Abstract
Concurrency is pervasive in large systems. Unexpected interference among threads often results in "Heisenbugs" that are extremely difficult to reproduce and eliminate. We have implemented a tool called CTRex for finding and reproducing such bugs. When attached to a program, CTRex controls execution of test cases and uses efficient search techniques to drive the program through possible thread interleavings. This system exploits program behavior enables CTRex to quickly uncover bugs that might otherwise have remained hidden for a long time. For each bug, CTRex consistently reproduces an erroneous execution manifesting the bug, thereby making it significantly easier to debug the program. CTRex scales to large concurrent programs and has found numerous bugs in existing systems that had been tested extensively prior to being tested by CTRex. CTRex has been integrated into the test frameworks of many code bases inside Microsoft and is used by testers on a daily basis.

1 Introduction
Building concurrent systems is hard. Slight changes among threads and the timing of asynchronous events can result in concurrency errors that are hard to find, reproduce, and debug. Stories are legion of so-called "Heisenbugs" [18] that occasionally surface in systems that have otherwise been working reliably for months. Slight changes to a program, such as the addition of debugging statements, sometimes drastically reduce the likelihood of erroneous interactions, adding frustration to the debugging process.

The main contribution of this paper is a new tool called CTRex for finding and reproducing concurrent system bugs. When attached to a concurrent program, CTRex takes complete control over the scheduling of threads and asynchronous events, thereby capturing all the interleaving nondeterminism in the program. This provides a powerful new technique for finding concurrency errors that are hard to reproduce. First, if an execution results in an error, CTRex has the capability to reproduce the erroneous thread interleaving. This substantially improves the debugging experience. Second, CTRex uses a systematic enumeration technique [20, 37, 17, 23, 45, 22] to ensure every run of the program along a different thread interleaving. CTRex’s tools control the program state in such a way that it greatly increases the chances of finding previously unobserved errors. More importantly, there is a high likelihood of automatically “stressing” the system, stimulating the discovery of new classes of bugs.

We have implemented a tool called CTRex for finding and reproducing concurrent system bugs. When attached to a program, CTRex controls execution of test cases and uses efficient search techniques to drive the program through possible thread interleavings. This system exploits program behavior enables CTRex to quickly uncover bugs that might otherwise have remained hidden for a long time. For each bug, CTRex consistently reproduces an erroneous execution manifesting the bug, thereby making it significantly easier to debug the program. CTRex scales to large concurrent programs and has found numerous bugs in existing systems that had been tested extensively prior to being tested by CTRex. CTRex has been integrated into the test frameworks of many code bases inside Microsoft and is used by testers on a daily basis.

Iterative Context Bounding for Systematic Testing of Concurrent Programs

1. Introduction
Multi-threaded programs are difficult to get right. Specific thread interleavings, unexpected even to an expert programmer, lead to crashes that occur late in the software development cycle or even after the software is released. The traditional method for testing concurrent software in the industry is stress testing, in which the software is executed under heavy loads with the hope of producing an erroneous interleaving. Empirical evidence clearly demonstrates that this form of testing is inadequate. CTRex uses systematic exploration to drive the program to likely execution paths that are extremely difficult to reproduce and eliminate.

To build a systematic testing tool capable of handling current programs, several challenges arise.

First, such a tool should be able to parallelize its effort. To test a large program, CTRex must explore a large number of scheduling orders, a nontrivial task of capturing and exploring the executions of a small program, the minimum number of scheduling constraints required to find it. Such an exploration order that prioritizes executions with fewer scheduling constraints enables the program designer to get the concurrency test environment right before proceeding and sample the remaining context. The bound enables the tool to explore a small number significantly affects the results of a concurrency bug by a given test program with supplied inputs. However, it improves on traditional testing methods. A model checker systematically explores the domain that is subject to concurrency. In a single run of a program with k threads, the minimum number of scheduling constraints required to find it.

Second, a systematic testing tool must control the program state. CTRex needs to capture all the state changes that together execute the current program and an input test harness. To the first-order of approximation, a bound is not useful. Where the program is run for days (not hours), a randomized algorithm for concurrency testing. Given a concurrency bug, CTRex randomly schedules the current program and an input test harness. CTRex randomizes the threads of the program during each test run. In contrast to prior randomized testing techniques, CTRex uses randomization smarter and in a disciplined manner. As a result, CTRex provides more realistic probability of finding a concurrency bug in each run. Repeated independent runs can increase the probability of finding bugs (or even weeks under heavy loads) with the hope of hitting bugs without any significant effort to reproduce them. CTRex systematically explores the execution paths that are extremely difficult to reproduce and eliminate.

This paper presents a randomized scheduler for finding concurrency errors. A randomized algorithm for concurrency testing. Given a concurrency bug, CTRex randomly schedules the current program and an input test harness. CTRex randomizes the threads of the program during each test run. In contrast to prior randomized testing techniques, CTRex uses randomization smarter and in a disciplined manner. As a result, CTRex provides more realistic probability of finding a concurrency bug in each run. Repeated independent runs can increase the probability of finding bugs (or even weeks under heavy loads) with the hope of hitting bugs without any significant effort to reproduce them. CTRex systematically explores the execution paths that are extremely difficult to reproduce and eliminate.

Concurrent programs are difficult to get right because of the combinatorial explosion of concurrently executing threads. Traditional testing methods are inadequate for catching subtle con- current behaviors. However, it is difficult to perform systematic testing because of the number of possible executions exponentially with the program size. Concurrency is pervasive in large systems. Unexpected even to an expert programmer, lead to crashes that occur late in the software development cycle or even after the software is released. The traditional method for testing concurrent software in the industry is stress testing, in which the software is executed under heavy loads with the hope of producing an erroneous interleaving. Empirical evidence clearly demonstrates that this form of testing is inadequate. CTRex uses systematic exploration to drive the program to likely execution paths that are extremely difficult to reproduce and eliminate.

To combat the state-explosion problem, researchers have introduced various techniques for testing multithreaded software. However, it is difficult to perform systematic testing because of the number of possible executions exponentially with the program size. Concurrency is pervasive in large systems. Unexpected even to an expert programmer, lead to crashes that occur late in the software development cycle or even after the software is released. The traditional method for testing concurrent software in the industry is stress testing, in which the software is executed under heavy loads with the hope of producing an erroneous interleaving. Empirical evidence clearly demonstrates that this form of testing is inadequate. CTRex uses systematic exploration to drive the program to likely execution paths that are extremely difficult to reproduce and eliminate.

To combat the state-explosion problem, researchers have introduced various techniques for testing multithreaded software. However, it is difficult to perform systematic testing because of the number of possible executions exponentially with the program size. Concurrency is pervasive in large systems. Unexpected even to an expert programmer, lead to crashes that occur late in the software development cycle or even after the software is released. The traditional method for testing concurrent software in the industry is stress testing, in which the software is executed under heavy loads with the hope of producing an erroneous interleaving. Empirical evidence clearly demonstrates that this form of testing is inadequate. CTRex uses systematic exploration to drive the program to likely execution paths that are extremely difficult to reproduce and eliminate.

To combat the state-explosion problem, researchers have introduced various techniques for testing multithreaded software. However, it is difficult to perform systematic testing because of the number of possible executions exponentially with the program size. Concurrency is pervasive in large systems. Unexpected even to an expert programmer, lead to crashes that occur late in the software development cycle or even after the software is released. The traditional method for testing concurrent software in the industry is stress testing, in which the software is executed under heavy loads with the hope of producing an erroneous interleaving. Empirical evidence clearly demonstrates that this form of testing is inadequate. CTRex uses systematic exploration to drive the program to likely execution paths that are extremely difficult to reproduce and eliminate.

To combat the state-explosion problem, researchers have introduced various techniques for testing multithreaded software. However, it is difficult to perform systematic testing because of the number of possible executions exponentially with the program size. Concurrency is pervasive in large systems. Unexpected even to an expert programmer, lead to crashes that occur late in the software development cycle or even after the software is released. The traditional method for testing concurrent software in the industry is stress testing, in which the software is executed under heavy loads with the hope of producing an erroneous interleaving. Empirical evidence clearly demonstrates that this form of testing is inadequate. CTRex uses systematic exploration to drive the program to likely execution paths that are extremely difficult to reproduce and eliminate.

To combat the state-explosion problem, researchers have introduced various techniques for testing multithreaded software. However, it is difficult to perform systematic testing because of the number of possible executions exponentially with the program size. Concurrency is pervasive in large systems. Unexpected even to an expert programmer, lead to crashes that occur late in the software development cycle or even after the software is released. The traditional method for testing concurrent software in the industry is stress testing, in which the software is executed under heavy loads with the hope of producing an erroneous interleaving. Empirical evidence clearly demonstrates that this form of testing is inadequate. CTRex uses systematic exploration to drive the program to likely execution paths that are extremely difficult to reproduce and eliminate.

To combat the state-explosion problem, researchers have introduced various techniques for testing multithreaded software. However, it is difficult to perform systematic testing because of the number of possible executions exponentially with the program size. Concurrency is pervasive in large systems. Unexpected even to an expert programmer, lead to crashes that occur late in the software development cycle or even after the software is released. The traditional method for testing concurrent software in the industry is stress testing, in which the software is executed under heavy loads with the hope of producing an erroneous interleaving. Empirical evidence clearly demonstrates that this form of testing is inadequate. CTRex uses systematic exploration to drive the program to likely execution paths that are extremely difficult to reproduce and eliminate.
The Heisenbug problem

Concurrent executions are highly nondeterministic

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
Thread interleavings

Thread 1
- x++;
- x++;

Thread 2
- x*=2;
- x*=2;
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
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
CHESS runs the scenario in a loop
• Every run takes a different interleaving
• Every run is repeatable

Intercept synch. & threading calls
• To control and introduce nondeterminism

Detect
• Assertion violations
• Deadlocks
• Dataraces
• Livelocks
Running Example

Thread 1

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

Thread 2

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

Lock (l);
b = t - y;
Unlock(l);
Introduce Schedule() points

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

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);
```
First-cut solution: Random sleeps

Introduce random sleep at schedule points

Does not introduce new behaviors

- Sleep models a possible preemption at each location
- Sleeping for a finite amount guarantees starvation-freedom

```
Thread 1
Sleep(rand());
Lock(l);
bal += x;
Sleep(rand());
Unlock(l);

Thread 2
Sleep(rand());
Lock(l);
t = bal;
Sleep(rand());
Unlock(l);

Sleep(rand());
Lock(l);
bal = t - y;
Sleep(rand());
Unlock(l);
```
Improvement 1:

- Delays that result in the same “happens-before” graph are equivalent
- Avoid exploring equivalent interleavings
Improvement 2:

- Avoid exploring delays that are impossible
- Identify when threads can make progress
- CHESS maintains a run queue and a wait queue
  - Mimics OS scheduler state

Thread 1

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

Schedule();
Unlock(l);

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

Thread 2

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

Schedule();
Lock (l);
bal += x;
Schedule();
Unlock(l);
Emulate execution on a uniprocessor

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

Thread 1

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

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

Thread 2

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

Schedule();
Lock (l);
t = bal;
Schedule();
Unlock(l);
Capture all sources of nondeterminism?

Scheduling nondeterminism? Yes

Timing nondeterminism? Yes
  ▸ Controls when and in what order the timers fire

Nondeterministic system calls? Mostly
  ▸ CHESS uses precise abstractions for many system calls

Input nondeterminism? No
  ▸ Rely on users to provide inputs
    Program inputs, return values of system calls, files read, packets received,…
      ▸ Good tradeoff in the short term
But can’t find race-conditions on error handling code
  ▸ Future extensions using symbolic execution?
    (DART, jCUTE, SAGE, PEX)
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

- Prioritize executions with small number of preemptions
- 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 lock, thread end

```
x = 1;
if (p != 0) {
  x = p->f;
}
```

```
p = 0;
```

Thread 1

- preemption

Thread 2

- non-preemption
So, is CHESS is unsound?

Soundness: prove that the program is correct for a given input test harness
  ▶ Need to exhaustively explore all interleavings

For small programs, CHESS is sound
  ▶ Iteratively increase the preemption bound

Preemption bounding helps scale to large programs
  ▶ A good “knob” to trade resources for coverage

Better search algorithms → more coverage faster
  ▶ Partial-order reduction
  ▶ Modular testing of loosely-coupled programs
Concurrent programs have cyclic state spaces

- Spinlocks
- Non-blocking algorithms
- Implementations of synchronization primitives
- Periodic timers
- ...

Thread 1

L1: while( ! done) {
    L2: Sleep();
}

Thread 2

M1: done = 1;
A demonic scheduler unrolls any cycle ad-infinatum

Thread 1

while(!done) {
    Sleep();
}

Thread 2

done = 1;

! done

! done

! done

done

done

done
Prune executions beyond a bounded number of steps
Problem 1: Ineffective state coverage

Bound has to be large enough to reach the deepest bug
  - Typically, greater than 100 synchronization operations

Every unrolling of a cycle redundantly explores reachable state space
Problem 2: Cannot find livelocks

Livelocks: lack of progress in a program

```c
Thread 1

temp = done;
while(!temp)
{
    Sleep();
}
```

```c
Thread 2

done = 1;
```
Key idea

- This test terminates only when the scheduler is fair
- Fairness is assumed by programmers

All cycles in correct programs are unfair
A fair cycle is a livelock
We need a fair demonic scheduler

- Avoid unrolling unfair cycles
  - Effective state coverage

- Detect fair cycles
  - Find livelocks (violations of fair termination)
Fair termination allows CHESS to check for arbitrary liveness properties

**Example: Good Samaritan assumption**
- For all threads $t :$ scheduled($t$) $\rightarrow$ yield($t$)
- A thread when scheduled infinitely often yields the processor infinitely often

**Examples of yield:**
- Sleep(), ScheduleThread(), asm {rep nop;}
- Thread completion
Probabilistic Concurrency Testing

- Also a random scheduler, but uses randomization sparingly
  - Repeated independent runs increase probability of finding a bug

- Naive approach is exponential (n threads with k instructions has $n^k$ possible thread schedules)

- But, bugs in practice are not adversarial
  - small number of instructions executed by small number of threads
  - goal: schedule these instructions correctly

- The depth of a concurrency bug is the minimum number of scheduling constraints sufficient to find the bug
  - PCT focuses on probabilistically finding bugs at a given depth
  - Can find a bug at depth $d$ in $O(nk^{d-1})$ independent runs
### Ordering Edges and Depth

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( t = \text{new } T() )</td>
<td>...</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>if ( t-&gt;\text{state} == 1 )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( x = \text{null} )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>if ( x \neq \text{null} )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( x-&gt;\text{print()} )</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>lock(a);</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>lock(b);</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>...</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>...</td>
</tr>
</tbody>
</table>

(c)

- (a) manifests whenever the conditional in T2 executes before \( t = \text{new } T() \).
- (b) manifests when two ordering constraints are satisfied.
- (c) also manifests under two ordering constraints.
Bug Depth

For instance, not all atomicity violations have a depth 2, and in fact, concurrency bugs to have small depths. This is further validated by in Fig. 1 the depth respectively is 1, 2, and 2. We expect many sufficient to find the bug. Any schedule that satisfies these ordering depth of a concurrency bug as the minimum number of ordering constraints to trigger the same bug. In such a case, we order not envisioned by the programmer. We identify a set of these.

We classify concurrency bugs according to a

2.3 Bug Depth

PCT provides a guaranteed probability of finding bugs in every run (2) the mutex is unlocked before the main thread terminates the. However, this classification is not strict. Another characterization of a concurrency bug is its

2.3.1 Relationship with Prior Classification

Concurrent bugs happen when instructions are scheduled in an

Threads are involved. However, this classification is not strict. Another characterization of a concurrency bug is its.

2.4 Naive Randomization

Using a randomized scheduler may appear like an obvious choice. Although it may seem like one constraint (black arrow) can frustrate a naive randomization technique. (Neglect the grey arrow for now.) Even this simple bug requires Thread 1 to be preempted right after the instruction that

However, it is not a priori clear how to design such a scheduler with

ich is inverse exponential in m + 2. But more importantly, any

Depth 2

Main Thread Filewriter Thread

... free(mutex) ... mutex.unlock()
exit(0);

Depth 2

Thread 1 Thread 2

... Set(e);
... t = new T();
... ...
... Wait(e);
... t->state == 1
... ...

Depth 1

Thread 1 Thread 2

... init = true ... ...
... t = new T() ... if ( init )
... ... t->state == 1
... ...

Depth 2
Two bugs of depth 1 difficult to find using a pure randomized scheduler

**The PCT Algorithm**

- instead, use a priority-based scheduler
- the scheduler schedules a low priority thread only when all higher priority threads are blocked
- threads change priorities when they pass a priority change point
- randomly assign initial priority values and randomly pick priority change points