# A Framework for Testing Concurrent Programs

PhD Proposal

Mathias Ricken

Rice University

December 2, 2010

# 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. Program Restrictions to Simplify Testing
4. Additional Tools for Testing
   a. Invariant Checker
   b. Execution Logger
5. 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

## 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

MS

1. Detects failures in all threads

MS

PhD

2. Warns if child threads or tasks in the event thread outlive main thread

PhD

3. Warns if child threads are not joined

# Sample JUnit Tests

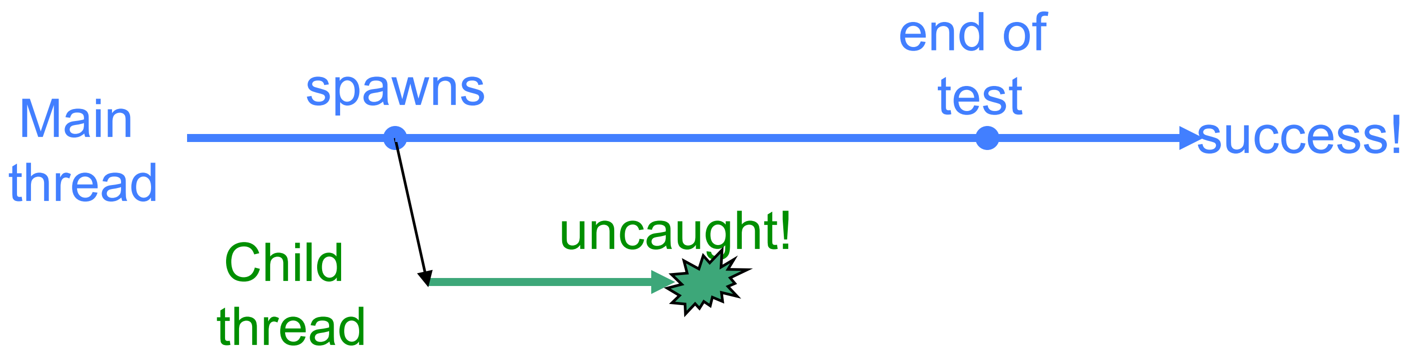
```
public class Test extends TestCase {
  public void testException() {
    throw new RuntimeException("booh!");
  }
} public void testAssertion() {
    assertEquals(0, 1);
  }
}
```

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();
  }
}
```



## Changes to JUnit

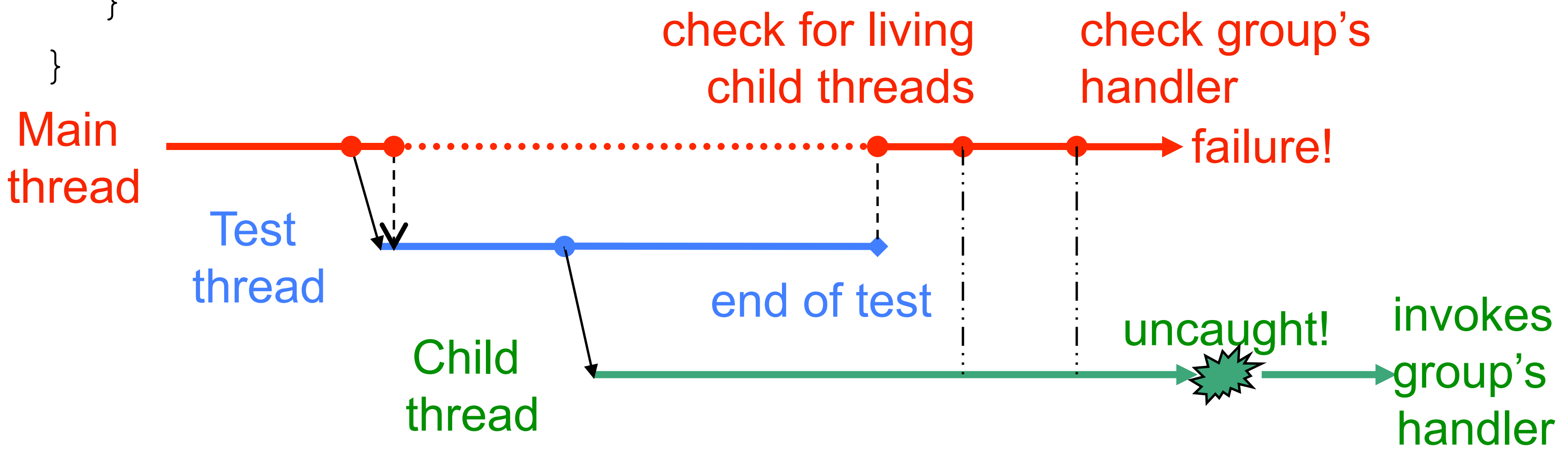
- Check for living child threads after test ends

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();
  }
}
```



## Changes to JUnit (2)

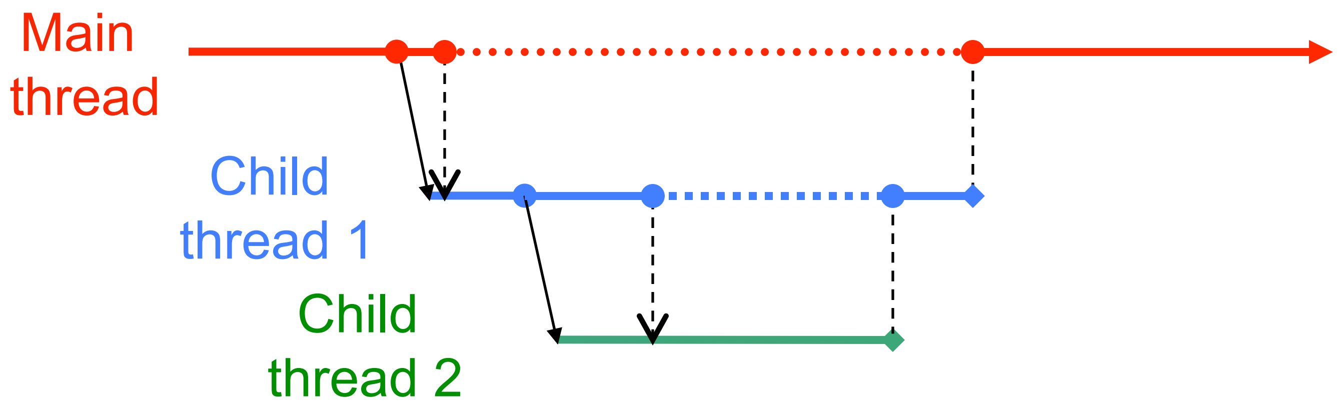
- Check if any child threads were not joined

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

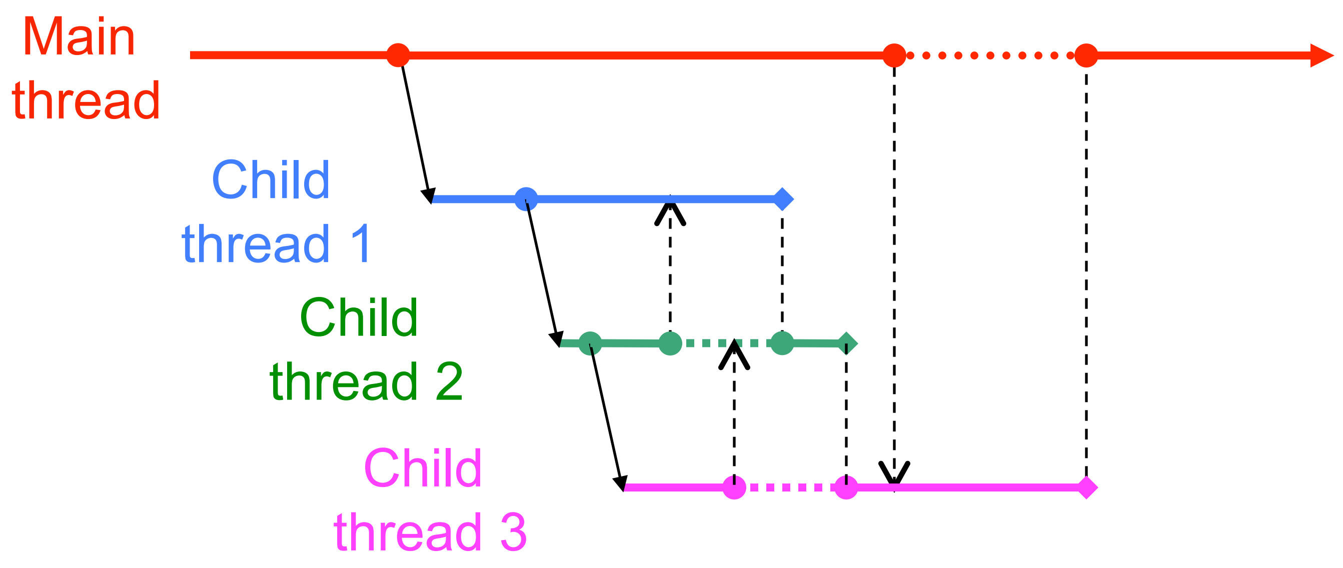
- Parent thread joins with each of its child threads



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

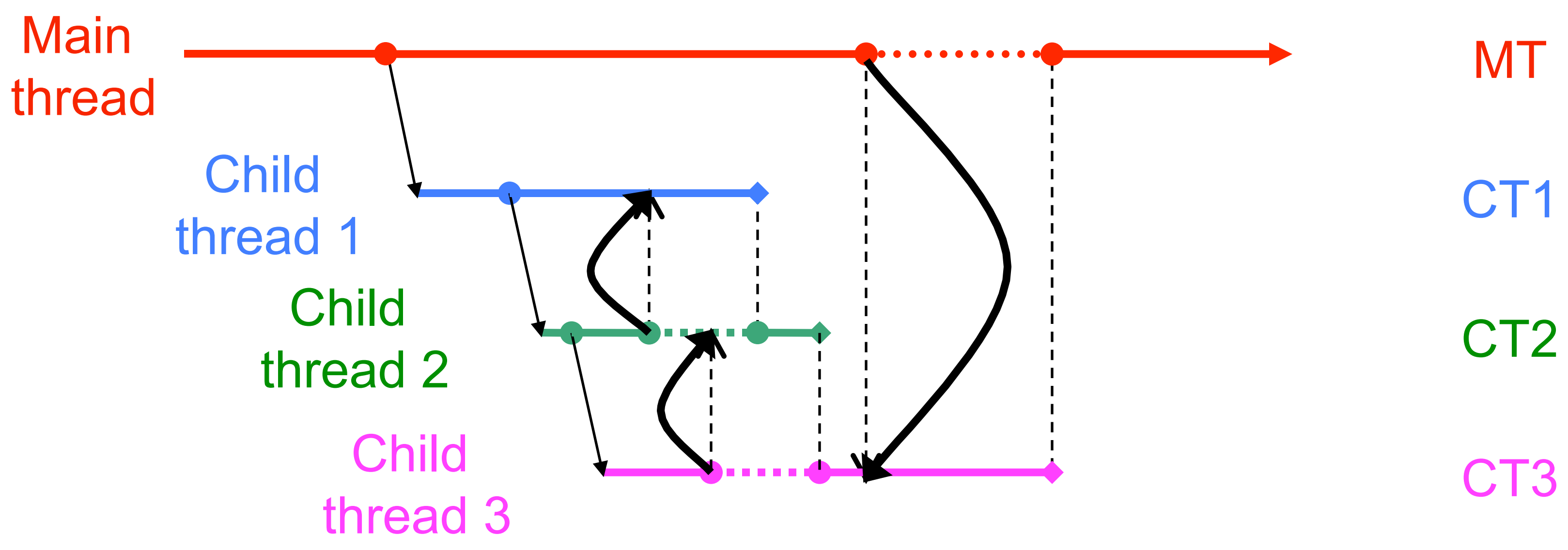
# Other Join Model Examples

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



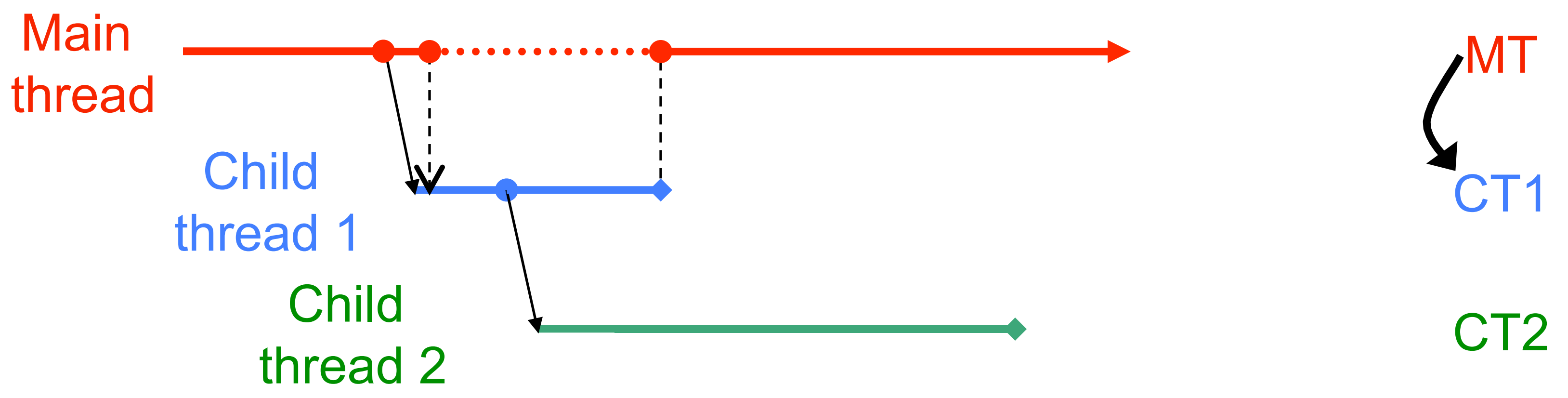
# Generalize to Join Graph

- 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



## 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)?
```

## Nondeterminism = 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 (with respect to thread scheduling)
  - 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

ConTest (Edelstein 2002)

- Programs seeded with calls to `sleep`, `yield`, or `priority` methods at synchronization events
- At runtime, random or coverage-based decision to execute seeded instructions
- `sleep` performed best
- Problem: predates Java Memory Model (JMM), does not treat volatile fields correctly

## Previous Work: rsTest

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
- Problem: also predates the JMM, does not treat volatile fields correctly

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

## Examples

- Multithreaded counter
  - If counter is `volatile`
- Multithreaded queue
- Early `notify`
- Missing `wait-notify` synchronization (assume another thread completed)

- Need more examples

## Benchmarks

- Still to do

## Program Restrictions to Simplify Testing

## 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

# Shared Variables

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

- This can be checked using a race detector (e.g. Chord, Naik 2006; FastTrack, Flanagan 2009)

## Volatile Variables

- Specify which volatile variables should be instrumented with random delays
    a. Manually (e.g. “in all user classes” or “in classes in package xyz”)
    b. Use static “may happen in parallel” (MHP) analysis (e.g. Soot MHP, Li 2005)

## Additional Tools for Testing

## Additional Tools for Testing

1. Annotations for Invariant Checking
   - Runtime warning if invariants for a method are not maintained
   - Annotations now support subtyping

MS

PhD

PhD

2. Annotations for Execution Logging

## 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

## 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

## 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`
- `@SynchronizedThis`
- `@Not, @And, @Or`
- etc.

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

## Additional Tools for Testing: Annotations for Execution Logging

## Need for Execution Logging

- Tests need to check if code was executed
- Implementation options 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() { … }
```

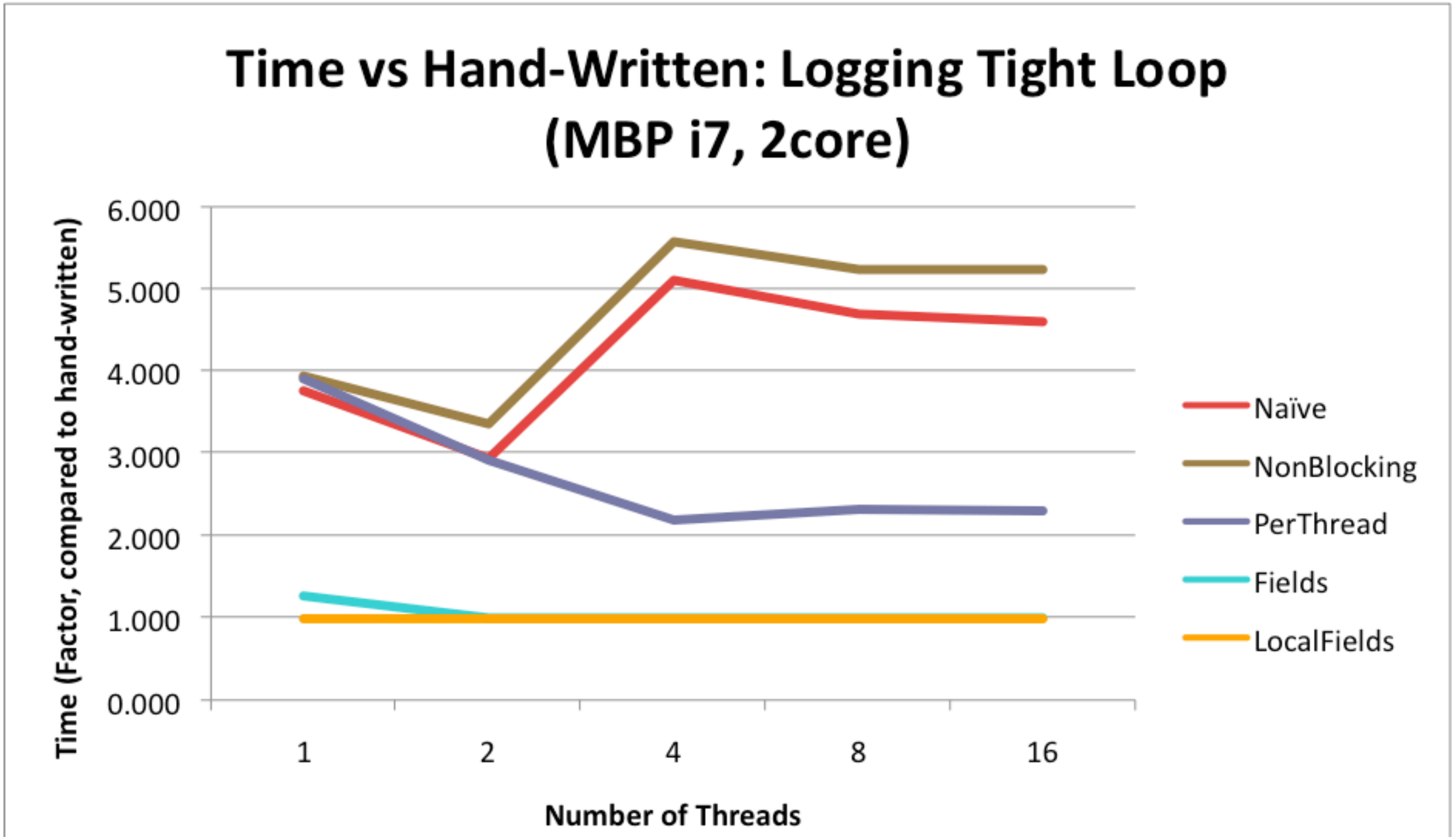
## Log Benchmarks Setup

- Different implementations
  - Naïve
  - Non-blocking
  - Per-thread
  - Fields
  - Local fields
- Different numbers of threads (1-16)

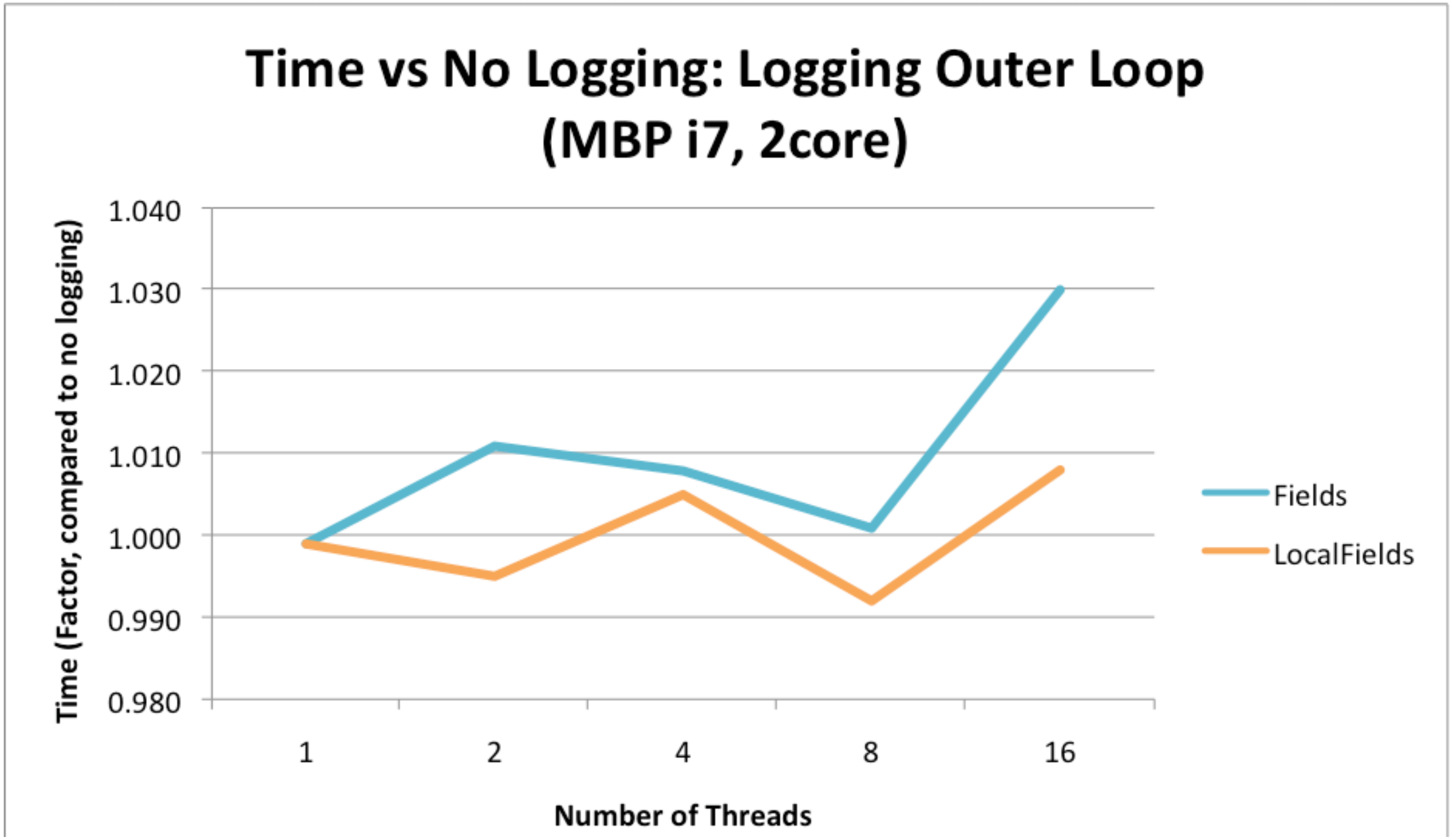
## Log Benchmarks Setup (2)

- 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

# Execution Log Benchmarks



# Execution Log Benchmarks



## Log Benchmark Results

- “Local fields” performs best
- Compared to hand-written logging
  - No slowdown
- Compared to no logging
  - 10% to 50% slowdown in tight loop
  - ~1% slowdown in outer loop
  - No measurable slowdown in DrJava

## 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

## Summary (3)

3. Program Restrictions to Simplify Testing

- Child threads in tests must be joined
- Shared variables must be consistently locked, volatile, or final
- Volatile variables to be instrumented must be listed

→ Restrictions are not prohibitive

## Summary (4)

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 (5)

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.

## Still To Do

- Execution with random delays
  - More examples
  - Benchmark
  - Evaluate choice of delay lengths
- Write, write, write

# 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

## 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!*

$$\prod^{t-1}\Big((t-x)s$$

## 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