Iterative Context Bounding for Systematic Testing of Multithreaded Programs

Madan Msuvathi Shaz Qadeer
Microsoft Research

Presented by Yuli Ye
2008

Concurrent software

- Operating systems
- Mail servers, web servers
- Databases
- Device drivers
- Games
- ……

Concurrency is important

- Internet and multi-user environment require more and more applications to handle concurrency

- Hardware changes, e.g., multiple cores, require software to harness the hardware parallelism to improve performance

Concurrency is a problem

- Windows 2000 hot fixes
  - Concurrency and synchronization errors are most common coding errors
- Windows Server 2003 late cycle defects
  - Synchronization errors are second in the list, next to buffer overruns
- Race conditions can lead to security vulnerability

Concurrent programs are hard

- It is hard to write a correct concurrent program
  - People get more used to think sequentially than concurrently

- It is also hard to test a concurrent program
  - Thread interleaving may create subtle errors which are hard to catch

- Even when found, errors are hard to debug
  - An error may not repeat itself very often
  - An error may occur far away from its source

Traditional testing methods

- Find interesting test scenario
  - Create some test cases that we think are "interesting"
- Stress testing
  - Run thousand threads for days
- Force scheduling variety
  - Use random() and sleep()

Disadvantages of the above three approaches

- Many are heuristic based
- No guarantees on coverage
- Rely too much on the tester

Testing with model checking

- Advantages
  - Systematically executes each thread schedule to control non-determinism
  - Capable of reproducing an error once found and hence easier to debug

- Disadvantages
  - State explosion: the number of possible program behaviors grow explosively with the size of the program
  - Almost infeasible for large concurrent program with limited resource of memory and time

State explosion I

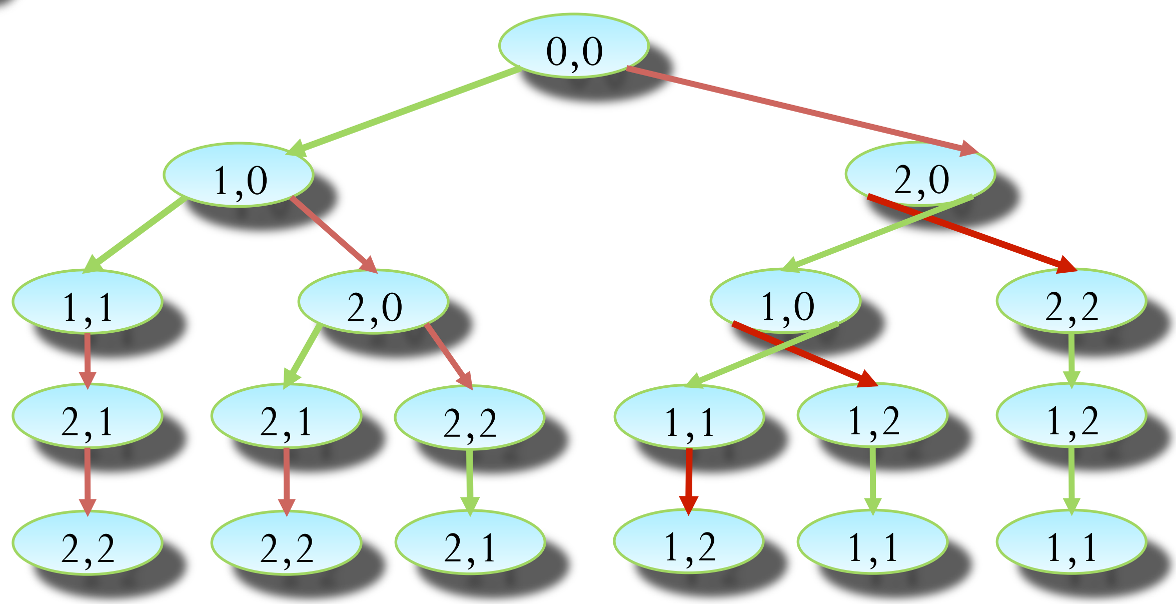
Thread 1

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

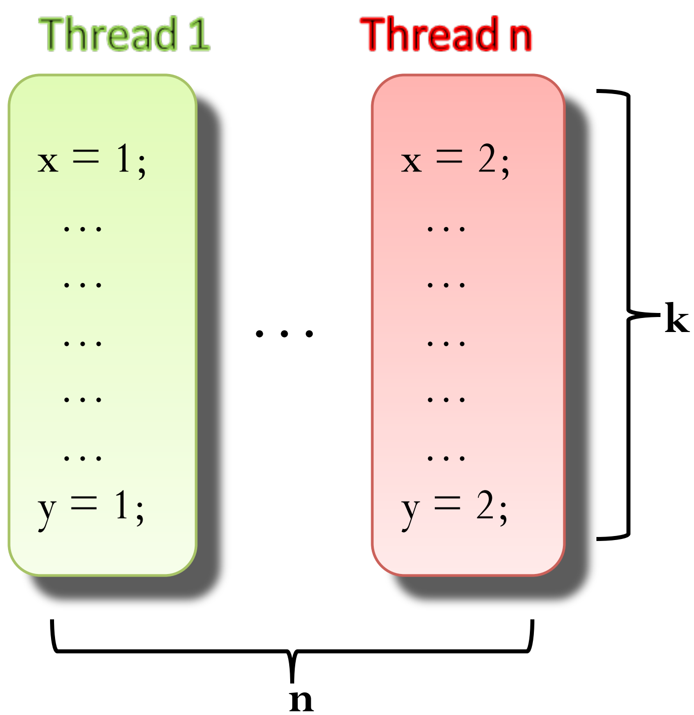
Thread 2

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

```
x = 1;
y = 1;
x = 2;
y = 2;
```



State explosion II



Theorem

With n threads and at most k steps at each thread, the total number of execution maybe as large as

$$(nk)!/(k!)^{n} >= (n!)^{k}$$

Iterative depth bounding

Iterative depth bounding limits the execution with a bounded number of steps

- Runs out of resource quickly as the depth is increased
- Most useful for program with small depth from the initial state, e.g., message-passing software
- Does not work well for multithread programs with fine-grained interaction through shared memory
- Usually have a very poor coverage of states explored

CHESS: Iterative context bounding

A **context switch** occurs at a schedule points if the scheduler chooses a thread different from the current running thread

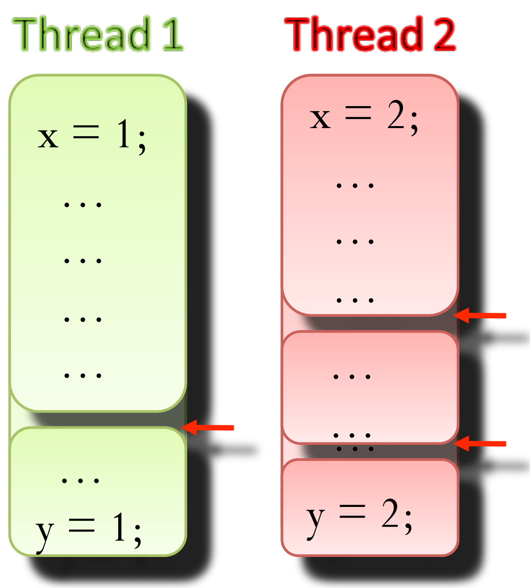
There are two kinds of context switches

- Preemptions – forced by the scheduler
  - e.g. time-slice expiration
- Non-preemptions – a thread voluntarily yields
  - e.g. blocking on an unavailable resource

In **context bounding**, we bound the number of preemptions but leave the number of non-preemptions unconstraint

Benefits of context bounding 1

Polynomial state space



Theorem

If a program has at most c preemptions and n threads. Each thread has at most k steps of with at most b are potentially-blocking, the total number of execution is bounded by

$$_{nk}C_c \,.\, (nb+c)! = O(\,(n^2kb)^c.(\,nb)!\,)$$

Benefits of context bounding 2

Possible deep exploration with small bounds

- The number of steps within each context remains unbounded, so we overcome the limitation of depth bounding
- The number of non-preemption within each context remains unbounded, therefore even a bound of zero may lead to complete termination executions

Benefits of context bounding 3

Better coverage metric

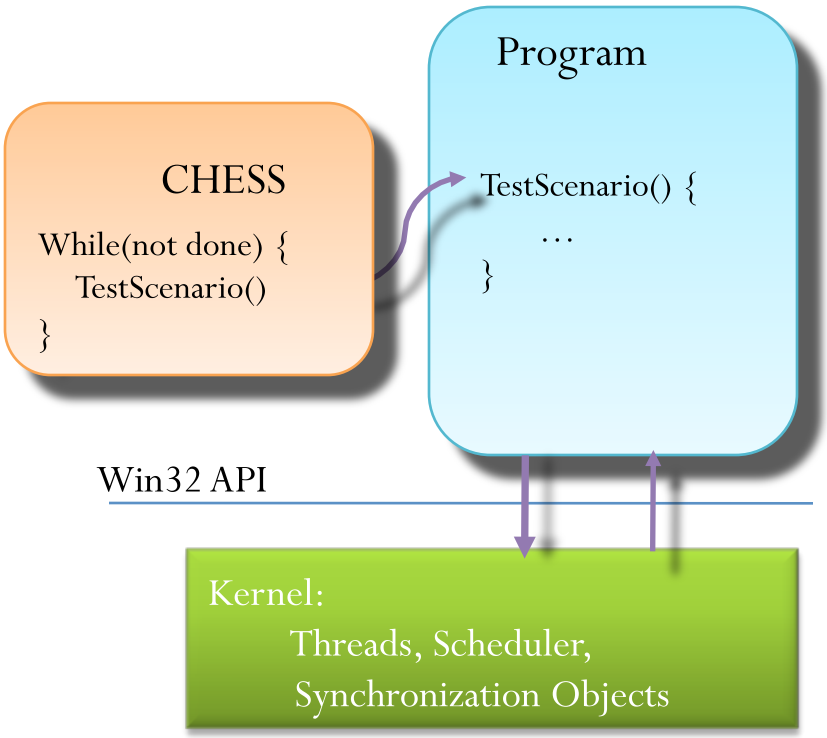
- Finds the smallest number of preemptions to the error
- Gives an estimate on the possible bugs remaining in the program and hence an estimate on the chance of their occurrence in practice

Benefits of context bounding 4

Many bugs within small number of preemptions

- Based on a non-blocking implementation of the work-stealing queue algorithm
  - Bounded circular buffers accessed concurrently by two threads
- A test harness and three bugs are given
  - Each bug found with at most 2 preemption
  - Although execution with 35 preemptions are possible

Architecture of CHESS



Tester Provides a Test Scenario

CHESS runs test scenario in a loop

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

Intercept synchronization and threading calls

- Control and schedule non-determinism

Detects

- Assertion violations
- Deadlock
- Livelock
- Data-races

Conditions on TestScenario()

- TestScenario() should terminate in all interleavings
- TestScenario() should be idempotent
  - Free all resourcs
  - Reset global states
- TestScenario() should not interfere with other tasks in the program being tested

Observation:

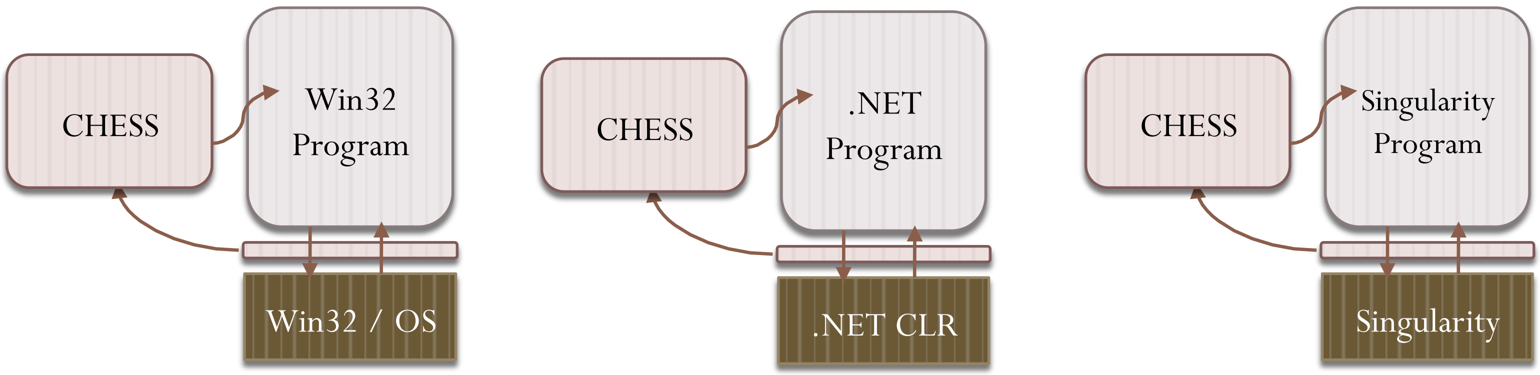
Existing stress tests usually satisfy these properties

Perturb the system as little as possible

- 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

CHESS methodology generalizes



- CHESS works for
  - Unmanaged programs, such as code written in C and C++
  - Managed programs, such as code written in C#
  - Singularity applications
- With appropriate wrappers, can work for Java and Linux applications

CHESS: The algorithm I

- Effectively search the state space of a program by systematically bounding the number of preemptions
- Assume the program is data-race free
- Context switch only at synchronization points
- Check for data-races in each execution

CHESS: The algorithm II

```
Input: initial state s0 ∈ State and context switch bound csb
struct WorkItem { State state; Tid tid; int phase; }
Queue⟨WorkItem⟩ workQueue;
WorkItem w;
int currPhase;
for t ∈ Tid do
    w.state := s0;
    w.tid := t;
    w.phase := 0;
    workQueue.Add(w);
end
currPhase := 0;
while ¬workQueue.Empty() do
    w := workQueue.Front();
    workQueue.Pop();
    if currPhase < w.phase then
        /* explored (currPhase + 1) ∗ csb + currPhase
           preempting context switches                  */
        currPhase := w.phase;
    end
    Search(w, 0);
end
Search(WorkItem w, int ncs) begin
    if ¬w.state.Enabled(w.tid) then
        return;
    end
    WorkItem x;
    x.state := w.state.Execute(w.tid);
    x.tid := w.tid;
    x.phase := w.phase;
    Search(x, ncs);
    for t ∈ Tid \ {w.tid} do
        x.tid := t;
        if ¬x.state.Enabled(w.tid) then
            x.phase := w.phase;
            Search(x, ncs);
        else if ncs = csb then
            x.phase := w.phase + 1;
            workQueue.Push(x);
        else
            x.phase := w.phase;
            Search(x, ncs+1);
        end
    end
end
```

Algorithm 1: Iterative context bounding

Why does this work?

Theorem

To check a program, it is sufficient to insert a scheduling point before a synchronization operation in the program, provided that the algorithm also checks for data-races

The strategy is essentially a partial-order reduction

Empirical evaluation

Evaluation is done on a set of benchmark programs

- Bluetooth
- File system model
- Work-stealing queue
- APE
- Dryad channels
- Transaction manager

Characteristics of benchmarks

Programs	LOC	Num Threads	Max K	Max B	Max c
Bluetooth	400	3	15	2	8
File System Model	84	4	20	8	13
Work Stealing Q.	1266	3	99	2	35
APE	18947	4	247	2	75
Dryad Channels	16036	5	273	4	167

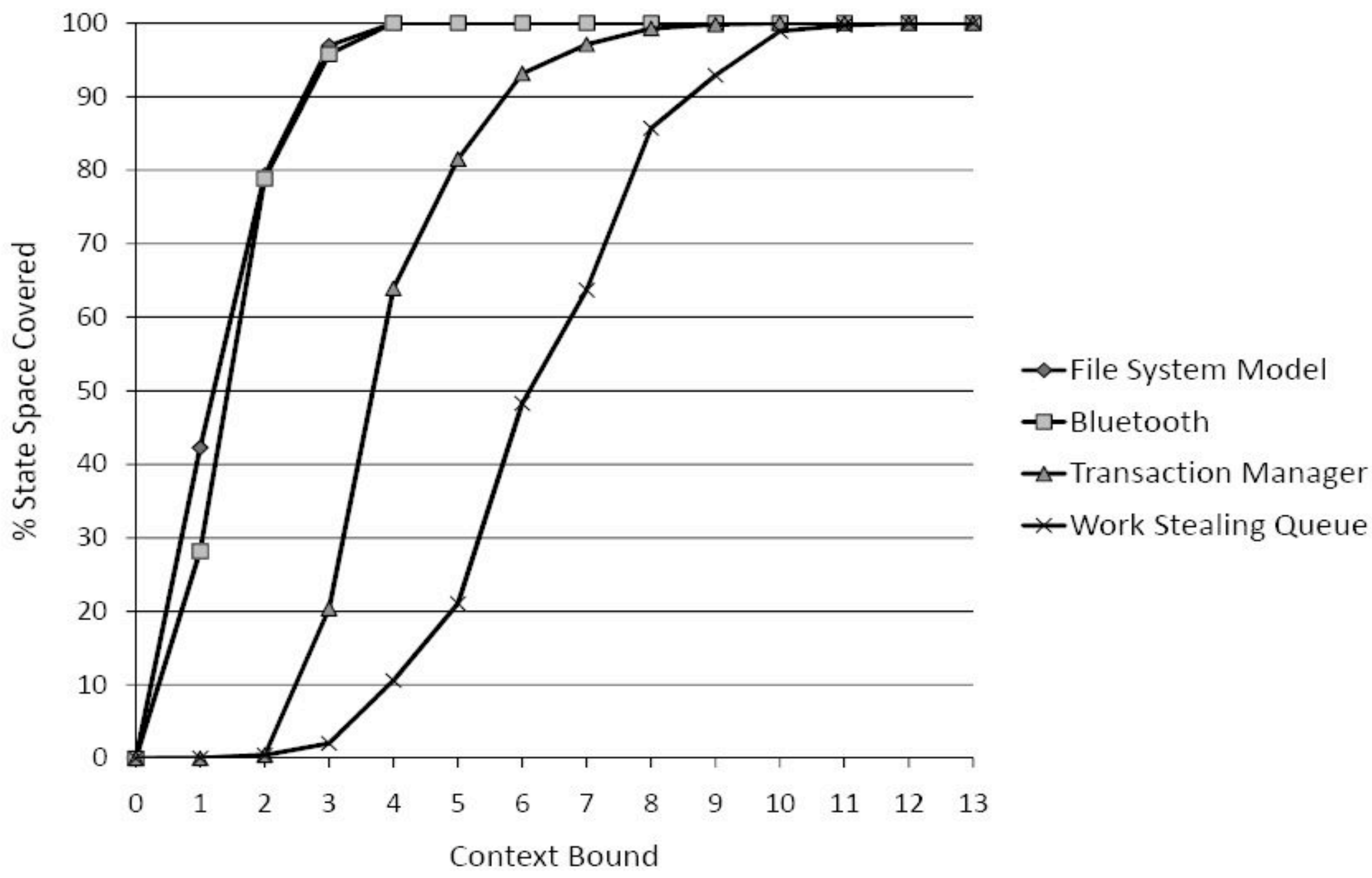
Table 1. Characteristics of the benchmarks. For each benchmark, this table reports the number of lines, the number of threads allocated by the test driver. For an execution, K is the total number of steps, B is the number of blocking instructions, and c is the number of preempting context switches. The table reports the maximum values of K,B, and c seen during our experiments.

Bugs found with small context bound

Programs	Total Bugs	Bugs with Context Bound			
		0	1	2	3
Bluetooth	1	0	1	0	0
Work Stealing Queue	3	0	1	2	0
Transaction Manager	3	0	0	2	1
APE	4	2	1	1	0
Dryad Channels	3	1	2	0	0

Table 2. For a total of 14 bugs that our model checker found, this table shows the number of bugs exposed in executions with exactly c preempting context switches, for c ranging from 0 to 3. The 7 bugs in the first three programs was previously known. Iterative context-bounding algorithm found the 7 previously *unknown* bugs in Dryad and APE.

Coverage vs. Context bound



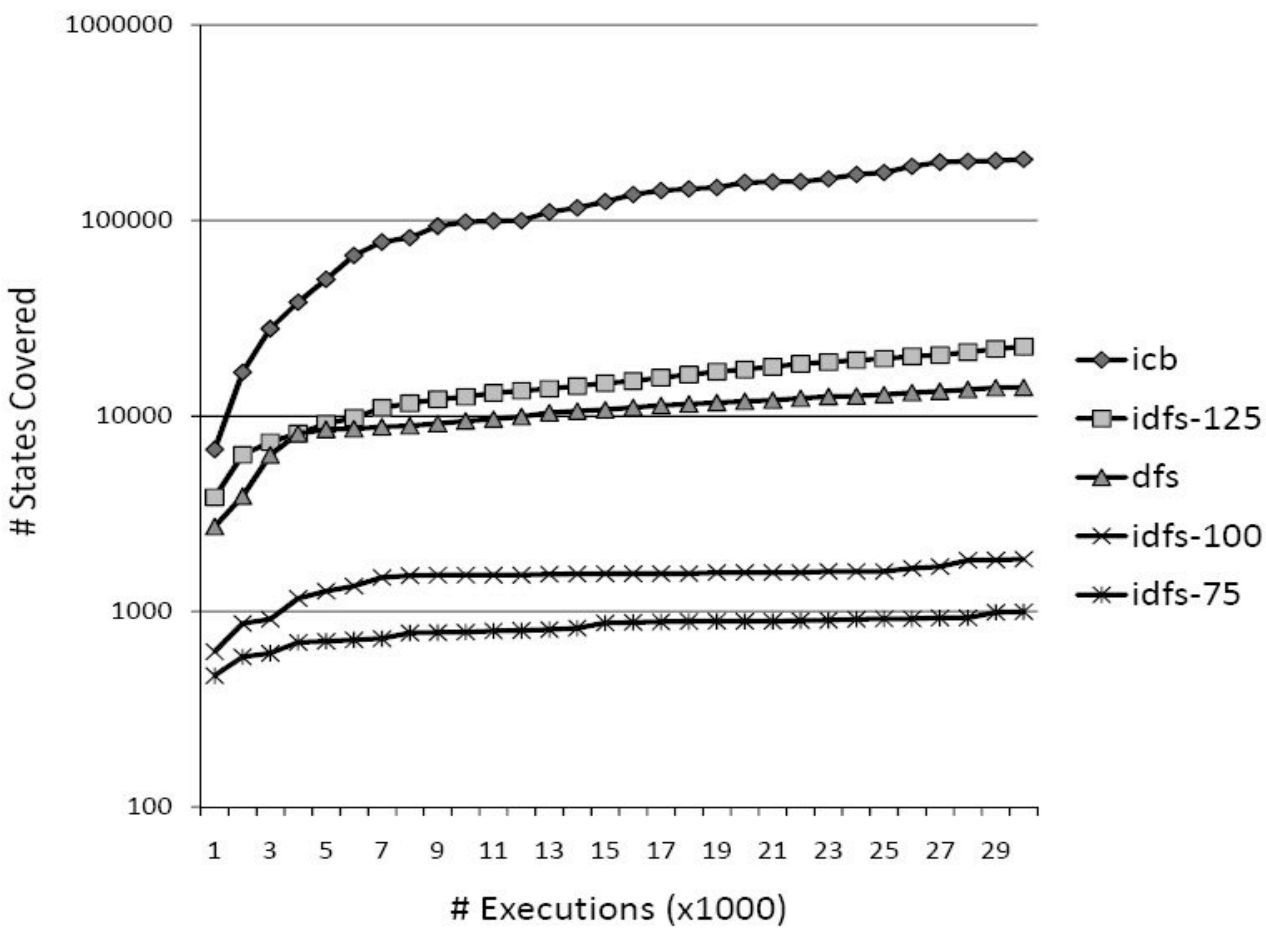
Dryad bugs

- Total of 7 bugs are found in spite of careful regression testing and months of production use

- The use-after-free bug has long error trace but requires only one preemption
  - Depth bounding is hard to find

- The error trace has 6 non-preempting context switches
  - Unrestricting non-preemption is important

Coverage vs. time in Dryad



Conclusion

- Currency is important but hard to get it right, building robust concurrency software remains a challenge
- Traditional testing and debugging methods are unsatisfying in providing guarantees of detecting and correcting errors
- CHESS is a systematic testing tool that provides:
  - Good coverage without scarifying the ability to go deep into the state space
  - Good integration capability with the existing test frameworks
  - Replay capability for debugging
- Iterative context bounding is a useful approach in designing concurrency testing tools

Thank you!

- Musuvathi, M and Qadeer, S. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07), pages 446 - 455. San Diego, California, USA, June 2007.*

- http://research.microsoft.com/projects/CHESS/

- http://research.microsoft.com/projects/CHESS/IterativeContextBounding.pdf