Fast Bytecodes for Funny Languages

Dr. Cliff Click
Chief JVM Architect & Distinguished Engineer
blogs.azulsystems.com/cliff
Azul Systems
Sept 24, 2008

AZUL SYSTEMS

Fast Bytecodes For Funny Languages

- JVMs are used for lots of non-Javac bytecodes
- Bytecode patterns are **different**
- Bytecode producer (this crowd) is curious:
  - Are these bytecodes “fast”? “fast enough”?
  - Am I inlining?
  - JIT'ing to good code? (JIT'ing at all?)
- Personal goal: you ARE using that fancy JIT, right?
  - Otherwise, why did I bother? ;-)
- 'nother goal: maybe help the world escape the Java box
- New language must have a fast reputation
  - Or a **large** programmer population won't look

Can Your Language Go “To The Metal”?

- Can your bytecodes go “To The Metal”?
  - e.g. Can simple code be mapped to simple machine ops
- Methodology:
  - Write dumb hot SIMPLE loop in lots of languages
  - Look at performance
  - Look at JIT'd code
  - Look for mismatch between language & JVM & machine code
- NOT-GOAL: Who is fastest
- NOT TESTED:
  - Ease of programming, time-to-market, maintenance cost
  - Domain fit (e.g. Ruby-on-Rails)
- IGNORING:
  - Blatant language / microbenchmark mismatch (FixNum/BigNum)

Can You Take THIS “To The Metal”?

- See “http://blogs.azulsystems.com/cliff/2008/09/a-plea-for-prog.html”

```
for( int i=1; i<100000000; i++ )
  sum += (sum^i)/i;
```

- Run with Java, Scala, Clojure, JRuby, JPC, Javascript/Rhino
  - Willing to try more languages
- NOT Goal:
  - Discover the fastest
- NOT Fair:
  - I changed benchmark to be more “fair” to various languages
  - e.g. Using 'double' instead of 'int' for Javascript.
- Tested on Azul JVM
  - Because it's got the best low-level JVM perf analysis tools I've seen

Scorecard

- Java (unrolled, showing 1 iter):

```
add  rTmp0, rI, 5 // for unrolled iter #5
xor  rTmp1, rSum, rTmp0
div  rTmp2, rTmp1, rTmp0
add  rSum,rTmp2,rSum
... repeat for unrolled iterations
```

- Scala – Same as Java!
  - Scala “Elegant Scala-ized version” - Ugh
- Clojure – Almost close; very allocation heavy; oddball 'holes'
- JRuby – Major inlining **not happening**,
  - misled by debugging flag?
- JPC – Fascinating; totally inlined X86 emulation
  - But JIT doesn't grok e.g. dead X86 flag setting
- Javascript/Rhino – Death by 'Double' (not 'double')

Deeper Dive

- Java – Unfair advantage. Semantics match JIT expectations.
- Scala – Borrowed heavily from Java semantics
  - Close fit allows close translation
  - And thus nearly direct translation to machine code
  - Penalty: Only have Java semantics,
    e.g. no 'unsigned' type, no auto-BigNum inflation
- JPC – Java "PC"; pure Java X86/DOS emulator
  - Massive bytecode generation; LOTS of JIT'ing
  - For this example: 16000 classes, 7800 compilations
  - But JIT falls down removing e.g. redundant X86 flag sets
  - Maybe fix by never storing 'flags' into memory
  - Also no fixnum issue

Deeper Dive

- Clojure - “almost close”
  - Good: no obvious subroutine calls in inner loop
  - Bad: Massive “ephemeral” object allocation - requires good GC
  - But needs Escape Analysis to go fast
  - Ugly: fix-num overflow checks everywhere
  - Can turn off fix-nums; could be same speed as Java
  - Weird “holes” - Not-optimized reflection calls here & there
- Jython - “almost close”
  - Also has Fixnum issue; massive allocation
  - Some extra locks thrown in
- JavaScript/Rhino
  - All things are Doubles – not 'double'
  - Same allocation issues as Fixnum
  - Otherwise looks pretty good (no fixnum checks)

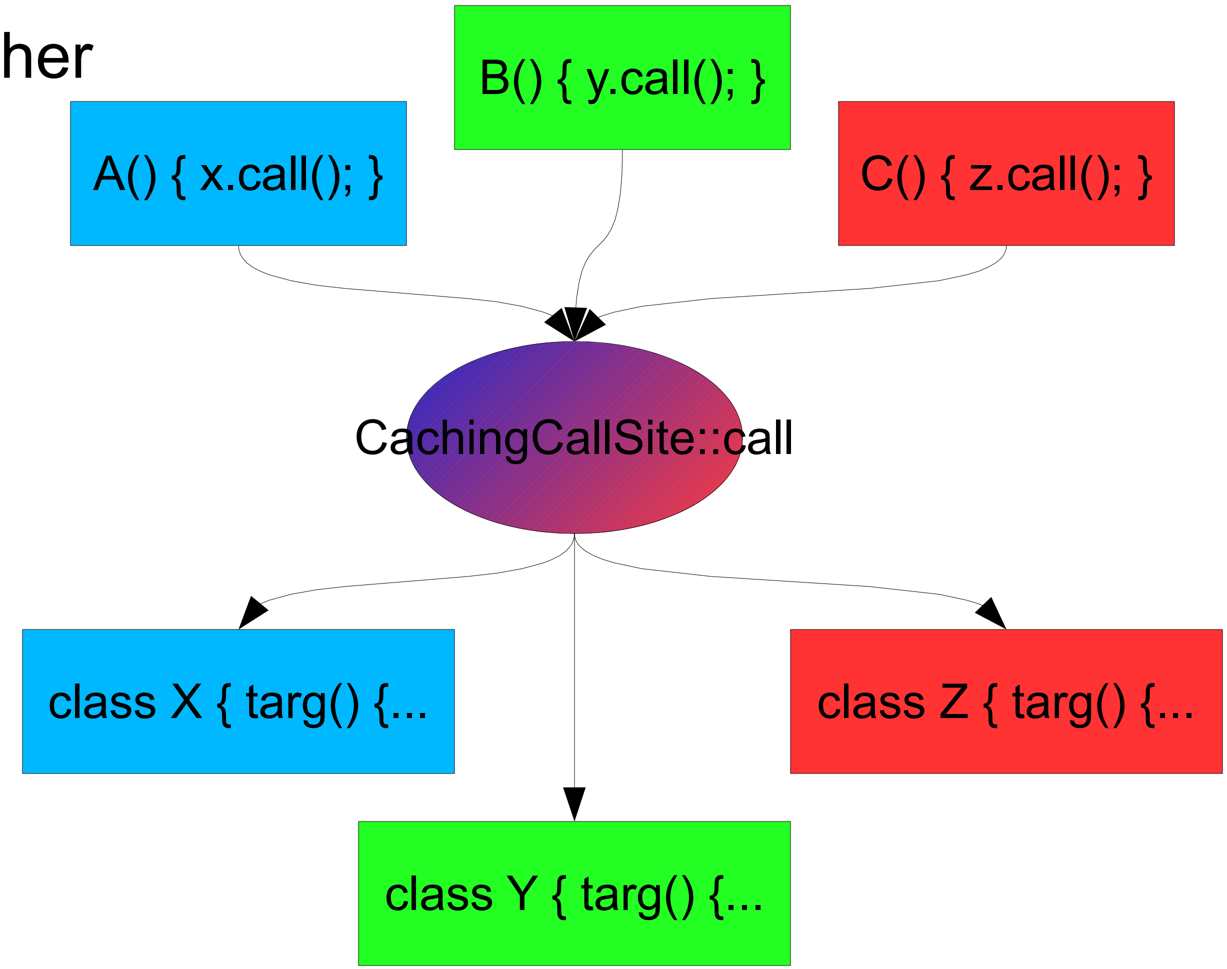
Deeper Dive

- Common Issue – FixNums
  - Allocation costs (but GC does not); final fields cost mfence
  - *Could* do much better w/JIT
  - Need ultra-stupid Escape Analysis
  - Need some (easy) JIT tweaking
  - e.g. Getting around Integer.valueOf(i) caching
- JRuby – Missed The Metal
  - Assuming CachingCallSite::call inlines (and allows further inlining)
  - Using +PrintInlining to determine
  - But flag is lying: claims inlined, but it's not
    - (issue is w/BimorphicInlining 'guessing' wrong target)
  - Confirmed w/debug Java5 & GDB on product-mode Java6
  - Confirmed w/Azul – My 1st impression: can't be inlined ever
    - (but please Charles tell me why you think it should!)

What Happened to JRuby?

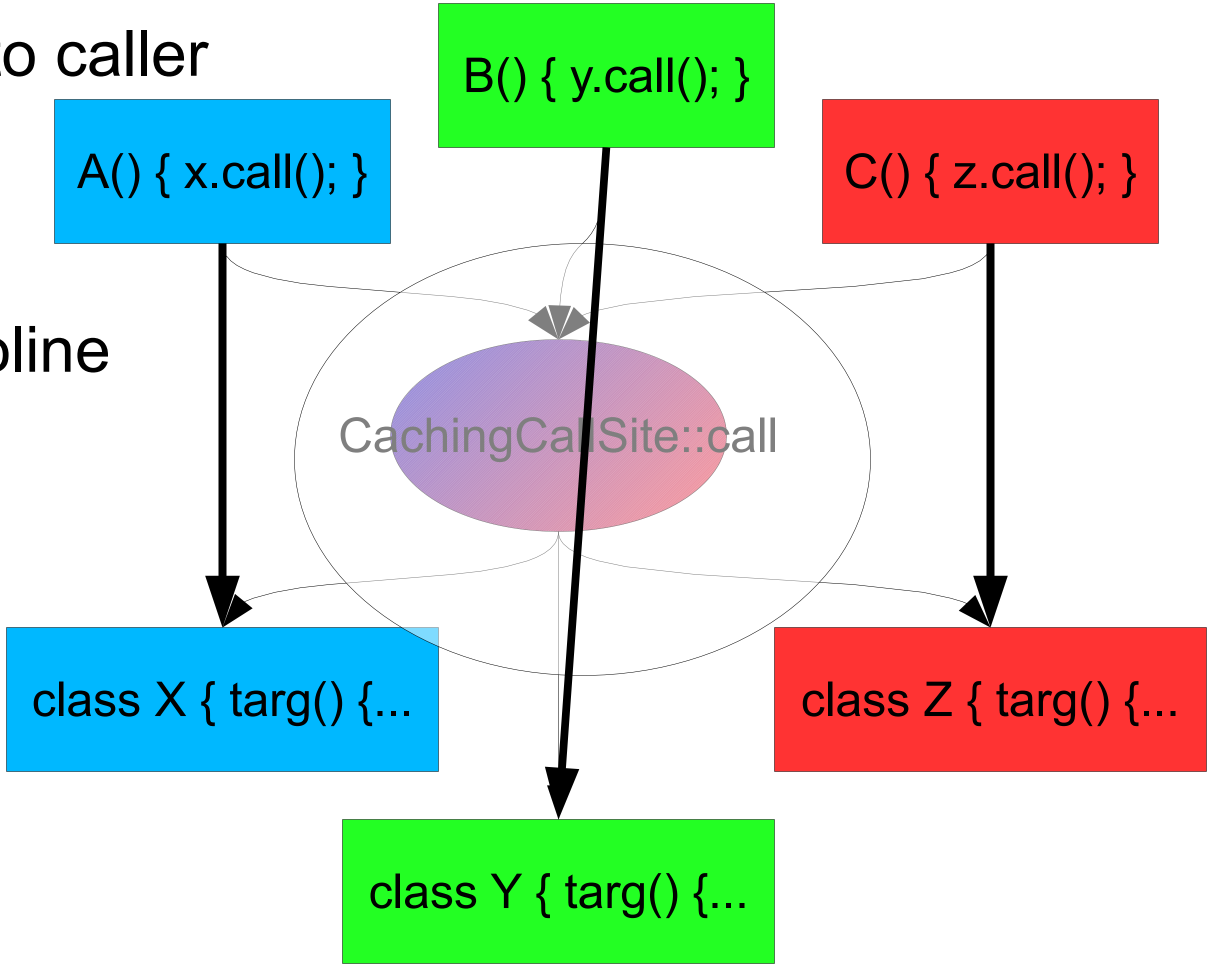
www.azulsystems.com

- Calls stitched together with trampoline



What Happened to JRuby?

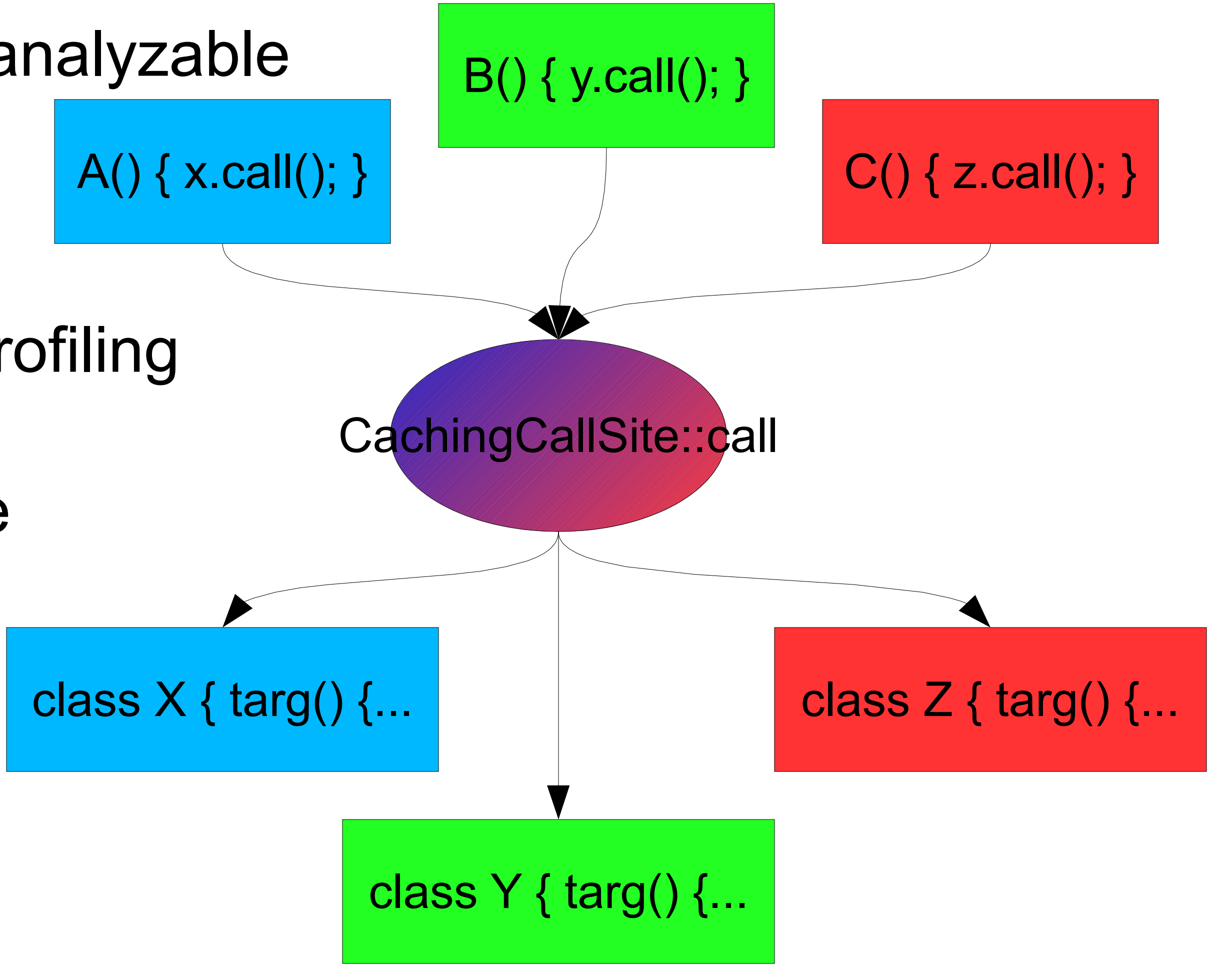
- Goal: inline 'targ' into caller
- JIT removes trampoline
- Possibly inlines



What Happened to JRuby?

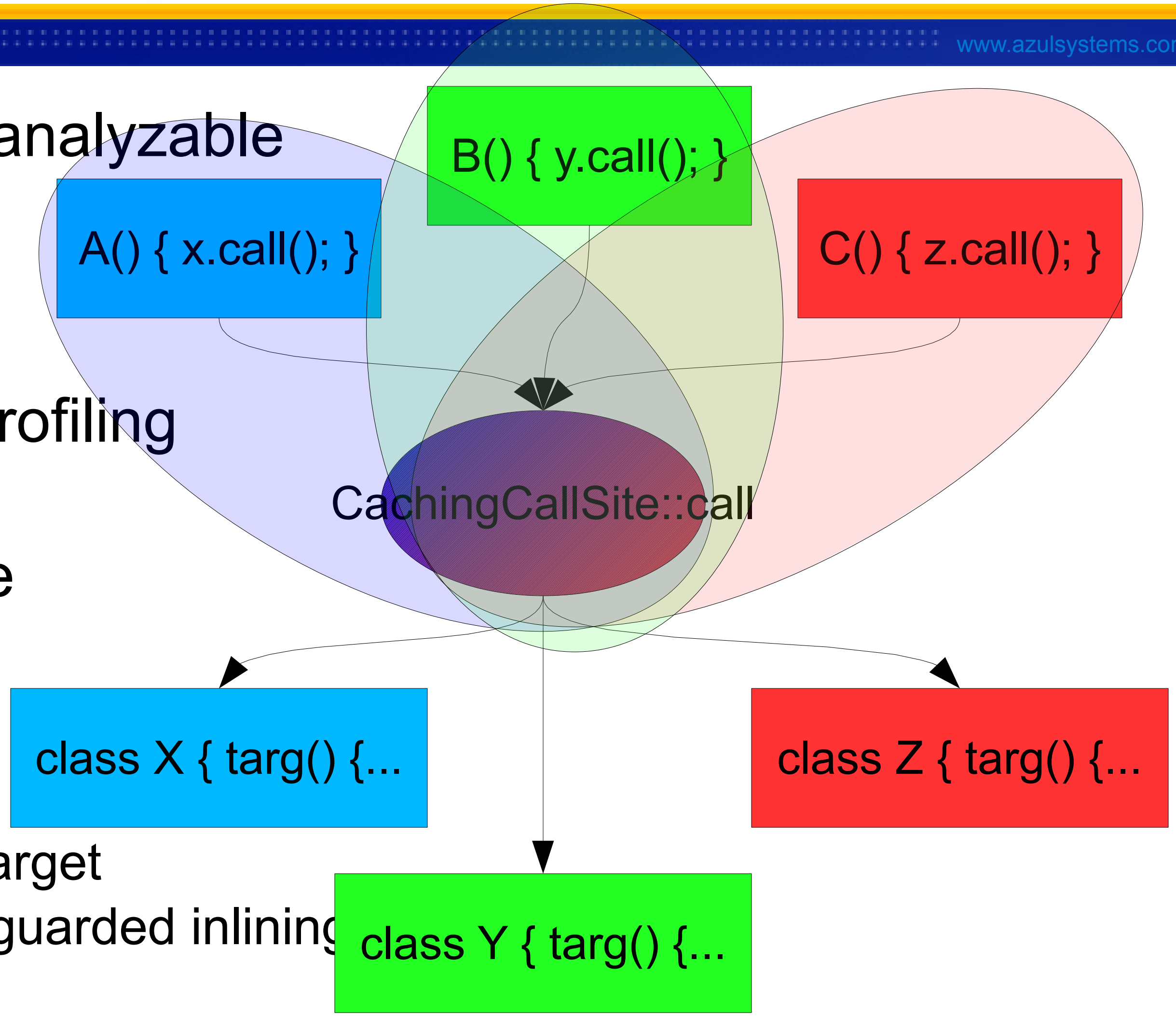
www.azulsystems.com

- Code not statically analyzable
- No inlining during profiling
- Profiling confuses callers per callsite



What Happened to JRuby?

- Code not statically analyzable
- No inlining during profiling
- Profiling confuses
  callers per callsite
- C2 inlines 'call'
  - No dominate target
  - No attempt at guarded inlining



Lessons

- CAN take “funny language” to the metal (e.g. Scala)
- Easy to be misled (JRuby)
  - Reliance on non-QA'd debugging flag +PrintInlining
  - Rules on inlining are complex, subtle
- But so much performance depends on it!
  - And exact details matter: class hierarchy, final, interface, single-target
- And how do you know about JRockit (BEA), IBM, etc?
- And heuristics change anyways,
  - what works today is dog-slow tomorrow
  - Unless your language hits the standard benchmark list
- Language/bytecode mismatch made worse by assuming
  - e.g. Uber GC or Uber Escape Analysis, or subtle final-field semantics

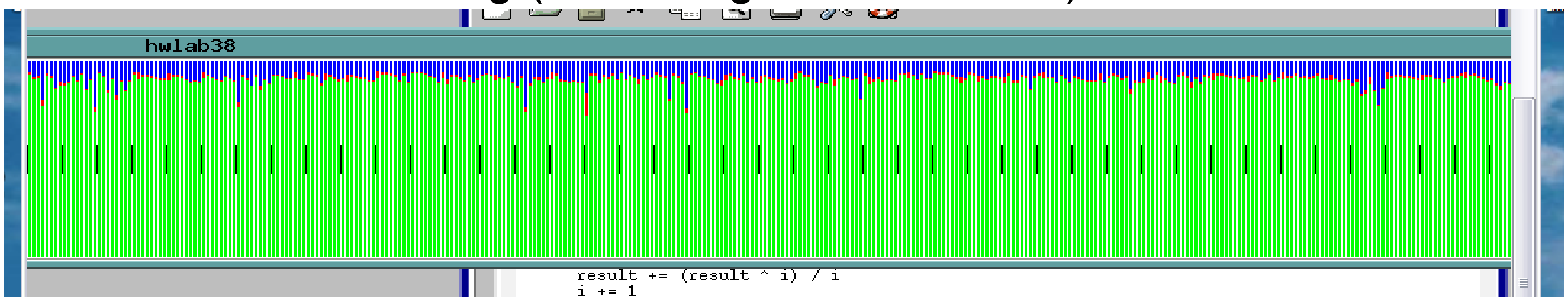
Some Suggestions

- How do you when your bytecodes are working well?
  - Lack of good tools support for telling **why** optimizer bailed out
  - Same lack for C++, Fortran for the past 50 yrs
- 1- Run microbenchmark w/hot simple code & GDB it
  - Why? So you can 'read' the asm
  - Break in w/GDB and – ***read the asm***
  - Did translation go as you expected? (no?)
  - Simplify even more...
- 2- Run Your Fav JVM w/debugging flags
  - But debugging flags can lie
  - Must cross-correlate with (1)
- 3- Run on Azul & use RTPM
  - Only sorta kidding: email me & ask for Academic Account

Round 2: Alternative Concurrency

www.azulsystems.com

- Only Clojure looked at - got ~~lazy~~ busy
- Clojure Traveling Salesmen Problem w/worker 'ants'
- Tried up to 600 'ants' on a 768-way Azul
  - Good scaling (but 20Gig/sec allocation)



- Tried contention microbenchmark
  - Performance died
  - Less TOTAL throughput as more CPUs added
  - JDK 6 Library failure? Clojure usage model failure?
    - Not graceful fall-back under pressure

Lessons?

- Too little data (e.g. no Scala), just my speculation
- Clojure-style MIGHT work, might not
  - 600-way thread-pool works well on Java also
- NOT graceful under pressure
  - Adding more CPUs should not be bad
  - Thru-put cap expected, or maybe slow degrade – but rapid falloff!?!?
  - Maybe STM becomes "graceful under pressure" later
  - (but it's been 15yrs at still not good)
- Note: complex locking schemes suffer same way
  - eg. add more concurrent DB requests, DB throughput goes up
  - ..then down, then craters
  - Why does DB not queue requests & maintain max throughput?

Future Big Problem

www.azulsystems.com

- Reliable performance “under pressure”
  - Eg adding more Threads to a hot lock drops throughput some
    - ...then stabilizes throughput as more threads added
  - Eg adding more CPUs to a wait/notifyall peaks throughput
    - ...then falls constant as most threads uselessly wakeup & sleep
- And we're working w/weak & immature concurrency libs
- Everybody has a max-throughput
  - And Fall-off Under Load is Bad (FULB™)
- Naively: just queue requests beyond saturation point
  - And maintain max-throughput
- Then why not just publish that (now reliable) max throughput
- So can predict performance as the min of max-thruput's...