JavaOne™

Sun's 2002 Worldwide Java Developer Conference

How NOT To Write A Microbenchmark

Dr. Cliff Click
Senior Staff Engineer
Sun Microsystems

JavaOne™

Sun's 2002 Worldwide Java Developer Conference

How NOT To Write A Microbenchmark
or
Lies, Damn Lies, and Microbenchmarks

Microbenchmarks are a sharp knife

Microbenchmarks are like a microscope

Magnification is high,
but what the heck are you looking at?

Like "truths, half-truths, and statistics",
microbenchmarks can be **very** misleading

Learning Objectives

- As a result of this presentation, you will be able to:
  - Recognize when a benchmark lies
  - Recognize what a benchmark can tell you (as opposed to what it purports to tell you)
  - Understand how some popular benchmarks are flawed
  - Write a microbenchmark that doesn't lie (to you)

Speaker’s Qualifications

- Dr. Click is a Senior Staff Engineer at Sun Microsystems
- Dr. Click wrote his first compiler at age 16 and has been writing:
  - runtime compilers for 15 years, and
  - optimizing compilers for 10 years
- Dr. Click architected the HotSpot™ Server Compiler

Your JVM is Lousy Because it Doesn't Do... "X"

I routinely get handed a microbenchmark and told "HotSpot doesn't do X"

49% chance it doesn't matter,

49% chance they got fooled,

1% chance HotSpot didn't do "Y" but should

1% chance HotSpot didn't do "X" but should

Agenda

- Recent real-word benchmark disaster
- Popular microbenchmarks and their flaws
- When to disbelieve a benchmark
- How to write your own

What is a Microbenchmark?

- Small program
  - Datasets may be large
- All time spent in a few lines of code
- Performance depends on how those few lines are compiled
- Goal: Discover some particular fact
- Remove all other variables

Why Run Microbenchmarks?

- Discover some targeted fact
  - Such as the cost of 'turned off' Asserts, or
  - Will this inline?
  - Will another layer of abstraction hurt performance?
- Fun
  - My JIT is faster than your JIT
  - My Java is faster than your C
- But dangerous!

How HotSpot Works

- HotSpot is a mixed-mode system
- Code first runs interpreted
  - Profiles gathered
- Hot code gets compiled
- Same code "just runs faster" after awhile
- Bail out to interpreter for rare events
  - Never taken before code
  - Class loading, initializing

Example from Magazine Editor

- What do (turned off) Asserts cost?
- Tiny 5-line function r/w's global variable
- Run in a loop 100,000,000 times
- With explicit check – 5 sec
- With Assert (off) – 0.2 sec
- With no test at all – 5 sec
- What did he really measure?

Assert Example

```
static int sval;    // Global variable
static int test_assert(int val) {
  assert (val >= 0) : "should be positive";
  sval = val * 6;
  sval += 3;        // Read/write global
  sval /= 2;
  return val+2;     //
}
static void main(String args[]) {
  int REPEAT = Integer.parseInt(args[0]);
  int v=0;
  for( int i=0; i<REPEAT; i++ )
    v = test_assert(v);
}
```

Assert Example

```
static int sval;    // Global variable
static int test_assert(int val) {
  assert (val >= 0) : "should be positive";
  sval = val * 6;
  sval += 3;        // Read/write global
  sval /= 2;
  return val+2;
}
static void main(String args[]) {
  int REPEAT = Integer.parseInt(args[0]);
  int v=0;
  for( int i=0; i<REPEAT; i++ )
    v = test_assert(v);
}
```

Assert Example

```
static int sval;    // Global variable
static int test_assert(int val) {
  assert (val >= 0) : "should be positive";
  sval = val * 6;
  sval += 3;        // Read/write global
  sval /= 2;
  return val+2;     //
}
static void main(String args[]) {
  int REPEAT = Integer.parseInt(args[0]);
  int v=0;
  for( int i=0; i<REPEAT; i++ )
    v = test_assert(v);
}
```

Assert Example

```
static int sval;    // Global variable
static int test_explicit(int val) {
  if( val < 0 ) throw ...;
  sval = val * 6;
  sval += 3;        // Read/write global
  sval /= 2;
  return val+2;     //
}
static void main(String args[]) {
  int REPEAT = Integer.parseInt(args[0]);
  int v=0;
  for( int i=0; i<REPEAT; i++ )
    v = test_explicit(v);
}
```

Assert Example (cont.)

- No synchronization ⇒ hoist static global into register, only do final write
- Small hot static function ⇒ inline into loop
- Asserts turned off ⇒ no test in code

```
static int sval;     // Global variable
static void main(String args[]) {
  int REPEAT = Integer.parseInt(args[0]);
  int v=0;

  for( int i=0; i<REPEAT; i++ ) {
    sval = (v*6+3)/2;
    v = v+2;
  }
}
```

Assert Example (cont.)

- Small loop ⇒ unroll it a lot
  - Need pre-loop to 'align' loop
- Again remove redundant writes to global static

```
static int sval;    // Global variable
static void main(String args[]) {
  int REPEAT = Integer.parseInt(args[0]);
  int v=0;
  // pre-loop goes here...
  for( int i=0; i<REPEAT; i+=16 ) {
    sval = ((v+2*14)*6+3)/2;
    v = v+2*16;
  }
}
```

Benchmark is 'spammed'

- Loop body is basically 'dead'
- Unrolling speeds up by arbitrary factor
- Note: 0.2 sec / 100Million iters * 450 Mhz clock
  ⇒ Loop runs too fast: < 1 clock/iter
- But yet...
- ... different loops run 20x faster
- What's really happening?

A Tale of Four Loops

```
static void main(String args[]) {
  int REPEAT = Integer.parseInt(args[0]);
  int v=0;
  for( int i=0; i<REPEAT; i++ )
    v = test_no_check(v);  // hidden warmup loop: 0.2 sec
  long time1 = System.currentTimeMillis()
  for( int i=0; i<REPEAT; i++ )
    v = test_explicit(v);  // Loop 1: 5 sec
  long time2 = System.currentTimeMillis()
  System.out.println("explicit check time="+...);
  for( int i=0; i<REPEAT; i++ )
    v = test_assert(v);    // Loop 2: 0.2 sec
  long time3 = System.currentTimeMillis()
  System.out.println("assert time=",...);
  for( int i=0; i<REPEAT; i++ )
    v = test_no_check(v);  // Loop 3: 5 sec
  long time4 = System.currentTimeMillis()
  System.out.println("no check time=",...);
}
```

On-Stack Replacement

- HotSpot is mixed mode: interp + compiled
- Hidden warmup loop starts interpreted
- Get's OSR'd: generate code for middle of loop
- 'explicit check' loop never yet executed, so...
  - Don't inline test_explicit(), no unroll
  - 'println' causes class loading ⇒ drop down to interpreter
- Hidden loop runs fast
- 'explicit check' loop runs slow

On-Stack Replacement (con't)

- Continue 'assert' loop in interpreter
- Get's OSR'd: generate code for middle of loop
- 'no check' loop never yet executed, so...
  - Don't inline test_no_check(), no unroll
- Assert loop runs fast
- 'no check' loop runs slow

Dangers of Microbenchmarks

- Looking for cost of Asserts...
- But found OSR Policy Bug (fixed in 1.4.1)
- Why not found before?
  - Only impacts microbenchmarks
  - Not found in any major benchmark
  - Not found in any major customer app
- Note: test_explicit and test_no_check: 5 sec
  - Check is compare & predicted branch
  - Too cheap, hidden in other math

Self-Calibrated Benchmarks

- Want a robust microbenchmark that runs on a wide variety of machines
- How long should loop run?
  - Depends on machine!
- Time a few iterations to get a sense of speed
  - Then time enough runs to reach steady state
- Report iterations/sec
- Basic idea is sound, but devil is in the details

Self-Calibrated Benchmarks

- First timing loop runs interpreted
  - Also pays compile-time cost
- Count needed to run reasonable time is low
- Main timed run used fast compiled code
- Runs too fast to get result above clock jitter
- Divide small count by clock jitter ⇒ random result
- Report random score

jBYTEmark Methodology

- Count iterations to run for 10 msec
  - First run takes 104 msec
  - Way too short, always = 1 iteration
- Then use 1st run time to compute iterations needed to run 100msec
  - Way too short, always = 1 iteration
- Then average 5 runs
  - Next 5 runs take 80, 60, 20, 20, 20 msec
  - Clock jitter = 1 msec

Non-constant work per iter

- Non-sync'd String speed test
- Run with & without some synchronization
- Use concat, continuously add Strings
- Benchmark requires copy before add, so...
  - Each iteration takes more work than before
- Self-timed loop has some jitter...
  - 10000 main-loop iters with sync
  - 10030 main-loop iters withOUT sync

Non-constant work per iter (con't)

- Work for Loop1:
  - 10000^2 copies, 10000 concats, 10000 syncs
- Work for Loop2:
  - 10030^2 copies, 10000 concats, no syncs
- Loop2 avoids 10000 syncs
- But pays 600,900 more copies
- Benchmark compares times, BUT
- Extra work swamps sync cost!

Beware 'hidden' Cache Blowout

- Work appears to be linear with time
- More iterations should report same work/sec
- Grow dataset to run longer
- Blow out caches!
- Self-calibrated loop jitter can make your data set fit in L1 cache or not!
- Jitter strongly affects apparent work/sec rate

Beware 'hidden' Cache Blowout

- jBYTEmark StringSort with time raised to 1 sec
- Each iteration adds a 4K array
- Needs 1 or 2 iterations to hit 100msec
- Scale by 10 to reach 1 sec
- Uses either 10 or 20 4K arrays
- Working set is either 40K or 80K
- 64K L1 cache

Explicitly Handle GC

- GC pauses occur at unpredictable times
- GC throughput is predictable
- Either don't allocate in your main loops
  - So no GC pauses
- OR run long enough to reach GC steady state
  - Each run spends roughly same time doing GC

CaffeineMark3 Logic Benchmark

- Purports to test logical operation speed
- With unrolling OR constant propagation, whole function is dead (oops!)
- First timing loop: 900K iterations to run 1 sec
- Second loop runs 2.7M iterations in 0.685 sec
- Score reports as negative (overflow) or huge
- Overall score dominated by Logic score

Know what you test

- SpecJVM98 209_db
  - 85% of time spent in shell_sort
- Scimark2 MonteCarlo simulation
  - 80% of time spent in sync'd Random.next
- jBYTEmark StringSort
  - 75% of time spent in byte copy
- CaffeineMark Logic test
  - "He's dead, Jim"

How To Write a Microbenchmark
(general – not just Java)

- Pick suitable goals!
  - Can't judge web server performance by jBYTEmarks
  - Only answer very narrow questions
  - Make sure you actually care about the answer!
- Be aware of system load, clock granularity
- Self-calibrated loops always have jitter
  - Sanity check the time on the main run!
- Run to steady state, at least 10 sec

How To Write a Microbenchmark
(general – not just Java)

- Fixed size datasets (cache effects)
  - Large datasets have page coloring issues
- Constant amount of work per iteration
  - Or results not comparable
- Avoid the 'eqntott Syndrome'
  - Capture 1st run result (interpreted)
  - Compare to last result

How To Write a Microbenchmark
(general – not just Java)

- Avoid 'dead' loops
  - These can have infinite speedups!
  - Print final answer
  - Make computation non-trivial
- Or handle dead loops
  - Report when speedup is unreasonable
  - CaffeineMark3 & JavaGrande Section 1 mix Infinities and reasonable scores

How To Write a Microbenchmark
(Java-specific)

- Be explicit about GC
- Thread scheduling is not deterministic
- JIT performance may change over time
- Warmup loop+test code before ANY timing
  - (HotSpot specific)

OSR Bug Avoidance

- Fixed in 1.4.1
- Write 1 loop per method for Microbenchmarks
- Doesn't matter for 'real' apps
  - Don't write 1-loop-per-method code to avoid bug

Summary

- Microbenchmarks can be easy, fun, informative
- They can be very misleading! Beware!
- Pick suitable goals
- Warmup code before timing
- Run reasonable length of time
- Fixed work / fixed datasets per iteration
- Sanity-check final times and results
- Handle 'dead' loops, GC issues

If You Only Remember One Thing…

Put Microtrust in a Microbenchmark

Q&A

JavaOne

JavaOne℠

Sun's 2002 Worldwide Java Developer Conference

BEYOND
BOUNDARIES

Session #