Concurrency Unit Testing

Shaz Qadeer
Microsoft Research
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
Concurrency unit testing (I)

• Concurrency unit test
  – initializes the module
  – creates few threads each performing a few operations
• Systematically explore all interactions of these threads
• Small-scope hypothesis
  – bugs manifest in corner cases of relatively small scenarios
  – validated by years of research in model checking
Concurrent unit testing (II)

- **Dynamic**
  - Concrete initial state and thread inputs
  - Concrete runtime (libraries/OS/hardware)
  - Execute program along many different schedules

- **Static**
  - Symbolic initial state and thread inputs
  - Abstract model of runtime (libraries/OS/hardware)
  - Represent behavior for many different schedules as a single logical formula
  - Solve formula using an automated theorem prover
CHESS

Joint work with

Tom Ball
Sebastian Burckhardt
Madan Musuvathi
CHESS in a nutshell

- 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 monitors

• 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
• Dataraces, memory model races
CHESS scheduler

• 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
CHESS synchronization wrappers

• Understand the semantics of synchronizations

• Provide enabled information

• Expose nondeterministic choices
  – An asynchronous ReadFile can possibly return synchronously
State space explosion

- Number of executions \( = O(n^{nk}) \)
- Exponential in both \( n \) and \( k \)
  - Typically: \( n < 10 \quad k > 100 \)
- 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

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

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

Exponential in n and c, but not in k

- Choose c preemption points
- Permute n+c atomic blocks
Advantages of preemption bounding

• Many errors are caused by few (<2) preemptions

• Generates an easy to understand error trace
  – Preemption points point to the root-cause of the bug

• A good coverage guarantee to the user
  – When CHESS finishes exploration with 2 preemptions, any remaining bug requires 3 preemptions or more

• Leads to the heuristic of preemption sealing
  – Seal preemptions in libraries to improve scalability
  – Seal preemptions to find multiple errors in single test run
CHESS status

• Publicly available at

• Basis for Concurrency Curriculum
  – Tom Ball (Microsoft Research)
  – Ganesh Gopalakrishnan (University of Utah)

• Being used by Microsoft product groups
  – Parallel Computing Platform
  – Windows Mobile

• Being used by other companies as well
STORM

Joint work with

Shuvendu Lahiri
Zvonimir Rakamaric
STORM in a nutshell

Maximum number of contexts per thread

- Concurrent C program
- HAVOC
- Concurrent Boogie program
- Concurrent → Sequential
- Sequential Boogie program
- Boogie/Z3
- OK
- BUG
Concurrent C program → Concurrent Boogie program

• Systems code written in C is messy
  – Heap, structures, fields
  – Pointers, pointer arithmetic, internal pointers
  – Dynamic memory allocation
  – Casts

• Boogie programs only have scalars and maps
  – easier to convert into formulas
Concurrent Boogie program $\rightarrow$
Sequential Boogie program

N contexts per thread, shared variable $g$

$T_1 \ || \ T_2$

assert $F$

\[
\begin{align*}
\text{int } & g_1, g_2, \ldots, g_N, v_1, \ldots, v_N; \\
\text{int } & k := 1; \\
\text{assume } & (g_1 = v_1 \land \ldots \land g_N = v_N);
\end{align*}
\]

\[st \rightarrow \text{switch}(k):\]
\[
\begin{align*}
\text{case } & 1: \text{Schedule; } st[g_1/g]; \\
\text{case } & 2: \text{Schedule; } st[g_2/g]; \\
\text{...}
\end{align*}
\]

\[\text{assert } F\]

\[\text{INIT;}
\]

\[L_1: T_1^s;\]

\[L_2: T_2^s;\]

\[L_3: \text{END;}\]

\[\text{assume } (g_1 = v_2 \land \ldots \land g_2 = v_3 \ldots);\]

\[\text{Schedule}\]
\[
\begin{align*}
\text{if } & (*) \{k++;\} \\
\text{if } & (k > N) \{k := 1; \text{goto } L_{i+1}\};
\end{align*}
\]
STORM applications

• WDM/KMDF device drivers
  – Correctness of IRP processing

• USB 3.0 device driver stack
  – Co-verification of driver against model of XHCI host controller