CHESS: Analysis and Testing of Concurrent Programs

Sebastian Burckhardt, Madan Musuvathi, Shaz Qadeer

Microsoft Research

Joint work with
Tom Ball, Peli de Halleux, and interns
Gerard Basler (ETH Zurich),
Katie Coons (U. T. Austin),
P. Arumuga Nainar (U. Wisc. Madison),
Iulian Neamtiu (U. Maryland, U.C. Riverside)

Adjusted by
Maria Christakis
Concurrent Programming is HARD

- 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
Main Takeaways

- You can find and reproduce Heisenbugs
  - new automatic tool called CHESS
  - for Win32 and .NET

- CHESS used extensively inside Microsoft
  - Parallel Computing Platform (PCP)
  - Singularity
  - Dryad/Cosmos

- Released by DevLabs
CHESS in a nutshell

- CHESS is a user-mode scheduler
  - Controls all scheduling nondeterminism

- Guarantees:
  - Every program run takes a different thread interleaving
  - Reproduce the interleaving for every run

- Provides monitors for analyzing each execution
CHESS Architecture

- Unmanaged Program
  - Win32 Wrappers
  - Windows

- Managed Program
  - .NET Wrappers
  - CLR

- Concurrency Analysis
  - Monitors

- CHESS Exploration Engine

- CHESS Scheduler

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

- Ability to explore all interleavings
  - Need to understand complex concurrency APIs (Win32, System.Threading)
  - Threads, threadpools, locks, semaphores, async I/O, APCs, timers, ...

- Does not introduce false behaviors
  - Any interleaving produced by CHESS is possible on the real scheduler
CHESS: Find and Reproduce Heisenbugs

**CHESS** runs the scenario in a loop
- Every run takes a different interleaving
- Every run is repeatable

Uses the CHESS scheduler
- To control and direct interleavings

Detect
- Assertion violations
- Deadlocks
- Dataraces
- Livelocks

**Program**
\[
\text{While(not done)} \{
\text{TestScenario()} \{
\ldots
\}
\}
\]

**Kernel:**
- Threads, Scheduler, Synchronization Objects
The Design Space for CHESS

- **Scale**
  - Apply to large programs

- **Precision**
  - Any error found by CHESS is possible in the wild
  - CHESS should not introduce any new behaviors

- **Coverage**
  - Any error found in the wild can be found by CHESS
  - Capture all sources of nondeterminism
  - Exhaustively explore the nondeterminism
CHESS Scheduler
Concurrent Executions are Nondeterministic

Thread 1
x = 1;
y = 1;

Thread 2
x = 2;
y = 2;
High level goals of the scheduler

- Enable CHESS on real-world applications
  - IE, Firefox, Office, Apache, ...

- Capture all sources of nondeterminism
  - Required for reliably reproducing errors

- Ability to explore these nondeterministic choices
  - Required for finding errors
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
Sources of Nondeterminism

3. Memory Model Effects

- Hardware relaxations
  - The processor can reorder memory instructions
  - Can potentially introduce new behavior in a concurrent program

- Compiler relaxations
  - Compiler can reorder memory instructions
  - Can potentially introduce new behavior in a concurrent program (with data races)
Sources of Nondeterminism

3. Memory Model Effects

- Hardware relaxations
  - The processor can reorder memory instructions
  - Can potentially introduce new behavior in a concurrent program
  - CHESS contains a monitor for detecting such relaxations

- Compiler relaxations
  - Compiler can reorder memory instructions
  - Can potentially introduce new behavior in a concurrent program (with data races)
  - Future Work
Interleaving Nondeterminism: Example

**Deposit Thread**

```c
void Deposit100()
{
    EnterCriticalSection(&cs);
    balance += 100;
    LeaveCriticalSection(&cs);
}
```

**Withdraw Thread**

```c
void Withdraw100()
{
    int t;
    EnterCriticalSection(&cs);
    t = balance;
    LeaveCriticalSection(&cs);
    EnterCriticalSection(&cs);
    balance = t - 100;
    LeaveCriticalSection(&cs);
}
```

**init:**

```c
balance = 100;
```

**final:**

```c
assert(balance = 100);
```
Invoke the Scheduler at Preemption Points

Deposit Thread

```c
void Deposit100()
{
    ChessSchedule();
    EnterCriticalSection(&cs);
    balance += 100;
    ChessSchedule();
    LeaveCriticalSection(&cs);
}
```

Withdraw Thread

```c
void Withdraw100()
{
    int t;
    ChessSchedule();
    EnterCriticalSection(&cs);
    t = balance;
    ChessSchedule();
    LeaveCriticalSection(&cs);

    ChessSchedule();
    EnterCriticalSection(&cs);
    balance = t - 100;
    ChessSchedule();
    LeaveCriticalSection(&cs);
}
```
Introducing Unpredictable Delays

Deposit Thread

```c
void Deposit100(){
    Sleep( rand() );
    EnterCriticalSection(&cs);
    balance += 100;
    Sleep( rand() );
    LeaveCriticalSection(&cs);
}
```

Withdraw Thread

```c
void Withdraw100(){
    int t;
    Sleep( rand() );
    EnterCriticalSection(&cs);
    t = balance;
    Sleep( rand() );
    LeaveCriticalSection(&cs);
    Sleep( rand() );
    EnterCriticalSection(&cs);
    balance = t - 100;
    Sleep( rand() );
    LeaveCriticalSection(&cs);
}
```
Introduce Predictable Delays with Additional Synchronization

Deposit Thread

```c
void Deposit100()
{
    WaitEvent(e1);
    EnterCriticalSection(&cs);
balance += 100;
    LeaveCriticalSection(&cs);
    SetEvent(e2);
}
```

Withdraw Thread

```c
void Withdraw100()
{
    int t;
    EnterCriticalSection(&cs);
t = balance;
LeaveCriticalSection(&cs);
    SetEvent(e1);
    WaitEvent(e2);
    EnterCriticalSection(&cs);
balance = t - 100;
LeaveCriticalSection(&cs);
}
```
Blindly Inserting Synchronization Can Cause Deadlocks

Deposit Thread

```c
void Deposit100()
{
    EnterCriticalSection(&cs);
    balance += 100;
    WaitEvent( e1 );
    LeaveCriticalSection(&cs);
}
```

Withdraw Thread

```c
void Withdraw100()
{
    int t;
    EnterCriticalSection(&cs);
    t = balance;
    LeaveCriticalSection(&cs);
    SetEvent( e1 );
    EnterCriticalSection(&cs);
    balance = t - 100;
    LeaveCriticalSection(&cs);
}
```
CHESS Scheduler Basics

- Introduce an event per thread
- Every thread blocks on its event
- The scheduler wakes one thread at a time by enabling the corresponding event
- The scheduler does not wake up a disabled thread
  - Need to know when a thread can make progress
  - Wrappers for synchronization provide this information
- The scheduler has to pick one of the enabled threads
  - The exploration engine decides for the scheduler
CHESS Algorithms
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 of 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

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

preemption

non-preemption
Polynomial state space

- Terminating program with fixed inputs and deterministic threads
  - \( n \) threads, \( k \) steps each, \( c \) preemptions
- Number of executions \( \leq \binom{n k}{c} \cdot (n+c)! \)
  \[ = O\left( (n^2k)^c \cdot n! \right) \]

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
Concurrent programs have cyclic state spaces

Thread 1

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

Thread 2

M1: done = 1;

! done

L1

! done

L2

done

L1

L2
A demonic scheduler unrolls any cycle ad-infinitum

Thread 1

while( ! done)  
{  
    Sleep();  
}  

done = 1;

Thread 2

! done

! done

done

! done

done
Depth bounding

- 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
int temp = done;
while( ! temp )
{
    Sleep();
}
done = 1;
```
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 scheduler

- Avoid unrolling unfair cycles
  - Effective state coverage
- Detect fair cycles
  - Find livelocks

Win32 API

Test Harness
Concurrent Program
Fair Demonic Scheduler
• What notion of “fairness” do we use?
Weak fairness

- A thread that remains enabled should eventually be scheduled

- A weakly-fair scheduler will eventually schedule Thread 2

- Example: round-robin

```
while(!done)
{
    Sleep();
}
done = 1;
```
Weak fairness does not suffice

Thread 1

Lock(l);
While(!done)
{
    Unlock(l);
    Sleep();
    Lock(l);
}
Unlock(l);

Thread 2

Lock(l);

done = 1;
Unlock(l);

en = \{T1, T2\}
T1: Sleep()
T2: Lock(l)

en = \{T1, T2\}
T1: Lock(l)
T2: Lock(l)

en = \{T1\}
T1: Unlock(l)
T2: Lock(l)

en = \{T1, T2\}
T1: Sleep()
T2: Lock(l)
Strong Fairness

- A thread that is enabled infinitely often is scheduled infinitely often

Thread 1

```c
Lock(l);
While( ! done) {
    Unlock(l);
    Sleep();
    Lock(l);
}
Unlock(l);
```

Thread 2

```c
Lock(l);
done = 1;
Unlock(l);
```

- Thread 2 is enabled and competes for the lock infinitely often
Implementing a strongly-fair scheduler

- A round-robin scheduler with priorities

- Operating system schedulers
  - Priority boosting of threads
We also need to be demonic

- Cannot generate all fair schedules
  - There are infinitely many, even for simple programs

- It is sufficient to generate enough fair schedules to
  - Explore all states (safety coverage)
  - Explore at least one fair cycle, if any (livelock coverage)
(Good) Programs indicate lack of progress

- Good Samaritan assumption:
  - A thread when scheduled infinitely often yields the processor infinitely often

- Examples of yield:
  - Sleep()
  - Blocking on synchronization operation
  - Thread completion
Fair demonic scheduler

- Maintain a priority-order (a partial-order) on threads
  - \( t < u \) : \( t \) will not be scheduled when \( u \) is enabled

- Threads get a lower priority only when they yield
  - When \( t \) yields, add \( t < u \) if
    - Thread \( u \) was continuously enabled since last yield of \( t \), or
    - Thread \( u \) was disabled by \( t \) since the last yield of \( t \)

- A thread loses its priority once it executes
  - Remove all edges \( t < u \) when \( u \) executes
Data Races
What is a Data Race?

- If two *conflicting* memory accesses happen *concurrently*, we have a data race.
- Two memory accesses *conflict* if
  - They target the same location
  - They are not both reads
  - They are not both synchronization operations

- Best practice: write “correctly synchronized” programs that do not contain data races.
What Makes Data Races significant?

• Data races may reveal synchronization errors
  • Most typically, programmer forgot to take a lock, or declare a variable volatile.

• Race-free programs are easier to verify
  • If a program is race-free, it is enough to consider schedules that preempt on synchronizations only
  • CHESS heavily relies on this reduction
How do we find races?

- Remember: races are concurrent conflicting accesses.
- But what does concurrent actually mean?
- Two general approaches to do race-detection

**Lockset-Based** (heuristic)
Concurrent ≈ “Disjoint locksets”

**Happens-Before-Based** (precise)
Concurrent = “Not ordered by happens-before”
This C# code contains **neither locks nor a data race**:

```csharp
int data;
volatile bool flag;

Thread 1

data = 1;
flag = true;

Thread 2

while (!flag)
yield();
it x = data;
```

- CHESS is *precise*: does not report this as a race. But *does* report a race if you remove the ‘volatile’ qualifier.
Happens-Before Order  [Lamport]

- Use **logical clocks** and **timestamps** to define a partial order called *happens-before* on events in a concurrent system.
- States *precisely* when two events are *logically* concurrent (abstracting away real time).

- Cross-edges from send events to receive events.
- \((a_1, a_2, a_3)\) happens before \((b_1, b_2, b_3)\) iff \(a_1 \leq b_1\) and \(a_2 \leq b_2\) and \(a_3 \leq b_3\). 

![Diagram of Happens-Before Order]
Happens-Before for Shared Memory

- **Distributed Systems:** Cross-edges from send to receive events

- **Shared Memory systems:** Cross-edges represent ordering effect of synchronization
  - Edges from lock release to subsequent lock acquire
  - Edges from volatile writes to subsequent volatile reads
  - Long list of primitives that may create edges
    - Semaphores
    - Waithandles
    - Rendezvous
    - System calls (asynchronous IO)
    - Etc.
### Example

<table>
<thead>
<tr>
<th>Static Program</th>
<th>Dynamic Execution Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>int data;</td>
<td>!flag) --&gt; true</td>
</tr>
<tr>
<td>volatile bool flag;</td>
<td>yield()</td>
</tr>
<tr>
<td>Thread 1</td>
<td>flag = true</td>
</tr>
<tr>
<td>data = 1;</td>
<td>!flag) --&gt; false</td>
</tr>
<tr>
<td>flag = true;</td>
<td>x = data</td>
</tr>
<tr>
<td>Thread 2</td>
<td></td>
</tr>
<tr>
<td>int x = data;</td>
<td>Code Execution Trace</td>
</tr>
</tbody>
</table>

- Not a data race because \((1,0) \leq (2,4)\)
- If flag were not declared volatile, we would not add a cross-edge, and this would be a data race.
Refinement Checking
Concurrent Data Types

• Frequently used building blocks for parallel or concurrent applications.

• Typical examples:
  • Concurrent stack
  • Concurrent queue
  • Concurrent deque
  • Concurrent hashtable
  • ....

• Many slightly different scenarios, implementations, and operations
Correctness Criteria

• Say we are verifying concurrent X (for $X \in$ queue, stack, deque, hashtable ...)

• Typically, concurrent X is expected to behave like atomically interleaved sequential X

• We can check this without knowing the semantics of X
Observation Enumeration Method
[CheckFence, PLDI07]

• Given concurrent test:

• (Step 1 : Enumerate Observations)
  Enumerate coarse-grained interleavings and record observations
  1. \( b_1 = \text{true} \quad i_1 = 1 \quad b_2 = \text{false} \quad i_2 = 0 \)
  2. \( b_1 = \text{false} \quad i_1 = 0 \quad b_2 = \text{true} \quad i_2 = 1 \)
  3. \( b_1 = \text{false} \quad i_1 = 0 \quad b_2 = \text{false} \quad i_2 = 0 \)

• (Step 2 : Check Observations)
  Check refinement: all concurrent executions must look like one of the recorded observations

Stack \( s = \text{new ConcurrentStack}() \);  \[ \begin{align*}
  s.\text{Push}(1); & \quad b_1 = s.\text{Pop}(\text{out} \; i_1); \\
  b_2 = s.\text{Pop}(\text{out} \; i_2); & 
\end{align*} \]
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