RICE

A Framework for Testing Concurrent Programs

PhD Thesis Defense

Mathias Ricken

Rice University

January 10, 2011

Concurrency in Practice

Brian Goetz, *Java Concurrency in Practice*, Addison-Wesley, 2006

Concurrency Practiced Badly

Concurrent programming is difficult and not well supported by today's tools. This framework simplifies the task of developing and debugging concurrent programs.

Contributions

1. Improved JUnit Framework
2. Execution with Random Delays
3. Additional Tools for Testing
    a. Invariant Checker
    b. Execution Logger
4. Miscellaneous

Unit Tests…

- Occur early
- Automate testing
- Keep the shared repository clean
- Serve as documentation
- Prevent bugs from reoccurring
- Allow safe refactoring

- Unfortunately not effective with multiple threads of control

RICE

Improvements to JUnit

Existing Testing Frameworks

- JUnit, TestNG

- Don't detect test failures in child threads
- Don't ensure that child threads terminate

- Tests that should fail may succeed

ConcJUnit

- Replacement for JUnit
  - Backward compatible, just replace junit.jar file

1. Detects failures in all threads
2. Warns if child threads or tasks in the event thread outlive main thread
3. Warns if child threads are not joined

ConcJUnit Evaluation

- JFreeChart
  - All tests passed; tests are not concurrent
- DrJava: 900 unit tests
  - Passed: 880
  - No join: 1
  - Lucky: 18
  - Timeout: 1
  - Runtime overhead: ~1 percent

ConcJUnit Limitations

- Only checks chosen schedule
  - A different schedule may still fail

- Example:

```
Thread t = new Thread(…);
if (nondeterministic()) t.join();
```

*2

Execution with Random Delays

Why Is This Necessary?

- Nondeterminism
  - Tests may execute under different schedules, yielding different results
  - Example: nondeterministic join (see above)
  - Example: data race (multithreaded counter)

```
int counter = 0;
// in M threads concurrently
for(int i=0; i<N; ++i) { ++counter; }
// after join: counter == M*N?
```

Race-Free ≠ Deterministic

- Race-free programs can still be nondeterministic

```
final Object lock = new Object();
final Queue q = new ArrayList();

// in one thread
... synchronized(lock) { q.add(0); } ...
// in other thread
... synchronized(lock) { q.add(1); } ...

// after join: q = (0, 1) or (1, 0)?
```

Non-Determinism = Error?

- Depends on the computation
  - If the queue (see previous example) was to contain {0, 1} in any order, then no error
  - If the queue was to contain (0, 1) in order, then error
- A unit test should be deterministic
  - Schedule should be considered an input parameter
- Run test under all possible schedules?

*3

Intractability

- Comprehensive testing is intractable
- Number of schedules (*N*)
  - *t*: # of threads, *s*: # of slices per thread

$$N = \prod_{x=0}^{t-1} \binom{(t-x)s}{s} = \binom{ts}{s}\binom{(t-1)s}{s}\cdots\binom{2s}{s}\binom{s}{s}$$

$$= \frac{(ts)!}{(s!)^t}$$

- Can we still find many of the problems?

*4

Previous Work

ConTest (Edelstein 2002)

- Programs seeded with calls to `sleep`, `yield`, or `priority` methods
  - At shared memory accesses
  - At synchronization events
- At runtime, random or coverage-based decision to execute seeded instructions
- `sleep` performed best
- Problem: predates Java Memory Model (JMM), ignores volatile fields

Previous Work (2)

ConTest (Edelstein 2002)

- Also included a record-and-replay feature
- Problems
  - Recording perturbs actual execution
  - No guarantee that replay will execute under same schedule, particularly on multicore systems
  - Did not focus on record-and-replay in my work

Previous Work (3)

rsTest (Stoller 2002)

- Similar to ConTest, but fewer seeds
  - Better classification of shared objects
- “Probabilistic completeness”
  - Non-zero probability rsTest will exhibit a defect, even if the scheduler on the test system normally prohibits it from occurring

Previous Work (4)

rsTest (Stoller 2002)

- Problem: also predates the JMM, ignores volatile fields
- Assumes an “as-if-serial” execution
  – Probabilistic completeness does not hold with JMM and programs with data races

Goal for Concutest

- Execution with random delays
  - Similar to ConTest
  - Cover all events relevant to synchronization, as specified by the JMM, i.e. particularly volatile fields

Synchronization Points

- `Thread.start` (before or after)
- `Thread.exit` (after)
- `Thread.join` (before and after)
- `Object.notify/notifyAll` (before)
- `Object.wait` (before)
- `MONITORENTER` (before)
- `MONITOREXIT` (before)
- Synchronized methods changed to blocks
- Access to `volatile` fields (before)

Examples

- Often inspired by tests used in ConTest and rsTest papers

- Allows a qualitative comparison
- No quantitative comparison
  - ConTest and rsTest not available
  - Not enough information on tests to accurately re-implement them

ConTest Examples (1)

1. Race: Threads race to set a flag first

ConTest: 0% of runs without
20% of runs with sleep
0.3% of runs with yield

My results:

(quad core) 0% without, 33% sleep

(dual core) 0% without, 27% sleep

ConTest Examples (2)

2. Atomicity: Threads read and write shared data, operations not atomic

ConTest: 0% without, 80% sleep

My results:

(quad core) 6% without, 99% sleep

(dual core) 0% without, 99% sleep

ConTest Examples (3)

3. Uninitialized data: Threads may run after notify, before data is initialized

ConTest: 0% without, 35% sleep/yield
("about 700x in 2000 tests")

My results:

(quad core) 0% without, 97% sleep

(dual core) 0% without, 93% sleep

rsTest Examples (1)

4. NASA Remote Agent: Deadlock if context switch after conditional, before wait

rsTest: 0% without, 100% (?) sleep
"Observed after 0.5 seconds"

My results:

(quad core) 7% without, 99% sleep

(dual core) 0% without, 99% sleep

rsTest Examples (2)

5. Atomicity: Threads read and write shared data, operations not atomic

rsTest: 0% without, 100% (?) sleep/yield
"many times in each run"

My results:
(quad core) 6% without, 99% sleep
(dual core) 0% without, 99% sleep

Analysis

- Concutest seems to perform just as well as ConTest and rsTest
- In my results, bugs are sometimes observed without sleeps/yields
  - Tested on dual core/quad core
  - Enhanced visibility of bugs, compared to single core?

Program Restrictions

- Some restrictions are useful
  - Minor inconvenience for programmer
    - e.g. must join child threads in some way
  - Major benefits for testing framework
    - e.g. don’t need to simulate child threads outliving the test
    - Reduces number of possible schedules

Restrictions: ConcJUnit

- Child threads must be joined
  - Only way to ensure that all errors are detected
- Slight inconvenience
  - Keep track of child threads when they are created
- ConcJUnit provides utilities for this

Restrictions: Shared Data

- Shared variables must be either
  - consistently protected by a lock, or
  - volatile, or
  - final

- This can be checked using a race detector (e.g. FastTrack, Flanagan 2009)

Restrictions: Volatile

- Specify which volatile variables should be instrumented with random delays
  a. Manually
  b. Use static "may happen in parallel" (MHP) analysis (e.g. Soot MHP, Li 2005)

Restrictions: Volatile (2)

- In most cases, we only need to focus on volatile variables in the application program
  - Test libraries separately
  - Then assume libraries are correct
  - Encode invariants and check for violations (see Invariant Checker contribution)
- Listing volatile variables to be instrumented is possible and not prohibitive

RICE

Additional Tools for Testing

Additional Tools for Testing

1. Annotations for Invariant Checking
   - Runtime warning if invariants for a method are not maintained
   - ~1% slowdown during testing, no slowdown during normal execution

2. Annotations for Execution Logging
   - Tests properly decoupled from application code
   - No slowdown compared to hand-written logging

Summary

Summary

1. Improved JUnit Framework

- Detects errors in all threads
- Warns if child threads are still alive and errors could be missed
- Warns if child threads ended on time, but not because they were joined
- Low overhead (~1%)

→ Much more robust unit tests

Summary (2)

2. Execution with Random Delays
- Detects many types of concurrency defects
- Updated for the Java Memory Model (JMM)

→ Higher probability of finding defects usually obscured by scheduler

→ Programmer restrictions not prohibitive

Summary (3)

4. Additional Tools for Testing

- Invariant Checker encodes and checks method invariants
- Execution Logger decouples tests and application code
- Low overhead (~1%)

→ Simpler to write good tests

Summary (4)

5. Miscellaneous

- Subtyping for annotations useful, compatible with existing Java
- DrJava integration makes better tools available to beginners

This framework simplifies the task of developing and debugging concurrent programs.

Acknowledgements

I thank the following people for their support.

- My advisor
  - Corky Cartwright
- My committee members
  - Walid Taha
  - David Scott
  - Bill Scherer (MS)
- NSF, Texas ATP, Rice School of Engineering
  - For providing partial funding

Conclusion

This framework simplifies the task of developing and debugging concurrent programs.

Concutest is open source and available for Windows, Linux and Mac

http://www.concutest.org/

RICE

More Information on Additional Tools

RICE

Additional Tools for Testing: Annotations for Invariant Checking

Concurrency Invariants

- Methods have to be called in event thread
  - `TableModel`, `TreeModel`
- Method may not be called in event thread
  - `invokeAndWait()`
- Must acquire readers/writers lock before methods are called
  - `AbstractDocument`
  - DrJava's documents

Invariants Difficult to Determine

- May be found in
  - Javadoc comments
  - Only in internal comments
  - Whitepapers
- Often not documented at all
- Errors not immediately evident
- Impossible to check automatically

Invariant Annotations

- Add invariants as annotations

```
@NotEventThread
public static void
  invokeAndWait(Runnable r) { ... }
```

- Process class files
  - Find uses of annotations
  - Insert code to check invariants at method beginning

Advantages of Annotations

- Java language constructs
  - Syntax checked by compiler
- Easy to apply to part of the program
  - e.g. when compared to a type system change
- Light-weight
  - Negligible runtime impact if not debugging (only slightly bigger class files)
  - <1% when debugging
- Automatic Checking

RICE

Additional Tools for Testing: Annotations for Execution Logging

Need for Execution Logging

- Tests need to check if code was executed
- Implementation when no variable can be checked
  - Add flag to application code
  - Add flag to test code, add call from application code to test code
- Application and test code become tightly coupled

Logging Annotations

- Annotate test with methods that need to be logged

```
@Log(@TheMethod(c=Foo.class, m="bar"))
void testMethod() { … }
```

- Process class files
  - – Find methods mentioned in annotations
  - – Insert code to increment counter at method beginning

Logging Annotations (2)

- Decouples application code from test
- Annotations with subtyping useful for logging too

```
@Log(@And({
         @TheMethod(c=Foo.class, m="bar",
                    subClasses=true),
         @InFile("SomeFile.java")
}))
void testMethod() { … }
```

Log Benchmarks Setup

- Different implementation strategies
- Different numbers of threads (1-16)
- Three different benchmarks
  - Tight loop
  - Outer loop
  - DrJava
    - subclasses of GlobalModelTestCase
- Expressed as factor of execution time with hand-written logging or no logging
  - 1.0 = no change

Log Benchmarks Setup (2)

- Tight loop

```
for(i=0; i<N; ++i) { loggedMethod(); }
@LogThis void loggedMethod() {/*no op*/ }
```

- Outer loop

```
for(i=0; i<N; ++i) { loggedMethod(); }
@LogThis void loggedMethod() {
  for(j=0; i<M; ++j) { gaussianBlur(); }
}
```

Execution Log Benchmarks

Time vs Hand-Written: Logging Tight Loop (RHEL i7, 4core)

Time (Factor, compared to hand-written)

3.000
2.500
2.000
1.500
1.000
0.500
0.000

1
2
4
8
16

Number of Threads

Naïve
NonBlocking
PerThread
Fields
LocalFields

Execution Log Benchmarks

Time vs No Logging: Logging Tight Loop (RHEL i7, 4core)

Time (Factor, compared to no logging)

1.160
1.140
1.120
1.100
1.080
1.060
1.040
1.020
1.000
0.980
0.960

1 2 4 8 16

Number of Threads

Fields
LocalFields

Log Benchmark Results

- "Local fields" performs best
  - Generates code identical to hand-written
- Compared to hand-written logging
  - No slowdown
- Compared to no logging
  - 10% to 50% slowdown in tight loop
  - ~1% slowdown in outer loop
  - No slowdown in DrJava

Extra Slides

Sample JUnit Tests

```
public class Test extends TestCase {
  public void testException() {
    throw new RuntimeException("booh!");
  }
} public void testAssertion() {
    assertEquals(0, 1);
  }
}
```

Both tests fail.

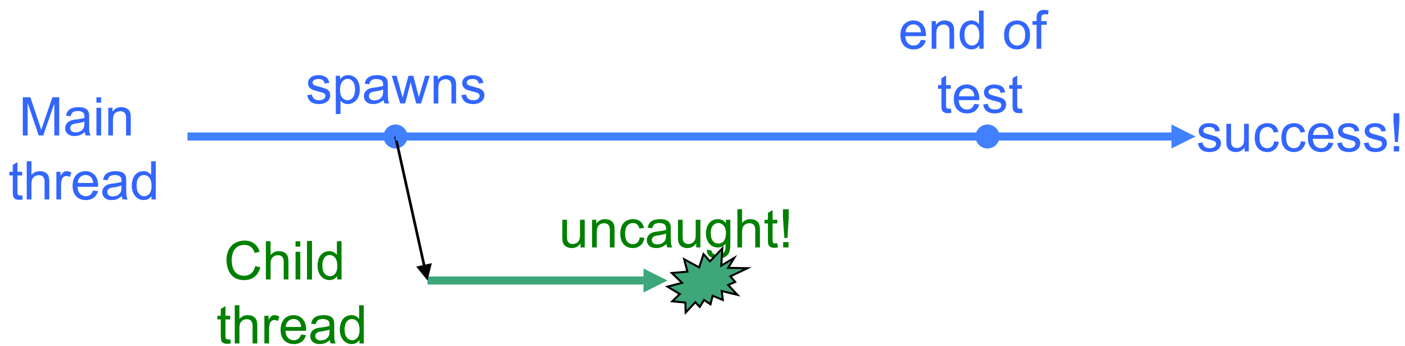
```
if (0!=1)
  throw new AssertionFailedError();
```

JUnit Test with Child Thread

```
public class Test extends TestCase {
  public void testException() {
    new Thread() {
      public void run() {
        throw new RuntimeException("booh!");
      }
    }.start();
  }
}
```

Main thread

Child thread



JUnit Test with Child Thread

```
public class Test extends TestCase {
  public void testException() {
    new Thread() {
      public void run() {
        throw new RuntimeException("booh!");
      }
    }.start();
  }
}
```

Uncaught exception, test should fail but does not!

- By default, no uncaught exception handler installed for child threads

Changes to JUnit (1 of 3)

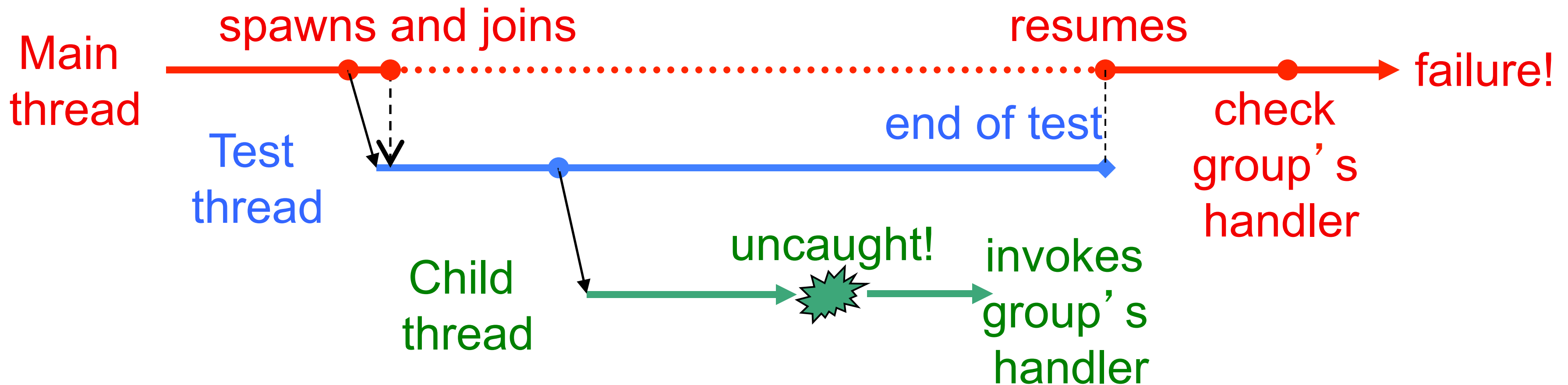
- Thread group with exception handler
  - JUnit test runs in a separate thread, not main thread
  - Child threads are created in same thread group
  - When test ends, check if handler was invoked

Reasoning:

- Uncaught exceptions in *all threads* must cause failure

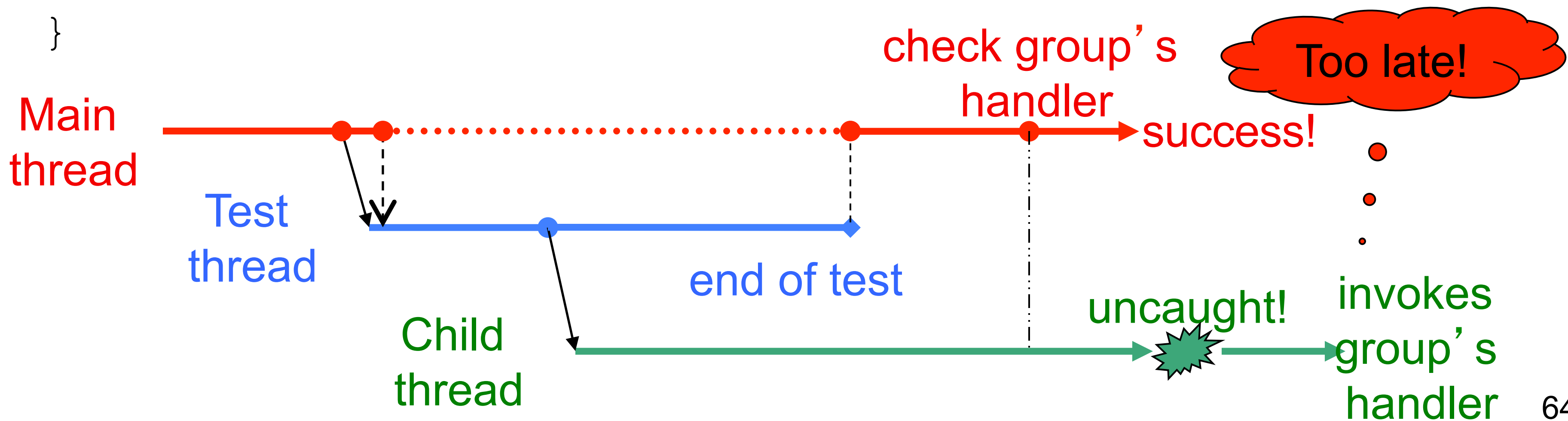
JUnit Test with Child Thread

```
public class Test extends TestCase {
  public void testException() {
    new Thread() {
      public void run() {
        throw new RuntimeException("booh!");
      }
    }.start();
  }
}
```



Child Thread Outlives Parent

```
public class Test extends TestCase {
  public void testException() {
    new Thread() {
      public void run() {
        throw new RuntimeException("booh!");
      }
    }.start();
  }
}
```



Changes to JUnit (2 of 3)

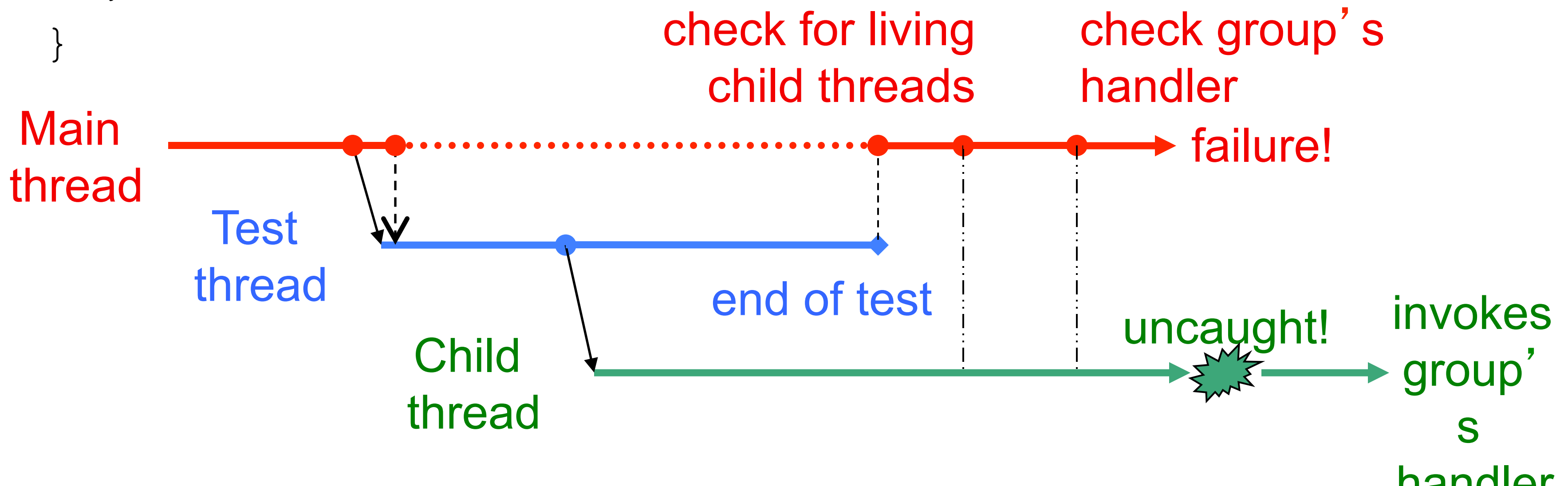
- Check for living child threads after test ends

Reasoning:

- Uncaught exceptions in *all threads* must cause failure
- If the test is declared a success before all child threads have ended, failures may go unnoticed
- Therefore, all child threads must terminate before test ends

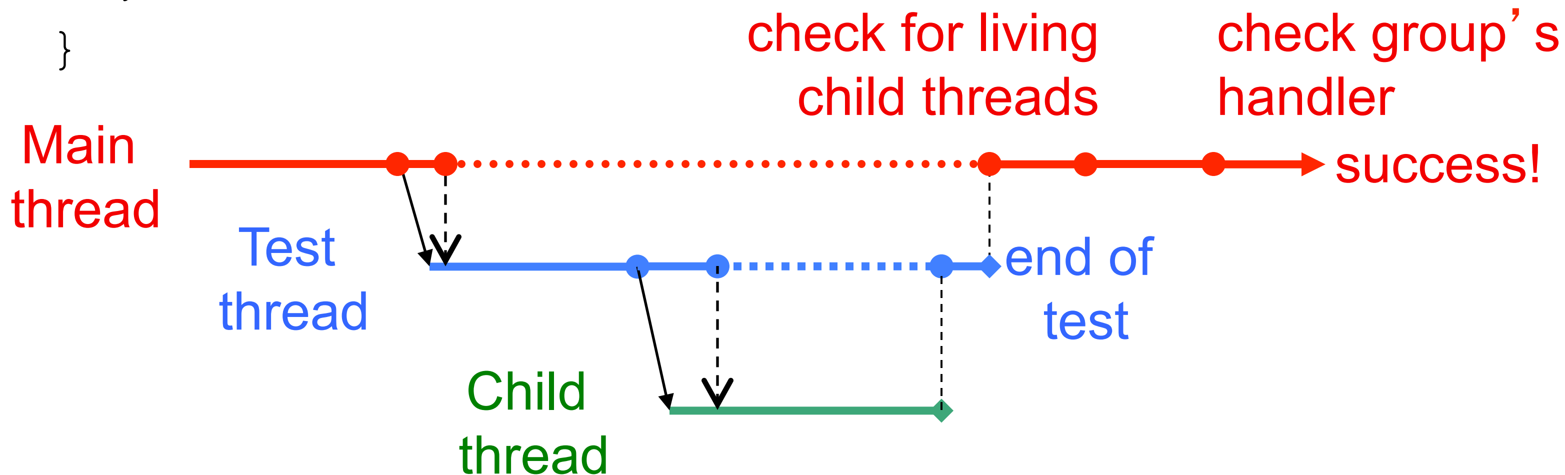
Check for Living Threads

```
public class Test extends TestCase {
  public void testException() {
    new Thread() {
      public void run() {
        throw new RuntimeException("booh!");
      }
    }.start();
  }
}
```



Correctly Written Test

```
public class Test extends TestCase {
  public void testException() {
    Thread t = new Thread() {
      public void run() { /* child thread */ }
    };
    t.start();
    t.join(); // wait until child thread has ended
  }
}
```



*4

Changes to JUnit (3 of 3)

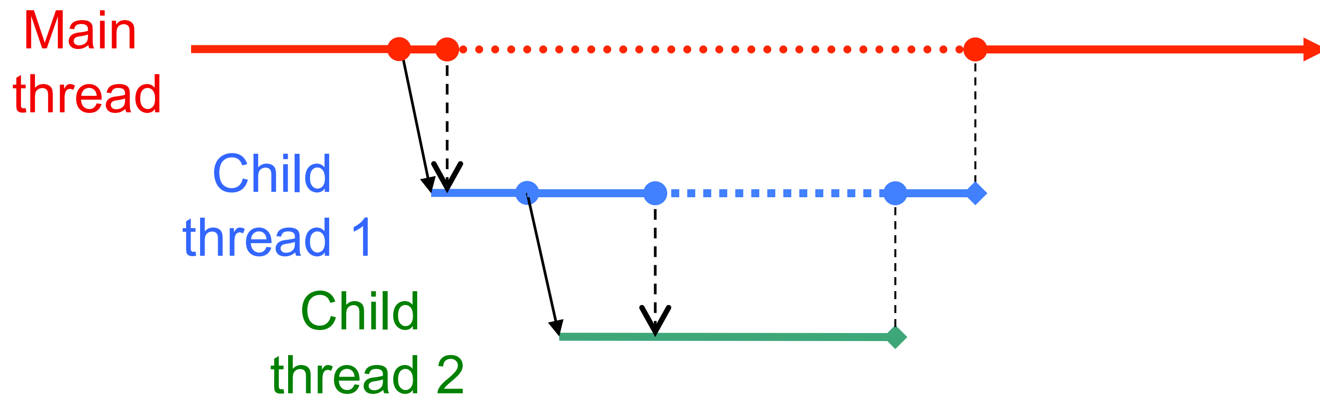
- Check if any child threads were not joined

Reasoning:

- All child threads must terminate before test ends
- Without `join()` operation, a test may get "lucky"
- Require all child threads to be joined

Fork/Join Model

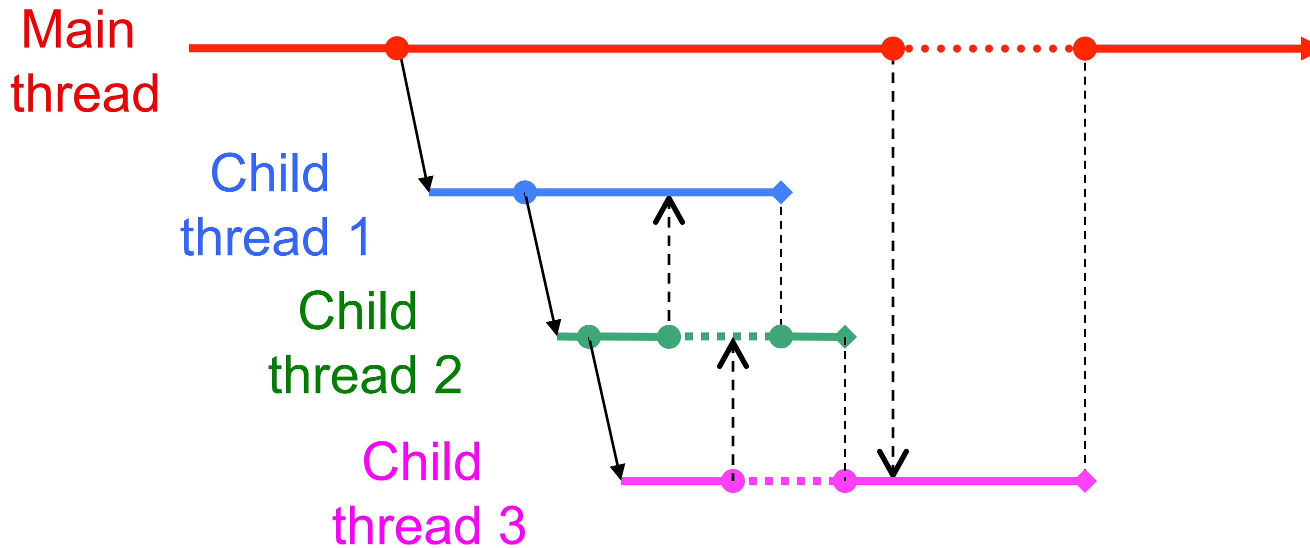
- Parent thread joins with each of its child threads



- May be too limited for a general-purpose programming language

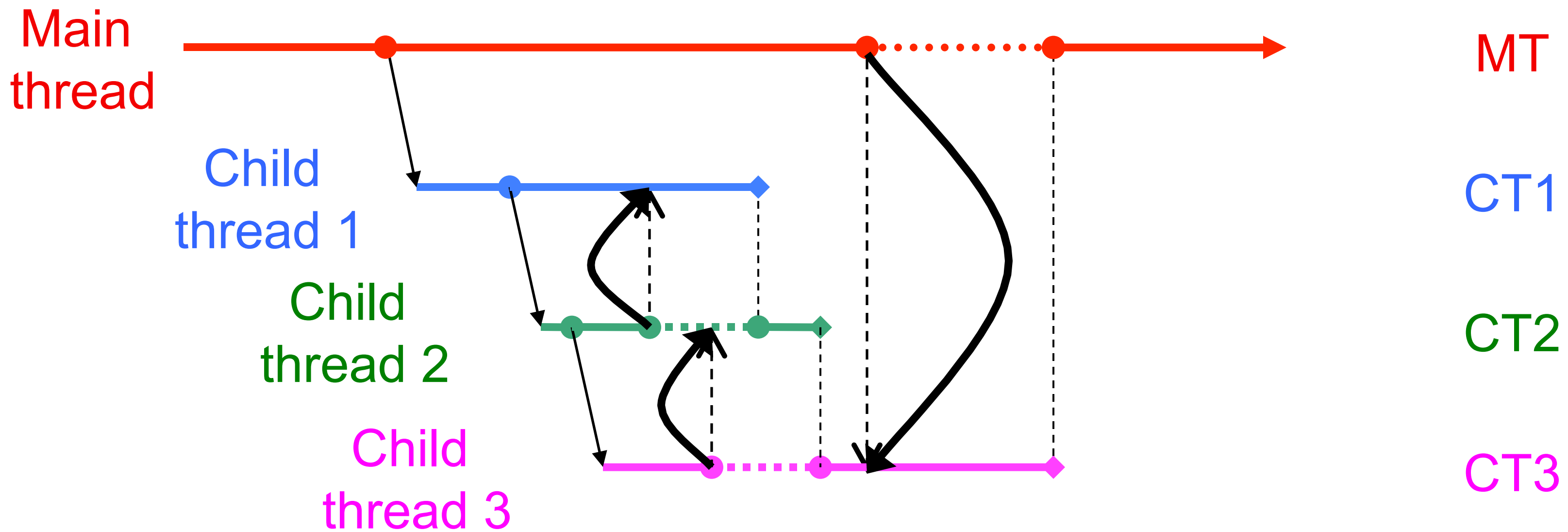
Other Join Model Examples

- Chain of child threads guaranteed to outlive parent
- Main thread joins with last thread of chain



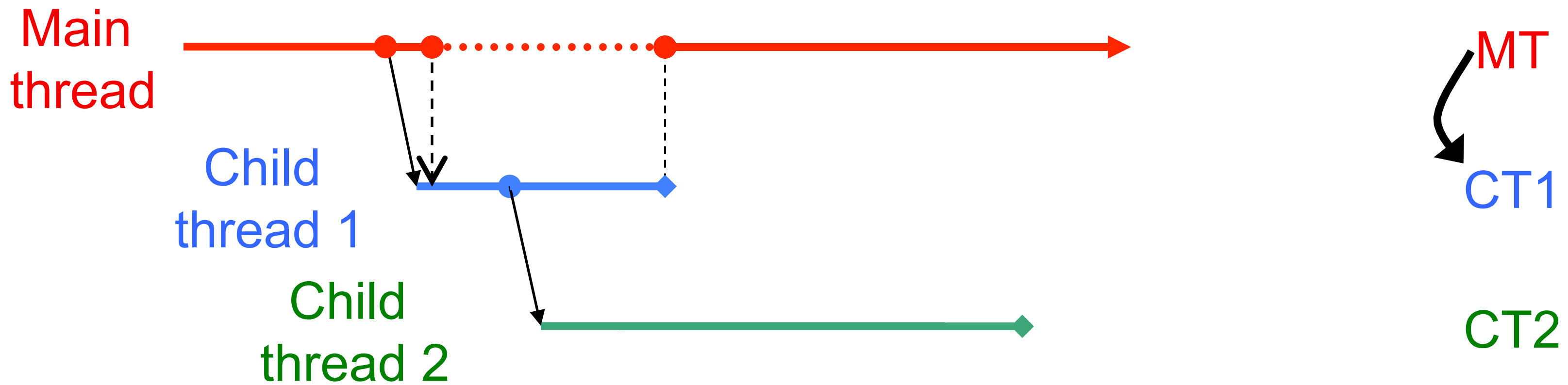
Generalize to Join Graph

- Threads as nodes; edges to joined thread
- Test is well-formed as long as all threads are reachable from main thread



Unreachable Nodes

- An unreachable node has not been joined
  - Child thread may outlive the test



Graph Construction: `start()`

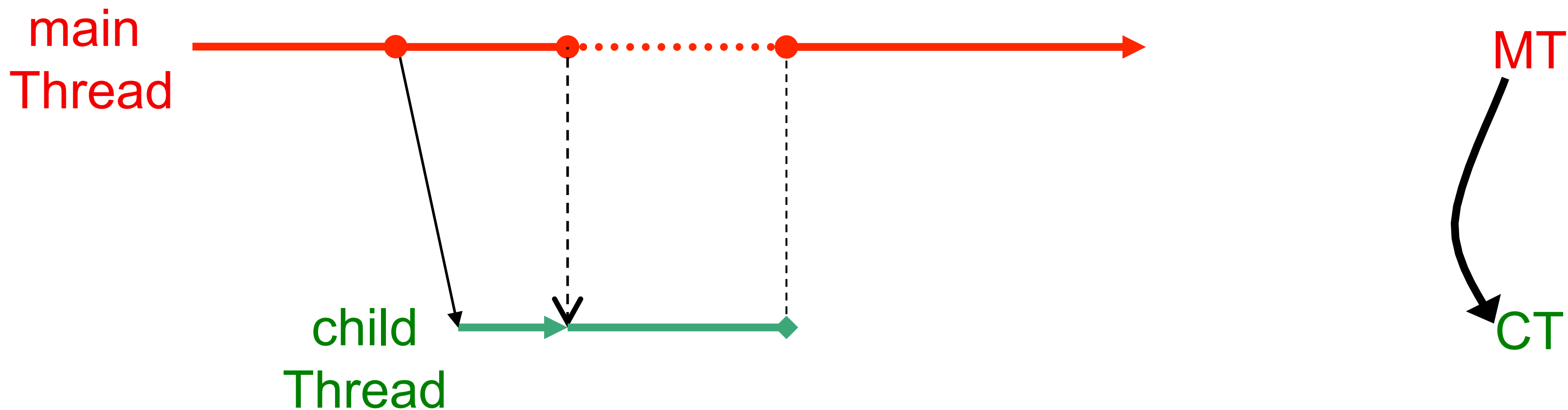
```
// in mainThread
childThread.start();
```

- Add node for `childThread`

Graph Construction: `join()`

```
// in mainThread
childThread.join();
```

- When leaving `join()`, add edge from `mainThread` to `childThread`



*1

Modifying the Java Runtime

- Changing `Thread.start()` and `join()`
  - Need to modify Java Runtime Library
  - Utility to process user’s rt.jar file
  - Put new jar file on boot classpath: `-Xbootclasspath/p:newrt.jar`
- Still works without modified `Thread` class
  - Just does not emit “lucky” warnings

Implementation

- `Thread` methods can be modified
  - Insert calls directly into Thread.class file

```
class Thread {
  public void start() {
    RandomDelay.threadStartDelay();
    // ...
  }
}
```

Implementation (2)

- `Object` methods may not be modified
  - New methods can be inserted
  - Replace calls with calls to wrapper methods

```
class Object {
  public void waitWrapper() {
    RandomDelay.objectWaitDelay();
    wait(); // call original
  }
}
foo.waitWrapper(); // used to be foo.wait
  ();
```

Implementation (3)

- `MONITORENTER`/`MONITOREXIT`
  - Insert calls before the instructions
- Synchronized methods
  - Convert to unsynchronized method with synchronized block
  - Add exception handler to mimic automatic release of lock
- Access to volatiles
  - Improvement using JSR/RET (JSR/RET deprecated)

Implementation (4)

- Access to volatile fields
  - Examine all `GETFIELD`/`PUTFIELD`/`GETSTATIC`/`PUTSTATIC` instructions
  - If referenced field is volatile, insert call before instruction
  - Improvement using `JSR`/`RET` to reduce number of method calls and code bloat
    - `JSR`/`RET` deprecated
    - Examine how `try-finally` will be compiled by future Java compilers
    - Not implemented yet

Interactions between Delays

- `wait`/`notify`
  - Delayed `wait` may cause a `notify` to be lost
  - Delayed `wait`/delayed `notify` may cancel each other out
- `MONITORENTER`
  - Delayed `MONITORENTER` in one thread may give other thread preference
  - Delayed `MONITORENTER` in all threads may cancel out
- etc.

Minimize Cancellations

- Strategies to minimize destructive interference
  - In one run, delay only `wait`; in the next run, delay only `notify`
  - In one run, delay only `MONITORENTER` in threads with even ID number; in the next run, delay in threads with odd ID
  - etc.
- Cancellation effects and delay lengths need more investigation

Benchmarks

- Performance impact still needs to be measured
- Right balance probably application-specific
  - More delays?
  - Faster execution?

Restrictions: Volatile (3)

- Soot MHP does not scale well (beyond toy examples)
  - A simpler “may be accessed in Runnable” (=child thread or event thread) analysis may be sufficient
- Did not implement and test this
  - Wanted to show that execution with random delays is effective, rather than improve an existing analysis

Limitations of Java Annotations

- Java does not allow the same annotation class to occur multiple times

```
@OnlyThreadWithName("foo")
@OnlyThreadWithName("bar") // error
void testMethod() { … }
```

- Conjunctions, disjunctions and negations?

Subtyping for Annotations

- Let annotation extend a supertype?

```
public @interface Invariant { }
public @interface OnlyThreadWithName
  extends Invariant { String name(); }
public @interface And extends Invariant {
  Invariant[] terms();
}
```

- Subtyping not allowed for annotations
  - Extended Annotations Java Compiler (xajavac)

Invariant Annotation Library

- `@EventThread`
- `@ThreadWithName`
- `@DistinctArguments`, `@SameArguments`
- `@SynchronizedThis`
- `@SynchronizedArgument`
- `@Not, @And, @Or`, etc.

- Subtyping reduced implementation size by a factor of 3 while making invariants more expressive

Java API Annotations

- Started to annotate Java API
  - 30 whole classes, 44 individual methods
- Community project at community.concutest.org
  - Suggest annotations and vote for them
  - Browse by class or annotation type
- Annotations can be extracted as XML
  - Share annotations
  - Add checks without needing source code

Logging Annotations

- Annotate test with methods that need to be logged

```
@Log(@TheMethod(c=Foo.class, m="bar"))
void testMethod() { … }
```

```
// "method literals" would be nice...
@Log(@TheMethod(Foo.bar.method))
void testMethod() { … }
```

Log Implementations

- Naïve
  - Single synchronized map (methods→counts)
- Non-blocking
  - Single non-blocking (unsynchronized) map (methods→counts)
  - Cliff Click's Highly Scalable Java

Log Implementations (2)

- Per-thread
  - Non-blocking map (threads→…) of unsynchronized maps (methods→counts)
    - First, look up by current thread
    - Then, look up by method
  - Inner map can be unsynchronized because they are thread-specific
  - Outer map is non-blocking because modifications are rare (only for new thread)

Log Implementations (3)

- Fields
  - Primitive `long` field for each logged method in the log class
  - Increment synchronized using log class

```
void foo() {
   synchronized(Log.class) { ++Log.fooCount; }
   // ...
}
```

Log Implementations (4)

- Local Fields
  - Primitive `long` field for each logged method in the class in which the method occurs
  - Increment synchronized by containing class
  - Equivalent to hand-written logging

```
public static volatile long fooCount = 0;
void foo() {
  synchronized(MyClass.class) { ++fooCount; }
  // ...
}
```

Log Implementation Notes

- "Naïve" easiest to implement
- "Fields" adds all fields to the log class
  - Easy to read
- "Local fields" most difficult to implement
  - Adds fields to all classes with logged methods
  - Fields are spread out, more difficult to read all counts to produce complete picture

Execution Log Benchmarks

Time vs Hand-Written: Logging Tight Loop (MBP i7, 2core)

Execution Log Benchmarks

Time vs No Logging: Logging Tight Loop (MBP i7, 2core)

Time (Factor, compared to no logging)

1.580
1.480
1.380
1.280
1.180
1.080
0.980

1 2 4 8 16

Number of Threads

Fields
LocalFields

Execution Log Benchmarks

Time vs No Logging: Logging Outer Loop (MBP i7, 2core)

Execution Log Benchmarks

Miscellaneous Contributions

Miscellaneous Contributions

- xajavac
  - Java Compiler with Extended Annotations (subtyping and multiple annotations)
- DrJava integration: make better tools available to beginners
  - ConcJUnit
  - xajavac
  - Invariant Checker and Execution Logger will be integrated soon

Notes

Notes (1)

1. Only add edge if joined thread is really dead; do not add if join ended spuriously. ←

```
public class Test extends TestCase {
  public void testException() {
    Thread t = new Thread(new Runnable() {
      public void run() {
        throw new RuntimeException("booh!");
      }
    });
    t.start();
    while(t.isAlive()) {
      try { t.join(); }
      catch(InterruptedException ie) { }
    }
} }
```

Loop since `join()` may end spuriously

Notes (2)

2. Also cannot detect uncaught exceptions in a program's uncaught exception handler (JLS limitation) ←
3. There are exceptions when a test may not have to be deterministic, but it should be probabilistic. Example: Data for some model is generated using a random number generator. ←

Notes (3)

3. Number of schedules, derived ←

$$N = \prod_{x=0}^{t-1} \binom{(t-x)s}{s}$$

Product of s-combinations:
For thread 1: choose s out of ts time slices
For thread 2: choose s out of ts-s time slices
…
For thread t-1: choose s out of 2s time slices
For thread t-1: choose s out of s time slices

$$= \binom{ts}{s}\binom{(t-1)s}{s}\cdots\binom{2s}{s}\binom{s}{s}$$

$$= \frac{(ts)!}{s!(ts-s)!}\frac{(ts-s)!}{s!(ts-2s)!}\cdots\frac{(2s)!}{s!s!}\frac{s!}{s!0!}$$

Writing s-combinations using factorial

$$= \frac{(ts)!}{s!}\frac{1}{s!}\cdots\frac{1}{s!}\frac{1}{s!}$$

Cancel out terms in denominator and next numerator

$$= \frac{(ts)!}{(s!)^t}$$

Left with (ts)! in numerator and t numerators with s!

Image Attribution

1. Image on Concurrency in Practice:
   Adapted from
   Brian Goetz et al. 2006, Addison Wesley
2. Image on Concurrency Practiced Badly:
   Caption Fridays