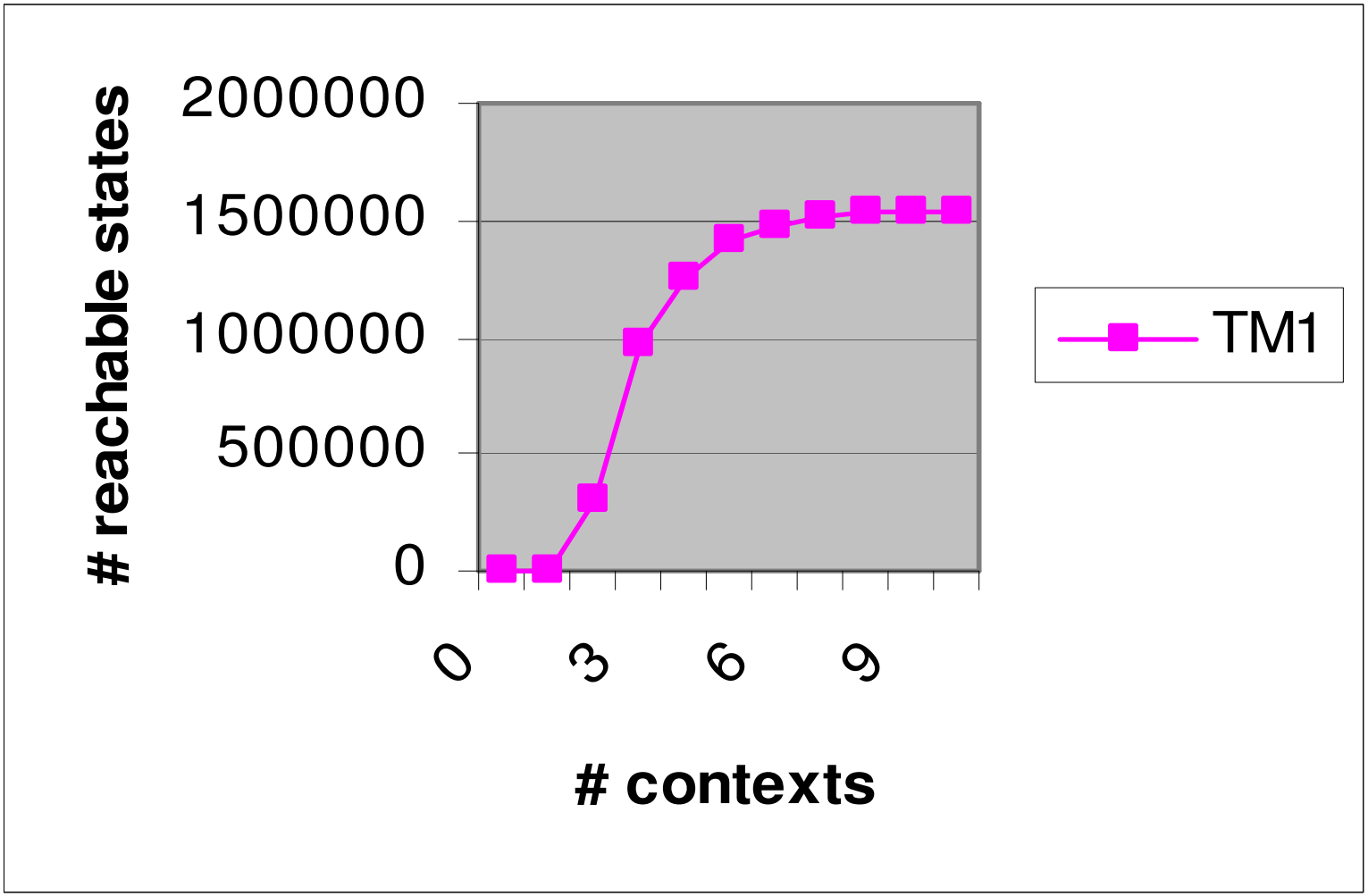
- Technique to use any sequential checker to perform context-bounded concurrency analysis
- Found a number of concurrency errors in NT device drivers even with a context-switch bound of two

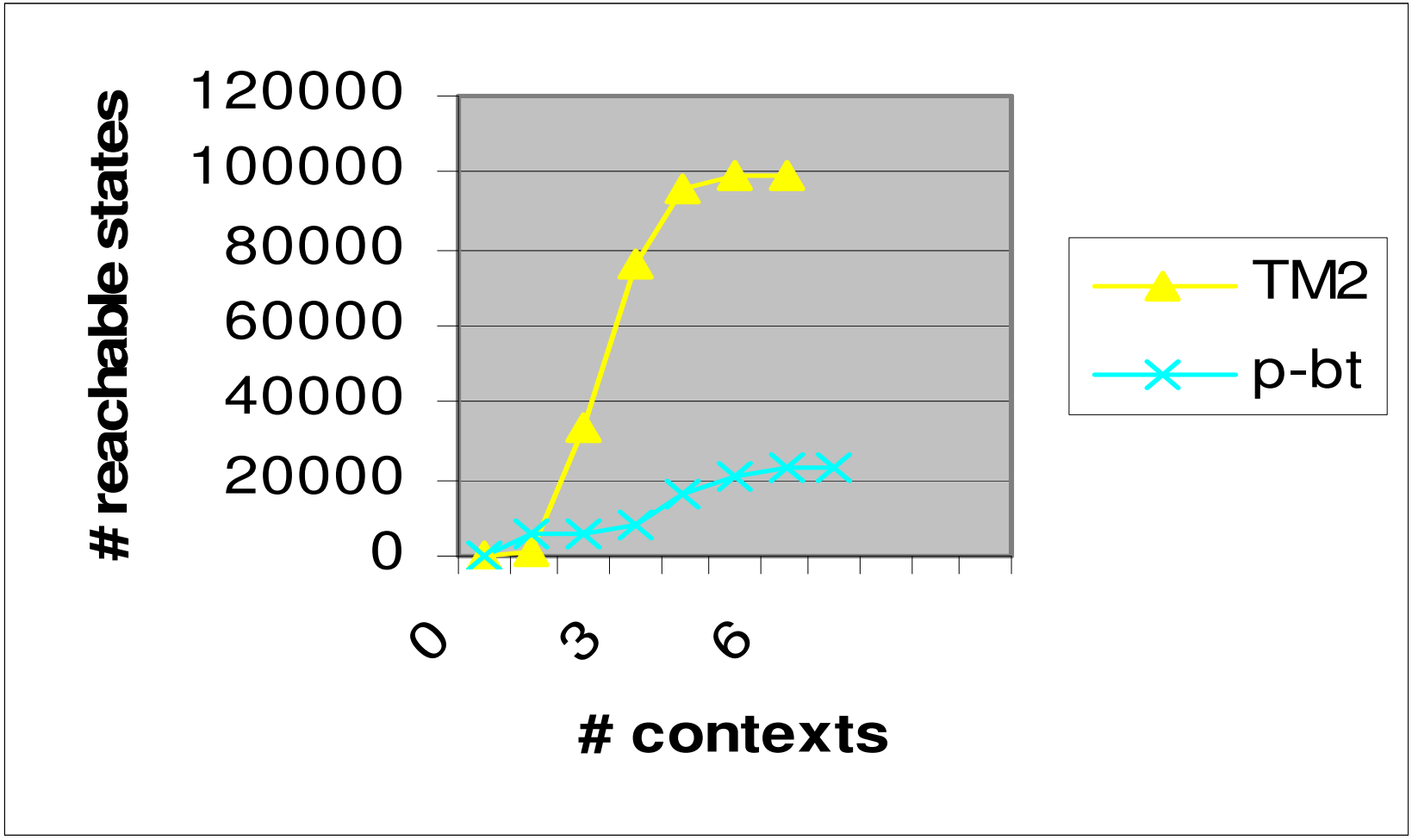
Driver	KLOC	# Fields	# Races
Tracedrv	0.5	3	0
Moufiltr	1.0	14	0
Kbfiltr	1.1	15	0
Imca	1.1	5	1
Startio	1.1	9	0
Toaster/toastmon	1.4	8	1
Diskperf	2.4	16	0
1394diag	2.7	18	0
1394vdev	2.8	18	1
Fakemodem	2.9	39	6
Toaster/bus	5.0	30	0
Serenum	5.9	41	2
Toaster/func	6.6	24	5
Mouclass	7.0	34	1
Kbdclass	7.4	36	1
Mouser	7.6	34	1
Fdc	9.2	92	9

Total:
30 races

Zing: an explicit-state model checker

- Case study (Naik-Rehof 04): Concurrent transaction management code from Microsoft product group
- Analyzed by the Zing model checker after automatically translating to the Zing input language
  - Found three bugs each requiring between three and four context switches


reachable states
2000000
1500000
1000000
500000
0
0
3
6
9
contexts
TM1


reachable states
120000
100000
80000
60000
40000
20000
0
0
3
6
contexts
TM2
p-bt

Three reasons

- Finds smallest number of context switches to the error
  - Error traces produced are easy to understand
- In practice, many errors manifest themselves within small context bounds
- Context bounding leads to polynomial bound on the number of executions
  - n threads, each executing k steps
  - total no. of executions = $\Omega(n^k)$
  - With context bound c, no. of executions = $O((n^2.k)^c)$

Challenges

- Scale to large programs
  - combine partial-order reduction and iterative context bounding
- Real software is nondeterministic for reasons other than thread scheduling
  - hardware performance counters, different thread handles in different runs
  - How do we ensure repeatability?

Related work

- Model checking
  - Partial-order reduction
  - SPIN, Java Pathfinder, Bogor, Verisoft
- Reachability testing
- Static and dynamic data-race detection

Conclusion

- Concurrency will become very important to computing
  - as multi-core processors become common
- Building robust concurrent software remains a challenge
- CHESS project attempts to achieve systematic testing of concurrent software
  - provide repeatable error traces for debugging
  - partial-order reduction and iterative context bounding for pruning the search space
- At MSR, we are starting a new group devoted to "Concurrency Research"

Thank You !