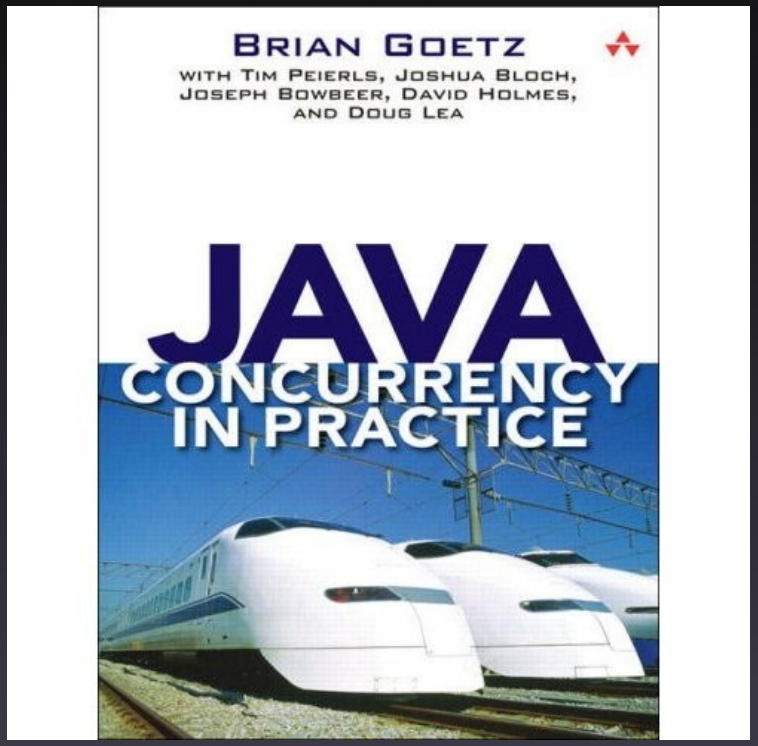
# Java Concurrency in practice




## Chapter: 12

Bjørn Christian Sebak (bse069@student.uib.no)

Karianne Berg (karianne@ii.uib.no)

## INF329 – Spring 2007

# Testing Concurrent Programs

- Conc. programs have a degree of non-determinism
- Most conc. tests are either safety tests or liveness tests
- Problem: Test code can introduce timing & sync. artefacts that masks bugs that otherwise might break the program

# Measuring performance

Three ways:

- ***Throughput***: Rate of which conc. tasks completes
- ***Responsiveness***: Delay between request and completion
- ***Scalability***: Performance when adding more resources (cpus, memory, etc)

# 12.1 Testing for correctness

- Start with basic unit tests NOT related to conc. helpful in discovering non-conc. related bugs
- *See BoundedBuffer.java*

# 12.1.2 Testing blocking methods

- Test for blocking methods should succeed only if the thread does NOT proceed (similar to testing for exceptions)
- **Strategy**: Start a blocking activity in a separate thread, wait until block, interrupt it and assert that the blocking method completed (threw InterruptedException)
- *See BoundedBufferTest.java*

# 12.1.3 Testing safety

- Run tests on multi-CPU-systems to increase diversity of potential interleavings.
- To maximize the chance of detecting timing-sensitive races, have more threads than CPUs (always some threads running and some switched out).
- *See PutTake.java & PutTakeTest.java*

# 12.1.4 Testing resource management

- Tests for detecting resource leaks
- Storage leaks can prevent the garbage collector from doing its job
- Easy to test with heap-inspection tools
- *See TestLeak.java*

# 12.1.5 Using callbacks

- Callbacks made at known points in an objects life cycle are good opportunities to assess invariants
- Ex: One way to test a thread pool is to have a counter every time a thread is created. Useful when testing pool behaviour (like that the thread pool increase in size when demand for execution increases

# 12.1.6 Generating more interleavings

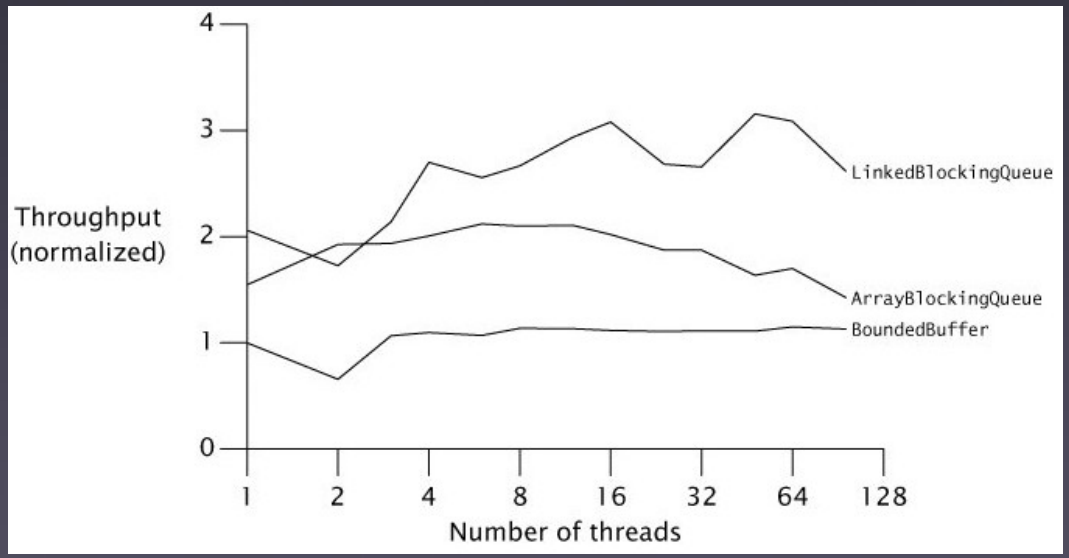
- One way to increase more interleavings is to use Thread.yield to encourage more context switches when accessing shared state.
- This way one might activate timing-sensitive bugs.
- To avoid cluttering production code with Thread.yield, one might use AOP to weave in test code dynamically

# 12.2 Testing for performance

- Performance tests seek to measure end-to-end performance metrics for representative use cases
- A common goal of perf. testing is finding optimal values for various bounds, like number of threads, size of pools, etc
- *Show demo PutTakeWithTimer.java*

# 12.2.2 Comparing multiple algorithms

- BoundedBuffer is no match for ArrayBlockingQueue or LinkedBlockingQueue
- Reason is that these two have fewer points of contention



# 12.3 Avoiding performance pitfalls

- In *theory*, developing performance tests are easy – find a typical usage scenario, make a program that runs for some time, and analyse the result.
- In *practice*, there are a number of things that can happen and ruin your test results

# 12.3.1 Garbage collection

- Timing of GC is unpredictable, might run during a test and ruin your result
- Two main strategies to avoid this:
  - Make sure GC isn't running during a test (-verbose:gc to find out)
  - Run the test longer, and let GC be a part of the result. Since GC will also run during production, this is more realistic.

# 12.3.2 Dynamic compilation

- Java is dynamically compiled, making perf. testing harder
- JVM will turn often used parts of bytecode into machine code in to improve performance, making timing unpredictable.
- Timing tests should only run after all code is turned into machine code (make the test run long enough)

# 12.3.3 Unrealistic sampling of code paths

- The JVM is permitted to use information specific to the execution to make better code
- Compiling method M in one program might generate different code in another
- Always mix tests for multi-threaded performance with tests for single-threaded performance

# 12.3.4 Unrealistic degree of contention

- As we saw in the BoundedBuffer test, producer & consumer operated on the collection (take/put).
- But they didn't actually DO anything with the items
- In an actual application, there will therefore be *less* contention than were measured in the tests

# 12.3.5 Dead code elimination

- Code that does not affect the outcome of operations will be *eliminated* by the JVMs optimizer.
- A problem since most tests don't actually do much useful work.
- This will speed up your tests and affect your test results
- Solution: Every computed result must be used *somehow*

# 12.4 Complementary testing approaches

- Finding all bugs is unrealistic goal, but testing increases confidence in code
- Manual code reviews by experts is the most effective approach
- Use of static analysis tools, tools that analyse code *without* executing it.
  - Inconsistent sync., unreleased locks and spin loops, etc
- AOP: Only limited use for conc. testing, most AOP frameworks do not support pointcuts at synch. points.

# 12.4.4 Profiling tools

- Tools that analyse thread execution and their lifecycle can help spot bottlenecks in code.
- Note that monitoring tools can affect the performance and your test result!
- *Demo of NetBeans Profiler, profiling PutTakeWithTimer.java*