
Java is a trademark of Sun Microsystems, Inc.

JavaOne℠

# The Art of (Java) Benchmarking

Dr. Cliff Click
Azul Systems
http://blogs.azulsystems.com/cliff

# Benchmarking is Easy!!!

- And Fun!!!
- “My Java is faster than your C!!!”
- And generally wrong...
- *Without exception every* microbenchmark I've seen has had **serious** flaws
  - Except those I've had a hand in correcting
- **Serious** =
  - “Score” is unrelated to intended measurement or
  - error bars exceed measured values

# Split out micro-bench vs macro-bench

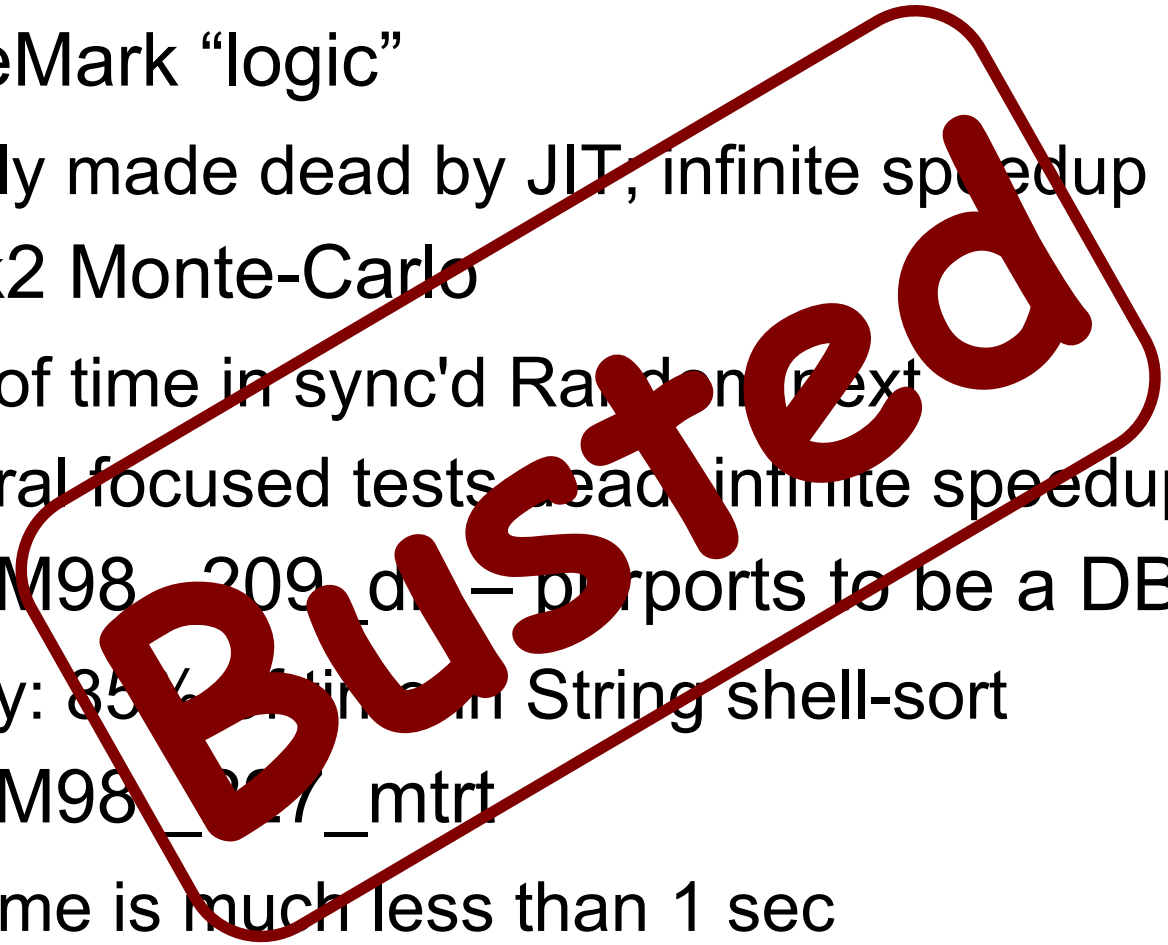
- Micro-benchmarks are things you write yourself
  - Attempt to discover some narrow targeted fact
  - Generally a timed tight loop around some “work”
  - Report score as iterations/sec
    - e.g. allocations/sec – object pooling vs GC
- Macro-benchmarks are supposed to be realistic
  - Larger, longer running
    - e.g. WebServer, DB caching/front-end, Portal App
    - SpecJBB, SpecJAppServer, XMLMark, Trade6
  - Load testing of Your Real App

# Some Older Busted Micro-Benchmarks

- CaffeineMark “logic”
  - trivially made dead by JIT; infinite speedup
- SciMark2 Monte-Carlo
  - 80% of time in sync'd Random.next
  - Several focused tests dead; infinite speedup
- SpecJVM98 _209_db – purports to be a DB test
  - Really: 85% of time in String shell-sort
- SpecJVM98 _227_mtrt
  - Runtime is much less than 1 sec

# Some Older Busted Micro-Benchmarks

- CaffeineMark “logic”
  - trivially made dead by JIT, infinite speedup
- SciMark2 Monte-Carlo
  - 80% of time in sync'd Random.next
  - Several focused tests dead, infinite speedup
- SpecJVM98 _209_db – purports to be a DB test
  - Really: 85% of time in String shell-sort
- SpecJVM98 _227_mtrt
  - Runtime is much less than 1 sec



# Dead Loops

```
// how fast is divide-by-10?
long start = Sys.CTM();
for( int i=0; i<N; i++ )
  int x = i/10;
return N*1000/(Sys.CTM()-start);
```

- Timeline:
  - 1- Interpret a while, assume 10ms
  - 2- JIT; “x” not used, loop is dead, removed, 10ms
  - 3- “Instantly” execute rest of loop
- Time to run: 20ms – ***Independent of N!***
  - Vary N ==> vary score ==> “Dial-o-Score!”

# Dead Loops

- Sometimes JIT proves “results not needed”
  - Then throws out whole work-loop
  - ***After*** running long enough to JIT
  - So loop runs at least a little while first
- Score “ops/sec” not related to trip count 'N'
  - Larger N ==> larger score
- Score can be infinite- or NaN
  - Generally reported as a very large, but valid #
  - And mixed in with other numbers, confusing things
    - (e.g. geomean of infinite's and other more real numbers)

# SciMark2 Monte-Carlo

- 80% of time in **`synchronize`**'d Random.next
- 3-letter-company “spammed” it by replacing with intrinsic doing a CompareAndSwap (CAS)
- I was ordered to follow suit (match performance)
- Doug Lea said “wait: just make a CAS from Java”
- Hence sun.misc.AtomicLong was born
- Rapidly replaced by Unsafe.compareAndSwap...
- ...and eventually java.lang.Atomic*

# Micro-bench Advice: Warmup

- Code starts interpreted, then JIT'd
  - JIT'd code is 10x faster than interpreter
- JIT'ing happens “after a while”
  - HotSpot -server: 10,000 iterations
  - Plus compile time
- Warmup code with some trial runs
  - Keeping testing until run-times stabilize

# Micro-bench Advice: Warmup

- Not allowing warmup is a common mistake
- Popular failure-mode of C-vs-Java comparisons
  - Found on many, many, many web pages
  - Entire benchmark runs in few milli-seconds
  - There are domains requiring milli-second reboots...
- **But** most desktop/server apps expect:
  - Reboots are minutes long and days apart
  - Steady-state throughput after warmup is key
  - So a benchmark that ends in <10sec
    probably does not measure anything interesting

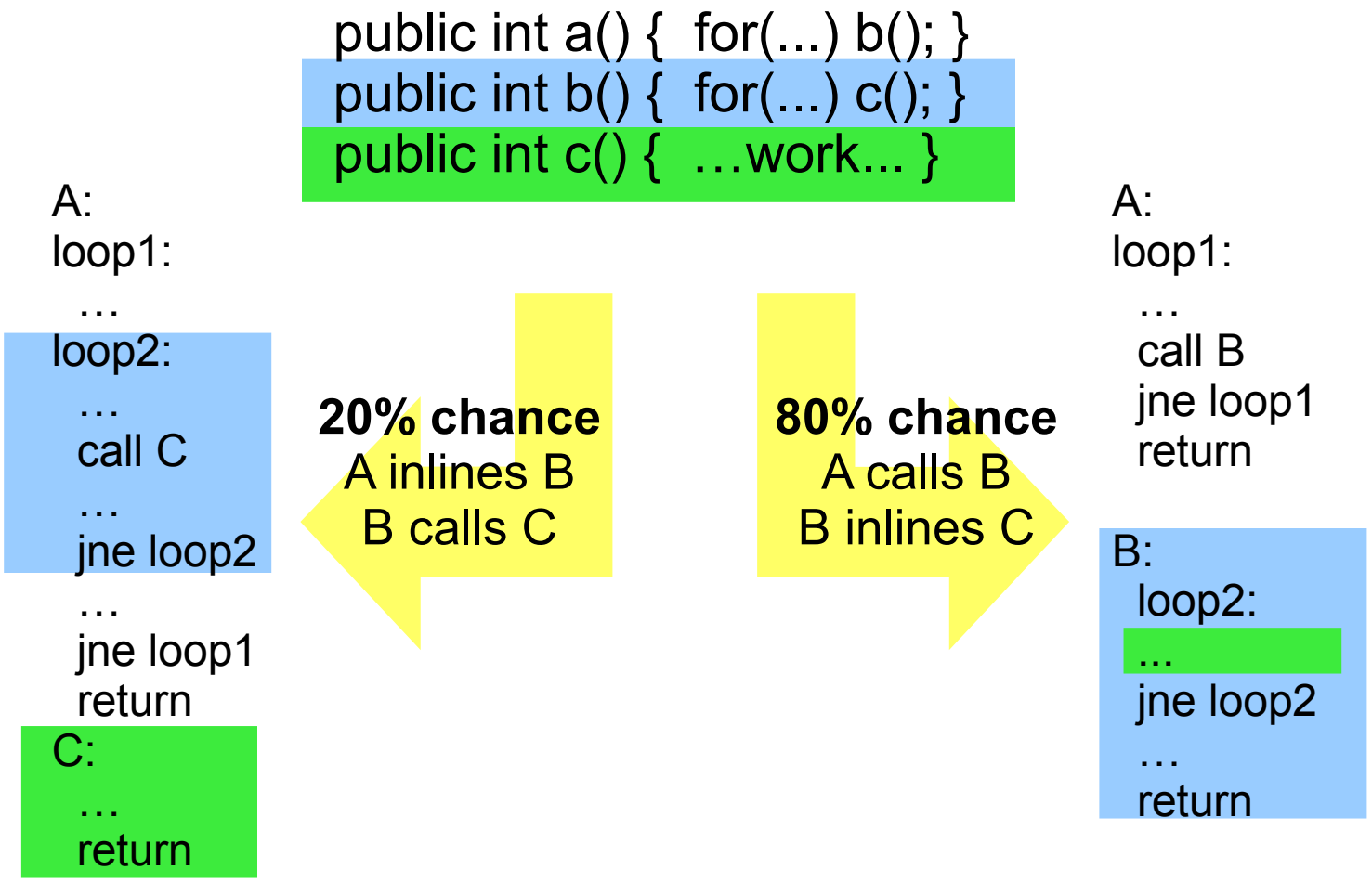
# Micro-bench Advice: “Compile plan”

- JIT makes inlining & other complex decisions
  - Based on very volatile & random data
  - Inline decisions vary from run-to-run
- Performance varies from run-to-run
  - Stable numbers within a single JVM invocation
  - But could vary by >20% with new JVM launch
  - Bigger apps are more performance-stable
- Micro-benchmarks tend to be “fragile” here
  - e.g. 1 JVM launch in 5 will be 20% slower*



*”*Statistically Rigorous Java Performance Evaluation*” OOPSLA 2007

# Micro-bench Advice: “Compile plan”



# Micro-bench Advice: “Compile plan”

- Launch the JVM many times
  - Toss 1st launch to remove OS caching effects
  - Average out “good” runs with the “bad”
  - Don't otherwise toss outliers
    - (unless you have good reason: i.e. unrelated load)
- Enough times to get statistically relevant results
  - Might require 30+ runs
- Report average **and** standard deviation
  - In this case, expect to see a large std.dev



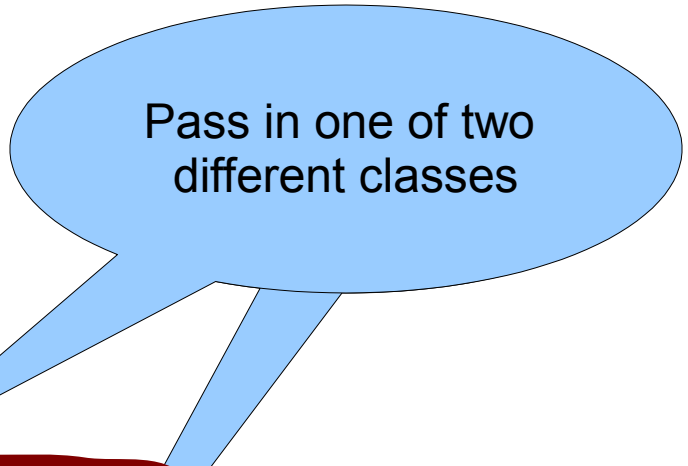
*”Statistically rigorous Java performance evaluation” OOPSLA 2007

# Micro-bench Advice: "1st fast, 2nd slow"

- Timing harness needs to invoke many targets
  - In a loop, repeatedly a few times
  - Else JIT sees 1 hot target in a loop
    - And then does a guarded inline
    - And then hoists the timed work outside of timing loop

```
class bench1 implements bench { void sqrt(int i); }
class bench2 implements bench { void sqrt(int i); }
static final int N=1000000; // million
...
static int test( bench B ) {
  long start = System.currentTimeMillis();
  for( int i=0; i<N; i++ )
    B.sqrt(i);      // hot loop v-call
  return N*1000/(System.currentTimeMillis()-start);
```

# Micro-bench Advice: "1st fast, 2nd slow"




```
class bench1 implements bench { void sqrt(int i); }
class bench2 implements bench { void sqrt(int i); }
static final int N=1000000; // million
...
static int test( bench B ) {
  long start = System.currentTimeMillis();
  for( int i=0; i<N; i++ )
    B.sqrt(i);      // hot loop v-call
  return N*1000/(System.currentTimeMillis()-start);
```

# Micro-bench Advice: v-call optimization

- First call: **`test(new bench1)`**

```
long start = Sys.CTM();
for( int i=0; i<N; i++ )
  B.sqrt(i);
return N*1000/(Sys.CTM()-start);
```

# Micro-bench Advice: v-call optimization

- First call: **`test(new bench1)`**
  - Single target callsite; JIT does guarded inlining
    - Inlines **`bench1.sqrt`**

```
long start = Sys.CTM();
for( int i=0; i<N; i++ )
  B.sqrt(i);
return N*1000/(Sys.CTM()-start);
```


```
long start = Sys.CTM();
for( int i=0; i<N; i++ )
  Math.sqrt(i); // inline bench1.sqrt
return N*1000/(Sys.CTM()-start);
```

# Micro-bench Advice: v-call optimization

- First call: **`test(new bench1)`**
  - Single target callsite; JIT does guarded inlining
    - Inlines **`bench1.sqrt`**
    - Hoists loop-invariants, dead-code-remove, etc
  - Execution time does NOT depend on N!!!
    - Dreaded "Dial-o-Score!"

```
long start = Sys.CTM();
for( int i=0; i<N; i++ )
  B.sqrt(i);
return N*1000/(Sys.CTM()-start);
```

```
long start = Sys.CTM();
// JIT'd loop removed
return N*1000/(Sys.CTM()-start);
```

guarded inline

```
long start = Sys.CTM();
for( int i=0; i<N; i++ )
  Math.sqrt(i); // inline bench1.sqrt
return N*1000/(Sys.CTM()-start);
```

dead code

# Micro-bench Advice: v-call optimization

- Second call: **`test(new bench2)`**
  - 2nd target of call; guarded inlining fails
  - Code is incorrect; must be re-JIT'd
  - Measures overhead of N calls to bench2.sqrt
    - Plus guard failure, deoptimization
    - Plus JIT'ing new version of test()
    - Plus virtual call overhead

```
long start = System.CTM();
for( int i=0; i<N; i++ )
  B.sqrt(i);
return N*1000/(System.CTM()-start);
```

```
int bench2.sqrt( int i ) {
  // alternate impl
}
```

# Micro-bench Advice: v-call optimization

- Reversing order of calls reverses “good” & “bad”
  - e.g. “`test(new bench2); test(new bench1);`”

# Micro-bench Advice: v-call optimization

- Reversing order of calls reverses “good” & “bad”
  - e.g. “`test(new bench2); test(new bench1);`”
- Timing harness needs to invoke all targets
  - In a loop, repeatedly a few times

```
class bench1 implements bench { void sqrt(int i); }
class bench2 implements bench { void sqrt(int i); }
...
  // warmup loop
  for( int i=0; i<10; i++ ) {
    test( new bench1 );
    test( new bench2 );
  }
  // now try timing
  printf(test(new bench1));
```

# Micro-bench Advice: v-call optimization

- Reversing order of calls reverses “good” & “bad”
  - e.g. “`test(new bench2); test(new bench1);`”
- Timing harness needs to invoke all targets
  - In a loop, repeatedly a few times

```
class bench1 implements bench { void sqrt(int i); }
class bench2 implements bench { void sqrt(int i); }
...
  // warmup loop
  for( int i=0; i<10; i++ ) {
    test( new bench1 );
    test( new bench2 );
  }
  // now try timing
  printf(test(new bench1));
```

# Micro-bench Advice: GC

- Avoid GC or embrace it
- Either no (or trivial) allocation, or use `verbose:gc` to make sure you hit steady-state
- Statistics: not just average, but also std-dev
- Look for trends
  - Could be creeping GC behavior
- Could be “leaks” causing more-work-per-run
  - e.g. leaky HashTable growing heap or
  - Growing a LinkedList slows down searches

# Micro-bench Advice: Synchronization

- Account for multi-threaded & locking
- I **do** see people testing, e.g. locking costs on single-threaded programs
- **Never** contended lock is very cheap
  - +BiasedLocking makes it even cheaper
- **Very slightly** contended lock is probably 4x more
- **Real** contention: Amdahl's Law
  - Plus **lots and lots** of OS overhead
- java.util.concurrent is your friend

# Micro-bench Advice

- Realistic runtimes
  - Unless you need sub-milli-sec reboots
- Warm-up loops - give the JIT a chance
- Statistics: plan for variation in results
- Dead loops – look for “Dial-o-Score!”, deal with it
- 1st run fast, 2nd run slow – look for it, deal with it
- GC: avoid or embrace

# Macro-bench warnings

- JVM98 is too small anymore
  - Easy target; cache-resident; GC ignored
- JBB2000, 2005
  - Not much harder target
  - VERY popular, easy enough to “spam”
  - Score rarely related to anything real
- SpecJAppServer, DaCapo, SpecJVM2008, XMLMark
  - Bigger, harder to spam, less popular

# Macro-bench warnings

- Popular ones are targeted by companies
- General idea: JVM engineers are honest
  - But want the best for company
  - So do targeted optimizations
    - e.g. intrinsic CAS for Random.next
  - Probably useful to somebody
  - Never incorrect
  - Definitely helps **this** benchmark

# Typical Performance Tuning Cycle

- Benchmark X becomes popular
- Management tells Engineer: “Improve X's score!”
- Engineer does an in-depth study of X
- Decides optimization “Y” will help
  - And Y is not broken for anybody
  - Possibly helps some other program
- Implements & ships a JVM with “Y”
- Management announces score of “X” is now 2*X
- Users yawn in disbelief: “Y” does not help them

# SpecJBB2000

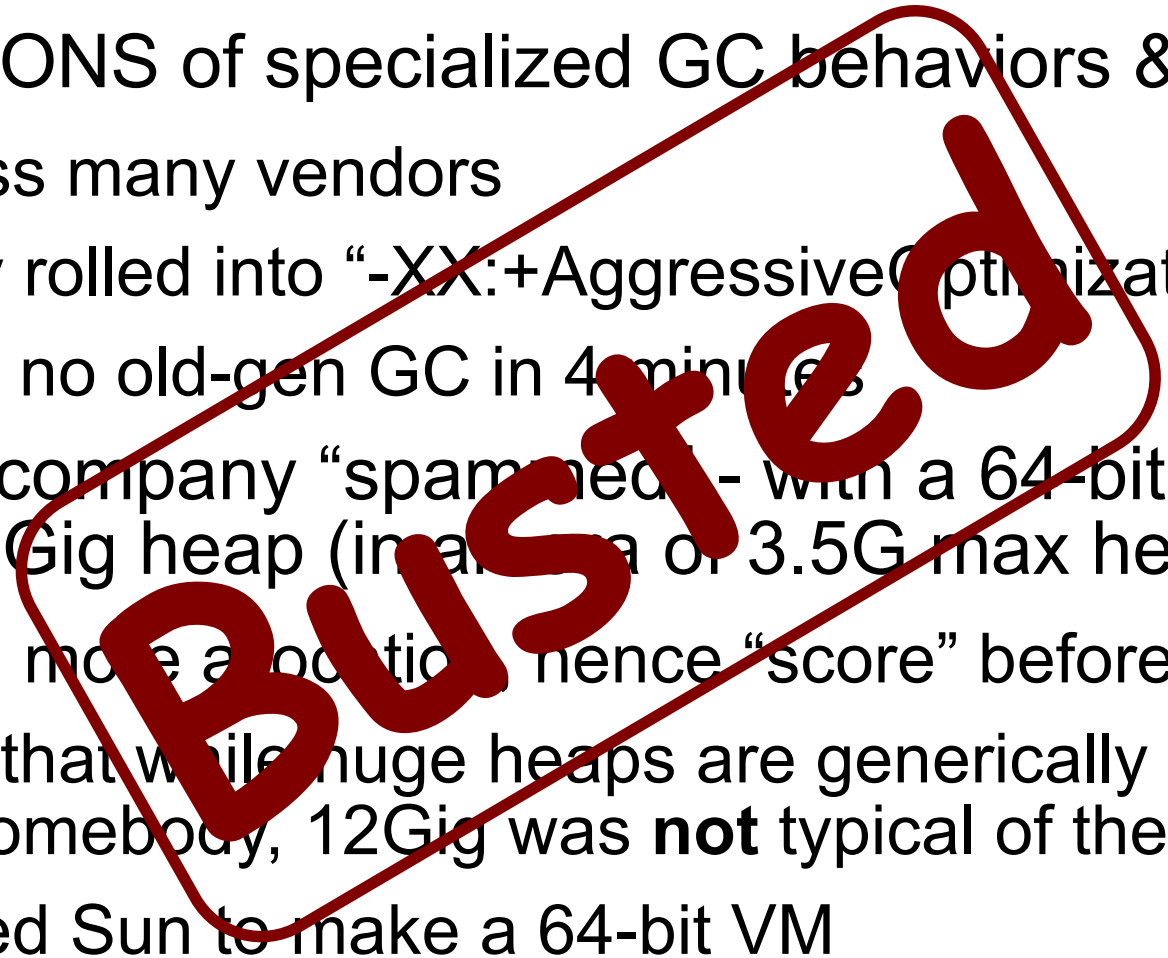
- Embarrassing parallel - no contended locking
- No I/O, no database, no old-gen GC
  - NOT typically of any middle-ware
  - *Very high* allocation rate of young-gen objects, definitely not typically
    - But maybe your program gets close?
- Key to performance:
  having enough Heap to avoid old-gen GC during 4-min timed window

# SpecJBB2000: Spamming

- Drove TONS of specialized GC behaviors & flags
  - Across many vendors
  - Many rolled into “-XX:+AggressiveOptimizations”
  - Goal: no old-gen GC in 4 minutes
- 3-letter-company “spammed” - with a 64-bit VM and 12Gig heap (in an era of 3.5G max heaps)
  - Much more allocation, hence “score” before GC
  - Note that while huge heaps are generically useful to somebody, 12Gig was **not** typical of the time
  - Forced Sun to make a 64-bit VM

# SpecJBB2000: Spamming

- Drove TONS of specialized GC behaviors & flags
  - Across many vendors
  - Many rolled into "-XX:+AggressiveOptimizations"
  - Goal: no old-gen GC in 4 minutes
- 3-letter-company "spammed" - with a 64-bit VM and 12Gig heap (in area of 3.5G max heaps)
  - Much more allocation, hence "score" before GC
  - Note that while huge heaps are generically useful to somebody, 12Gig was **not** typical of the time
  - Forced Sun to make a 64-bit VM



# What can you read from the results?

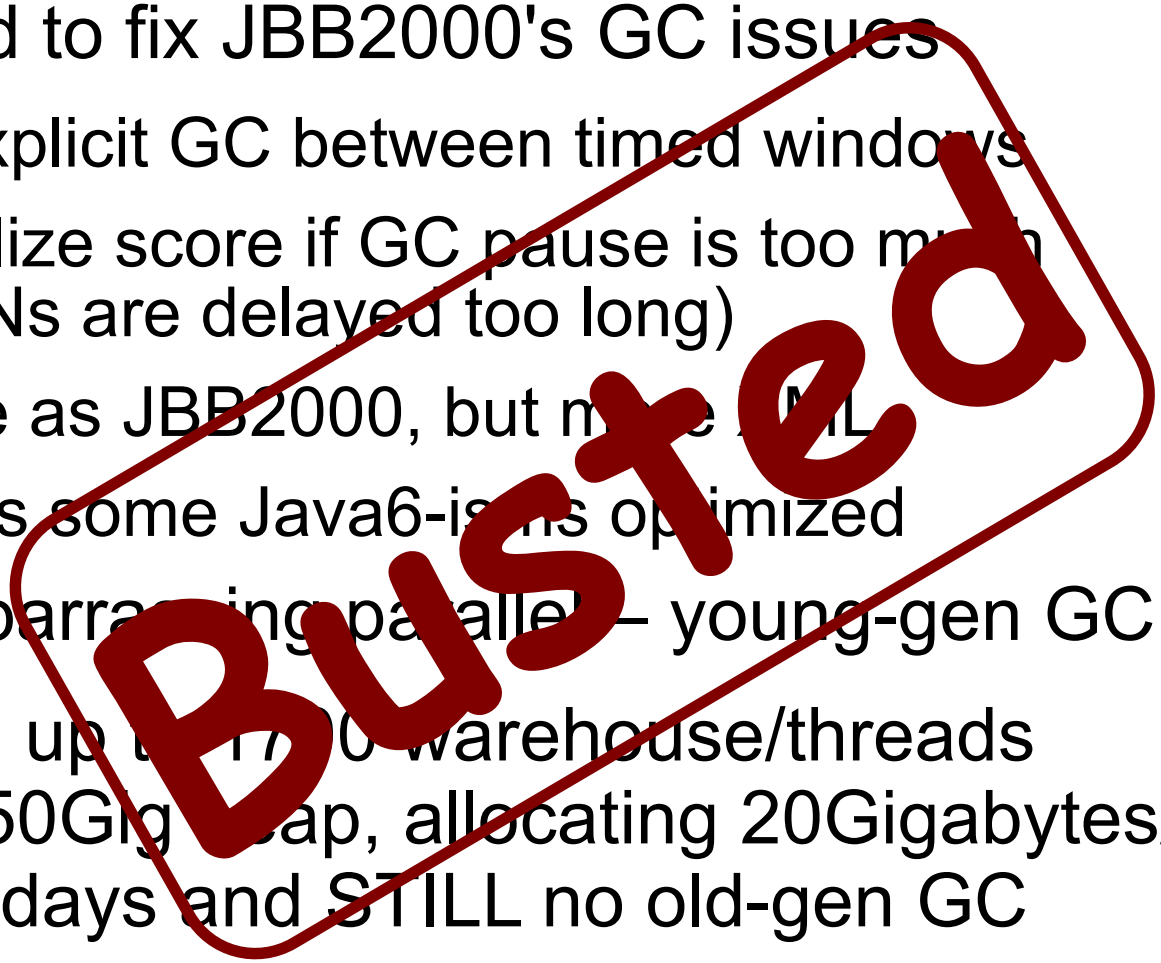
- The closer your apps resemble benchmark “X”
  - The closer improvements to X's score impact you
- Huge improvements to **unrelated** benchmarks
  - *Might be worthless to you*
- e.g. SpecJBB2000 is a perfect-young-gen GC test
  - Improvements to JBB score have been tied to better young-gen behavior
  - Most web-servers suffer from OLD-gen GC issues
    - Improving young-gen didn't help web-servers much

# SpecJBB2005

- Intended to fix JBB2000's GC issues
  - No explicit GC between timed windows
  - Penalize score if GC pause is too much (XTNs are delayed too long)
  - Same as JBB2000, but more XML
  - Needs some Java6-isms optimized
- Still embarrassing parallel – young-gen GC test
- Azul ran up to 1700 warehouse/threads on a 350Gig heap, allocating 20Gigabytes/sec for 3.5 days and STILL no old-gen GC

# SpecJBB2005

- Intended to fix JBB2000's GC issues
  - No explicit GC between timed windows
  - Penalize score if GC pause is too much (XTNs are delayed too long)
  - Same as JBB2000, but more XML
  - Needs some Java6-isms optimized
- Still embarrassingly parallel – young-gen GC test
- Azul ran up to 1700 warehouse/threads on a 350Gig heap, allocating 20Gigabytes/sec for 3.5 days and STILL no old-gen GC



# Some Popular Macro-Benchmarks

- SpecJVM98 – too small, no I/O, no GC
  - 227_mtrt – too short to say **anything**
    - Escape Analysis pays off too well here
  - 209_db – string-sort NOT db, performance tied to TLB & cache structure, not JVM
  - 222_mpegaudio – subject to odd FP optimizations
  - 228_jack – throws heavy exceptions – but so do many app-servers; also parsers are popular. Improvements here might carry over
  - 213_javac – generically useful metric for modest CPU bound applications

# Some Popular Macro-Benchmarks

- SpecJVM98 – too small, no I/O, no GC
  - 227_mtrt – too short to say **anything**
    - Escape Analysis pays off too well here
  - 209_db – String sort NOT db, performance tied to TLB & cache structure, not JVM
  - 222_mpegaudio – subject to odd FP optimizations
  - 228_jack – throws heavy exceptions – but so do many app-servers; also parsers are popular. Improvements here might carry over
  - 213_javac – generically useful metric for modest CPU bound applications

Busted

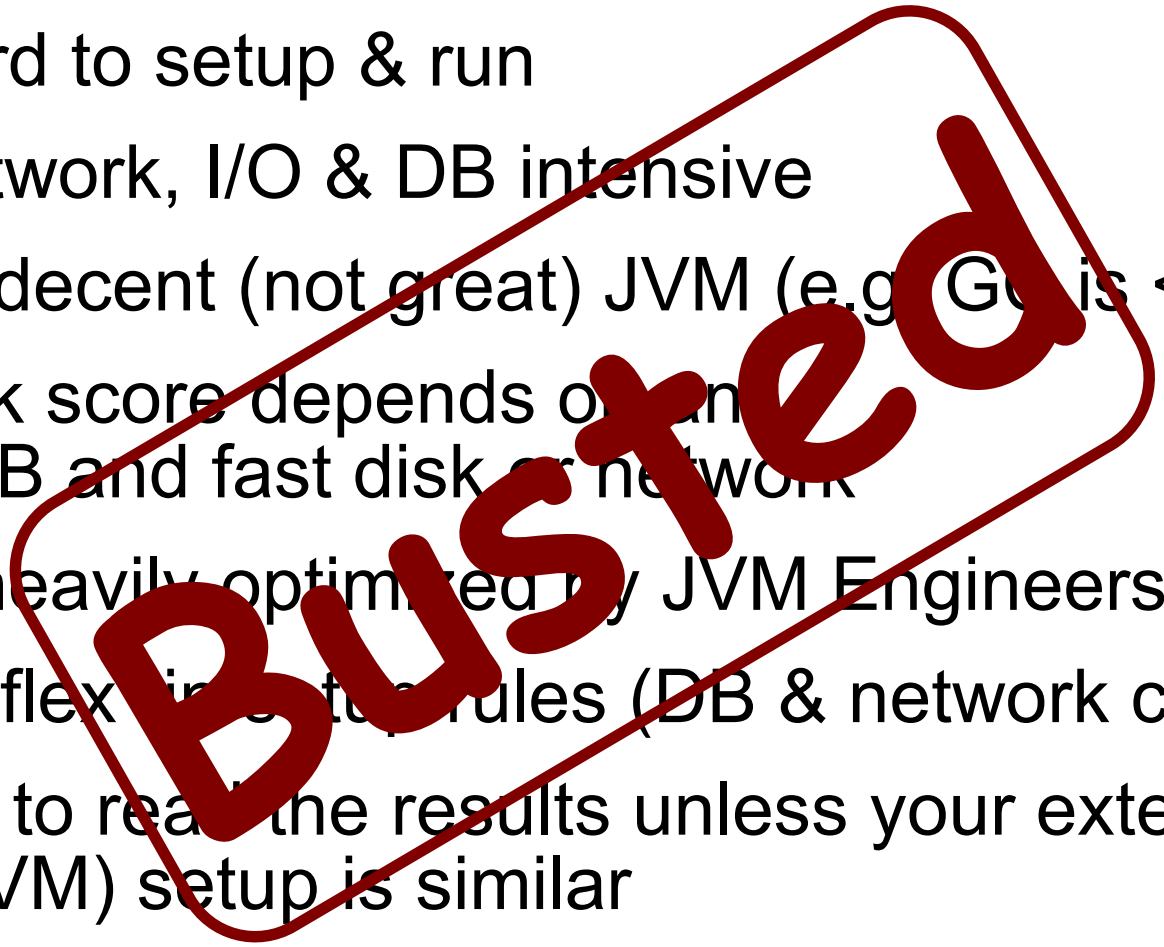
Plausible

# SpecJAppServer

- Very hard to setup & run
- Very network, I/O & DB intensive
- Need a decent (not great) JVM (e.g. GC is < 5%)
- But peak score depends on an uber-DB and fast disk or network
- Not so heavily optimized by JVM Engineers
- Lots of “flex” in setup rules (DB & network config)
- So hard to read the results unless your external (non-JVM) setup is similar

# SpecJAppServer

- Very hard to setup & run
- Very network, I/O & DB intensive
- Need a decent (not great) JVM (e.g. GC is < 5%)
- But peak score depends on an uber-DB and fast disk or network
- Not so heavily optimized by JVM Engineers
- Lots of "flex" setup rules (DB & network config)
- So hard to read the results unless your external (non-JVM) setup is similar

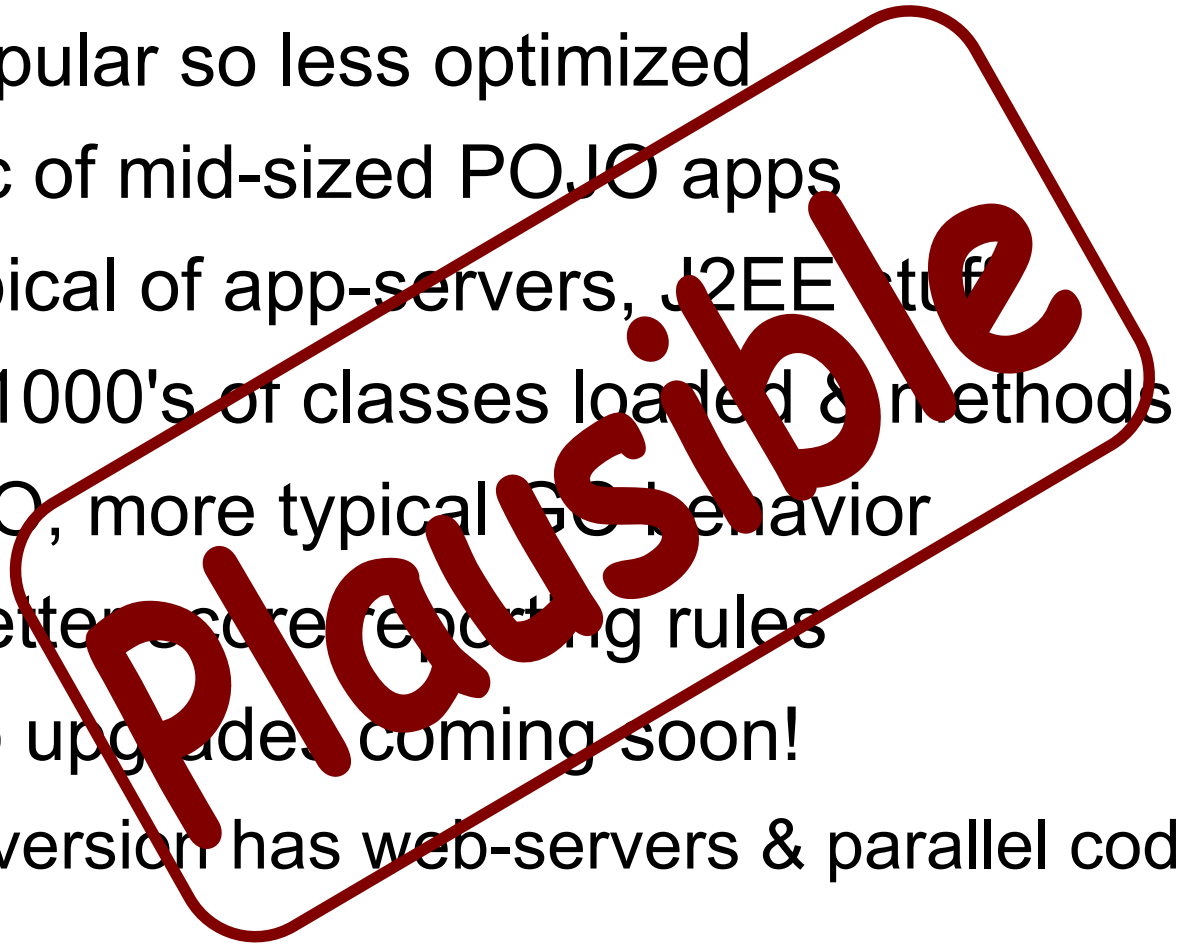


# DaCapo

- Less popular so less optimized
- Realistic of mid-sized POJO apps
- NOT typical of app-servers, J2EE stuff
- Expect 1000's of classes loaded & methods JIT'd
- Some I/O, more typical GC behavior
- Much better score reporting rules
- DaCapo upgrades coming soon!
  - New version has web-servers & parallel codes

# DaCapo

- Less popular so less optimized
- Realistic of mid-sized POJO apps
- NOT typical of app-servers, J2EE stuff
- Expect 1000's of classes loaded & methods JIT'd
- Some I/O, more typical GC behavior
- Much better score reporting rules
- DaCapo upgrades coming soon!
  - New version has web-servers & parallel codes

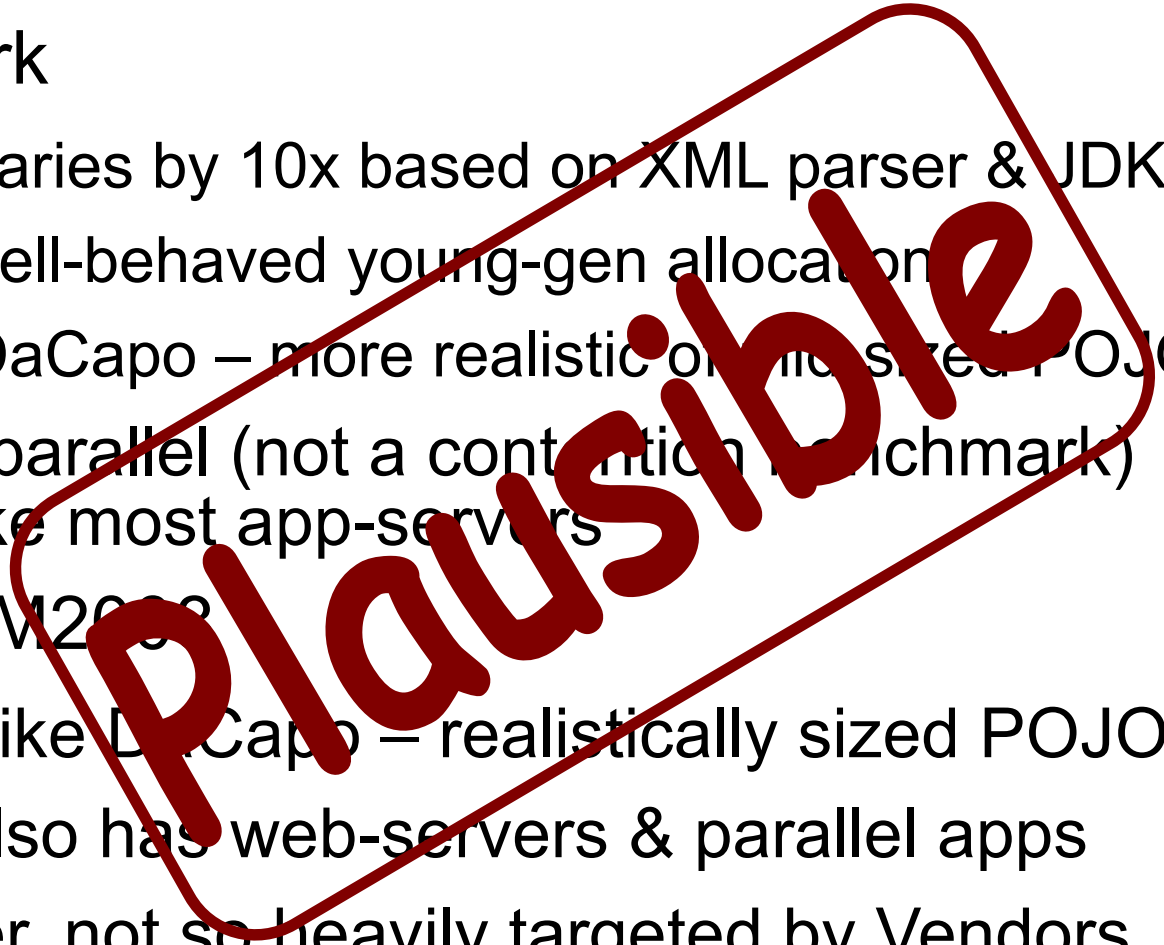


# Some Popular Macro-Benchmarks

- XMLMark
  - Perf varies by 10x based on XML parser & JDK version
  - Too-well-behaved young-gen allocation
  - Like DaCapo – more realistic of mid-sized POJO apps
  - Very parallel (not a contention benchmark) unlike most app-servers
- SpecJVM2008
  - Also like DaCapo – realistically sized POJO apps
  - But also has web-servers & parallel apps
  - Newer, not so heavily targeted by Vendors

# Some Popular Macro-Benchmarks

- XMLMark
  - Perf varies by 10x based on XML parser & JDK version
  - Too-well-behaved young-gen allocation
  - Like DaCapo – more realistic of medium-sized POJO apps
  - Very parallel (not a contention benchmark) unlike most app-servers
- SpecJVM2008
  - Also like DaCapo – realistically sized POJO apps
  - But also has web-servers & parallel apps
  - Newer, not so heavily targeted by Vendors



# “Popular” Macro-Benchmark Problems

- Unrealistic treatment of GC
  - e.g. None in timed window
  - Or perfect young-gen collections
  - Real apps typical trigger full GC every hour or so
- Unrealistic load generation
  - Not enough load to stress system
  - Or very simple or repetitive loads
  - Bottlenecks in getting load to server

# “Popular” Macro-Benchmark Problems

- Benchmark too short for full GC
  - Many real applications *leak*
    - Broken 3rd party libs, legacy code, etc
  - Leaks accumulate in old-gen
    - Which makes old-gen full GC expensive
  - But benchmark never triggers old-gen full GC
- I/O & DB not benchmarked well
  - But make a huge difference in Real Life
  - Your app might share I/O & DB with others

# Summary

- Macrobenchmarks
  - Targeted by JVM Engineers
    - *Buyer Beware!*
  - The closer the benchmark is to your problem
    - The more likely improvements will impact you
  - GC is likely to ***not be typical*** of real applications
    - *Your* applications ever go 3.5 days without a full GC?
  - I/O & DB load also probably not typical

# Summary

- Microbenchmarks
  - Easy to Write, Hard to get Right
  - *Easy to be Fooled*
  - Won't tell you much about macro-code anyways
  - **Warmup** – 1's of seconds to 10's of seconds
  - **Statistics** – average lots of runs
    - Even out variations in the “compile plan”
  - **Call out** to many methods in the hot loop
  - Be wary of **dead-code** super-score results

# Put Micro-Trust in a Micro-Benchmark!

# Put Micro-Trust in a Micro-Benchmark!