Scaling Up a Real Application on Azul

Dr. Cliff Click

Distinguished Engineer
Azul Systems
www.azulsystems.com

TS-5354

Scaling Up a Real Application

Basic steps to using more cores

Benchmark (time, measure)

Analyze (discover bottlenecks)

Tune (remove bottlenecks)

[Lather, rinse, repeat ...]

45x Speedup: The Journey

Benchmarking

Easy Stuff (Thread Pools and Heap Size)

Cracking Hot Lock #1—Atomic

Cracking Hot Lock #2—Striping

Turn Down Logging

Using java.util.concurrency

Cracking Hot Lock #3—Chunking

Wrap-up

Benchmarking

What are we trying to speed up?

- App is the model checker TLA/TLC*
  - Trying to prove correctness of device drivers
  - Fine grained tasks
  - Fine sharing of data structures
    - Large shared hash table: 10 million to 100 million entries
    - Large shared work queue
  - Thread pools built-in
- Task explodes exponentially in size
  - “Big” job >> 1 week on fast P4
- “Should be” ideal for large SMP or multi-core

* Source: TLC was jointly developed by Leslie Lamport and Yuan Yu

Benchmarking

Where are we starting from?

- First up we need:
  - Repeatable self-checking setup
  - Measurable results: modify app to display performance
  - Current best performance
- Machines:
  - Native: Dual 3.2Ghz HT P4, 4Gig
  - Azul: 384-way, 128Gig
  - HotSpot 5.0 Java™ VM
- Measure starting point (often forgotten!!!)
  - Native: 14000 ops/sec
  - Azul: 2200 ops/sec

45x Speedup: The Journey

Benchmarking 14000 ops/sec

Easy Stuff (Thread Pools and Heap Size)

Cracking Hot Lock #1—Atomic

Cracking Hot Lock #2—Striping

Turn Down Logging

Using java.util.concurrency

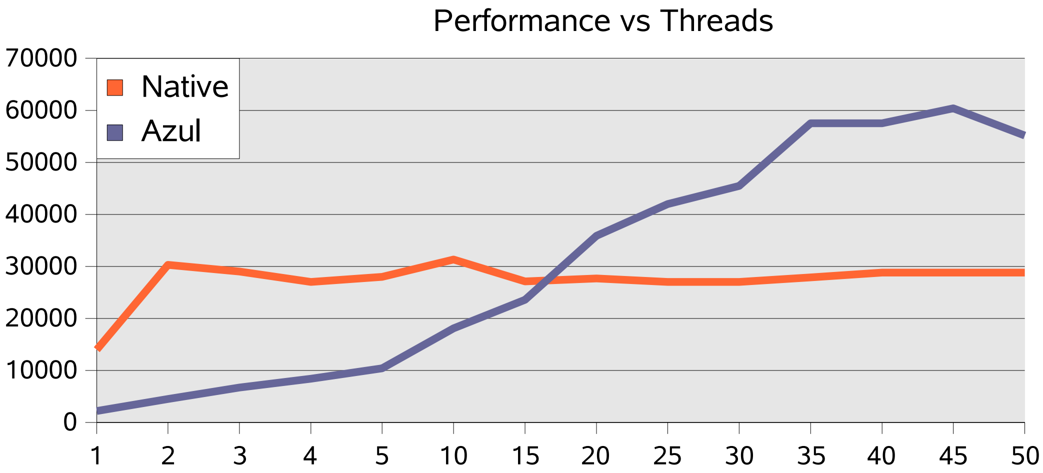
Cracking Hot Lock #3—Chunking

Wrap-up

Easy Stuff: Thread Pools

Crank up the configuration

- Many multi-threaded apps contain thread pools
- Defaults are often too low for Azul
- Try adding “-workers 4”
  - Dual-HT P4 acts like 4-cpu multi-core
- Actually:
  try a range!



Easy Stuff: Bigger Heap

Always suspect GC on bigger programs

- GC is easy to test: -verbose:gc
- See lots of GC cycles in output
  - (but not much time spent in GC)
- Crank native to 2G limit
- Crank Azul further: 8Gig heap
- Still see full GC cycles—but heap is not full!
  - Must be explicit System.gc()
  - Turn it off: -XX:+DisableExplicitGC
- Result: no improvement :-(

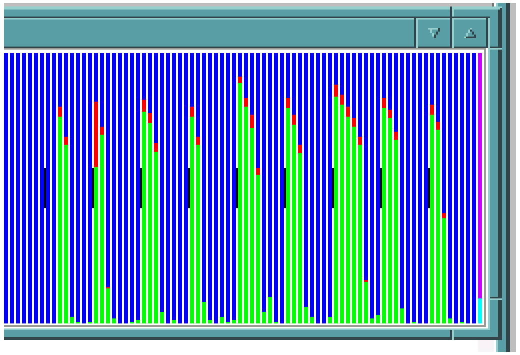
45x Speedup: The Journey

Benchmarking	14,000 ops/sec
Easy Stuff	60,000 ops/sec
Cracking Hot Lock #1—Atomic	
Cracking Hot Lock #2—Striping	
Turn Down Logging	
Using java.util.concurrency	
Cracking Hot Lock #3—Chunking	
Wrap-up	

Next Steps

Cracking Hot Lock #1

- Look at 'top'—many CPUs idle—why?
- Look at 'perfbar'—see sawtooth pattern:
  - Characteristic of a hot lock
  - Short hold time, many threads

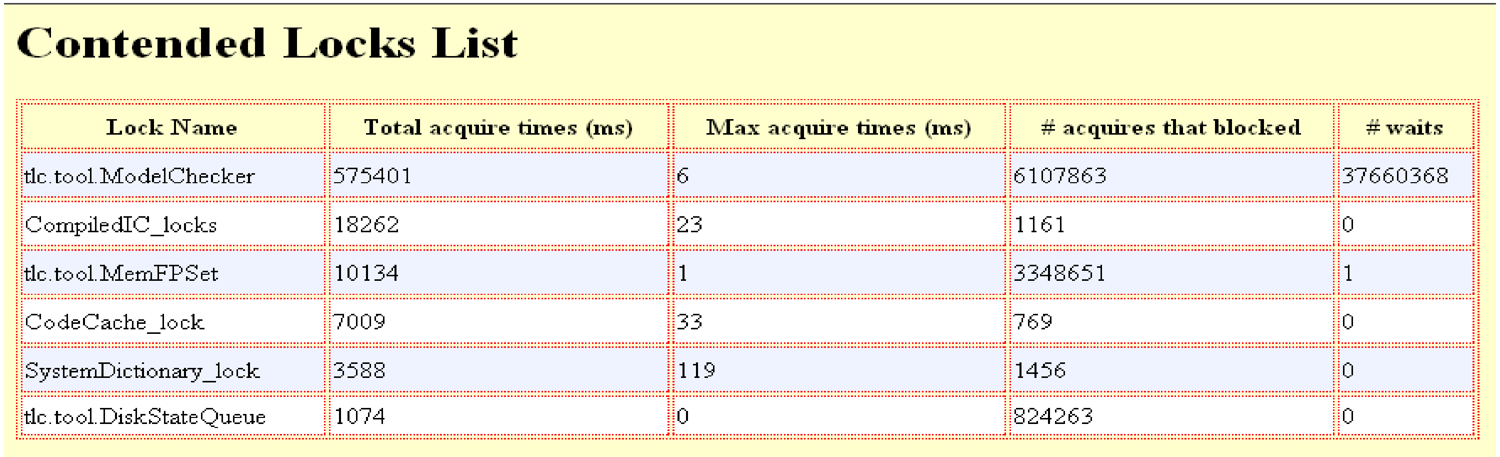


- So run again w/HTTP port opened into the JVM
- Attach with the REALTime Perf Monitor, and
- Check out “Hot Locks” list…

REALTime Performance Monitor

Exploring internal Java VM state

- The Azul RTPM can explore VM innards
  - (Yes, there is security on who can see what)
- Can see many interesting things
- We focus on contended locks:
  - Locks where the OS is sleeping/waking threads



Contended Locks List

Lock Name	Total acquire times (ms)	Max acquire times (ms)	# acquires that blocked	# waits
tlc.tool.ModelChecker	575401	6	6107863	37660368
CompiledIC_locks	18262	23	1161	0
tlc.tool.MemFPSet	10134	1	3348651	1
CodeCache_lock	7009	33	769	0
SystemDictionary_lock	3588	119	1456	0
tlc.tool.DiskStateQueue	1074	0	824263	0

tlc.tool.ModelChecker

Cracking Hot Lock #1

- Search App source for ModelChecker
  - Find ModelChecker.java
- Grep for synchronization keywords
- Find several, mostly around error conditions
- Find this:

```
synchronized void incGenStates(int n) {
  this.numOfGenStates += n;
}
```

- Common case of a synchronized counter!

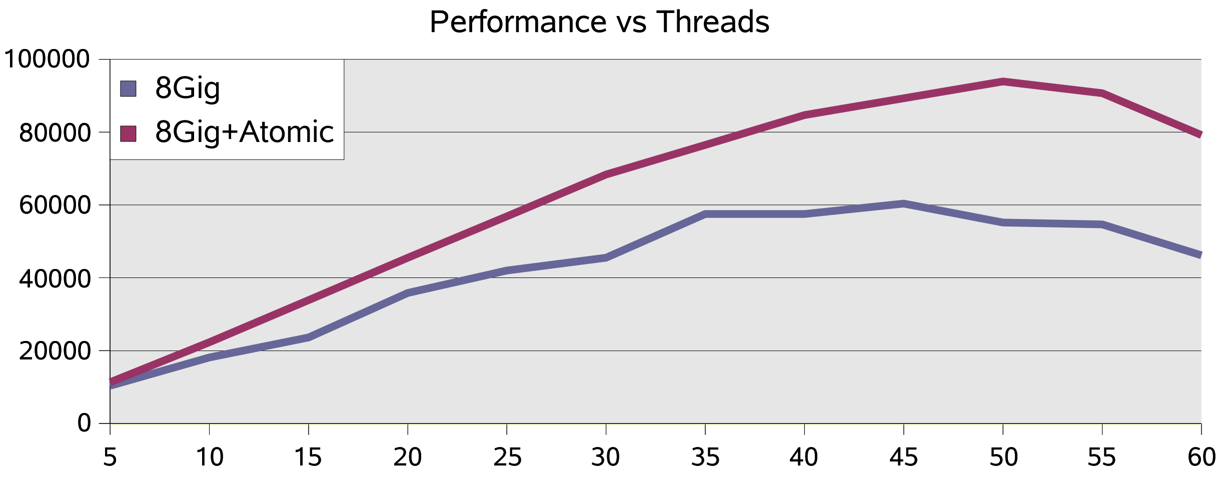
Synchronized Counters

Cracking Hot Lock #1

- Common cause of not scaling
- Sometimes no need to lock
  - e.g., Lossy performance counters
- But here, locking needed for correctness
- Use AtomicLong instead
- Implements non-blocking counter
  - Uses hardware compare-and-swap instruction
  - No locks!

Hot Lock #1 Cracked!

- New performance curve as I add threads:



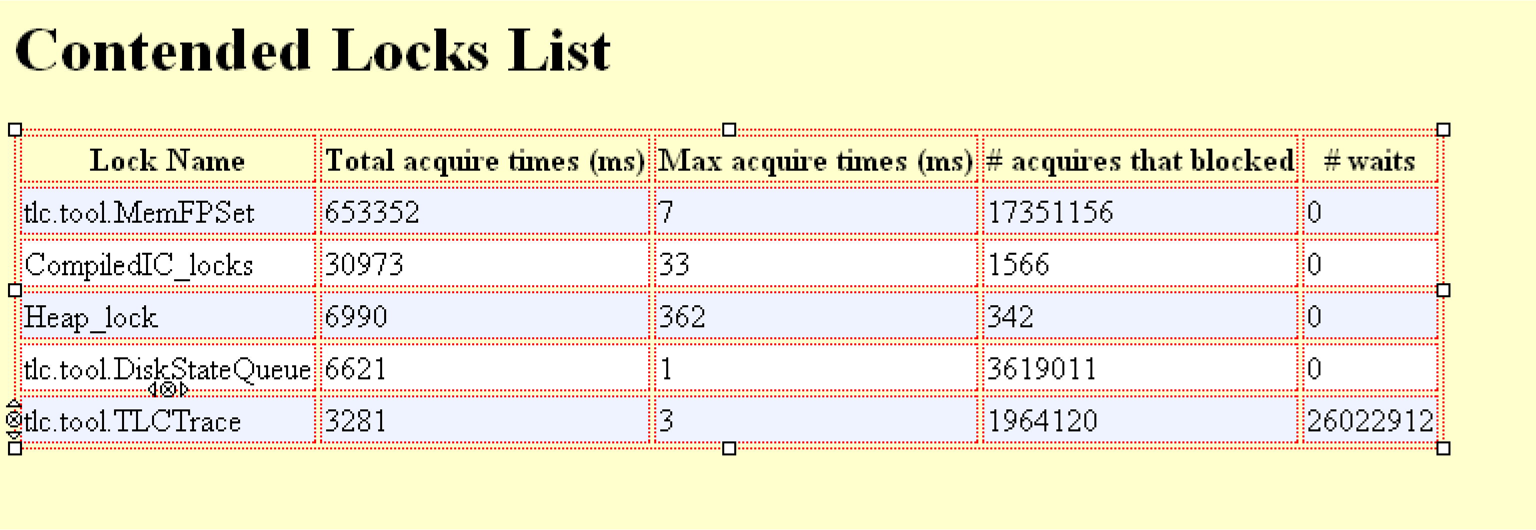
- Peak is higher and with more threads
- Still see sawtooth pattern
    - At least one more hot lock out there...

45x Speedup: The Journey

Benchmarking	14,000 ops/sec
Easy Stuff	60,000 ops/sec
Cracking Hot Lock #1	94,000 ops/sec
Cracking Hot Lock #2—Striping	
Turn down Logging	
Using java.util.concurrency	
Cracking Hot Lock #3—Chunking	
Wrap-up	

Hot Lock #2

- Back to the Contended Locks list…



Contended Locks List

Lock Name	Total acquire times (ms)	Max acquire times (ms)	# acquires that blocked	# waits
tlc.tool.MemFPSet	653352	7	17351156	0
CompiledIC_locks	30973	33	1566	0
Heap_lock	6990	362	342	0
tlc.tool.DiskStateQueue	6621	1	3619011	0
tlc.tool.TLCTrace	3281	3	1964120	26022912

- Now it’s tlc.tool.MemFPSet
  - (notice tlc.tool.ModelChecker has disappeared)

tlc.tool.MemFPSet

Cracking Hot Lock #2

- As before, find MemFPSet source code
- Find loads of synchronization keywords
- It’s some sort of sync’d home grown hashtable
- Hey! There’s a MultiFPSet file...
  - And it’s a striped wrapper around MemFPSet

```
this.theStateQueue = new DiskStateQueue(this.metadir);
// this.theStateQueue = new MemStateQueue(this.metadir);
this.theFPSet = new MultiFPSet(1);
// this.theFPSet = new DiskFPSet(-1);
// this.theFPSet = new MemFPSet();
```

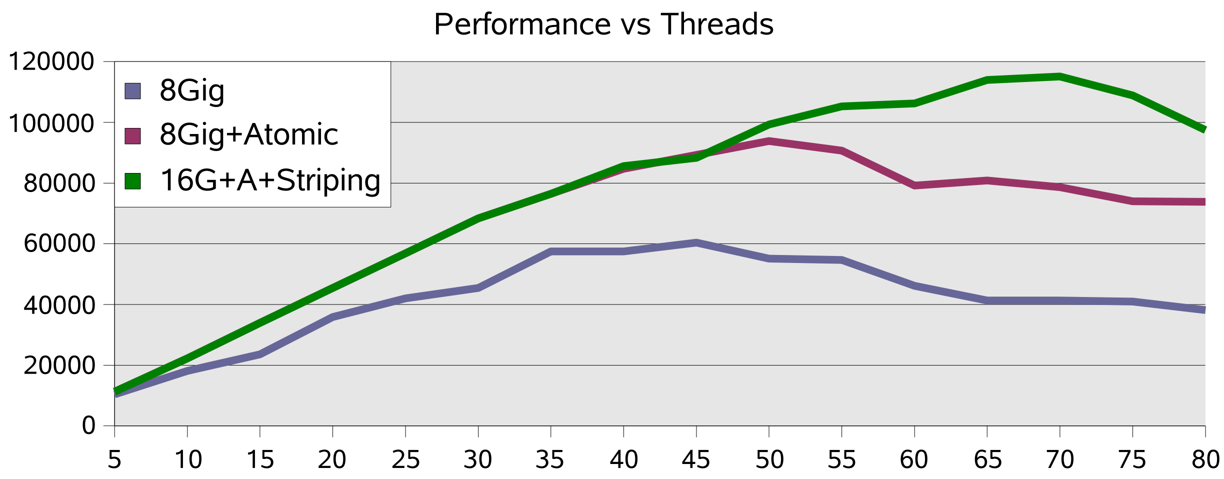
Striping a Hot Lock

Cracking Hot Lock #2

- Striping a lock: make many copies of the lock
- Threads pick one based on psuedo-random fcn
  - In this case, hash from the hashtable
- With enough locks, collisions are rare
  - So no contention
  - Locks still required in case of collision
- Hack striping to allow 1024 locks (Up from 1!)
  - Why so many? Hope to get 350+ cpus busy

Hot Lock #2 Cracked!

- New performance curve as I add threads:



- Peak is still higher and with more threads
  - (But now needs more heap also)

45x Speedup: The Journey

Benchmarking	14,000 ops/sec
Easy Stuff	60,000 ops/sec
Cracking Hot Lock #1	94,000 ops/sec
Cracking Hot Lock #2	115,000 ops/sec
Turn Down Logging	
Using java.util.concurrency	
Cracking Hot Lock #3—Chunking	
Wrap-up	

Next Steps

Back to benchmark and analyze

- General rule:

  Recheck threads and heap after cracking a lock

- Back to 'perfbar'
  - No more sawtooth
  - But all threads idle/busy in a cycle
  - 'top' shows some I/O Activity
- Oh looky—1 Gig of log file output
  - Checkpointing makes sense for a week-long job

Turning Down Logging

Too much junk I/O!

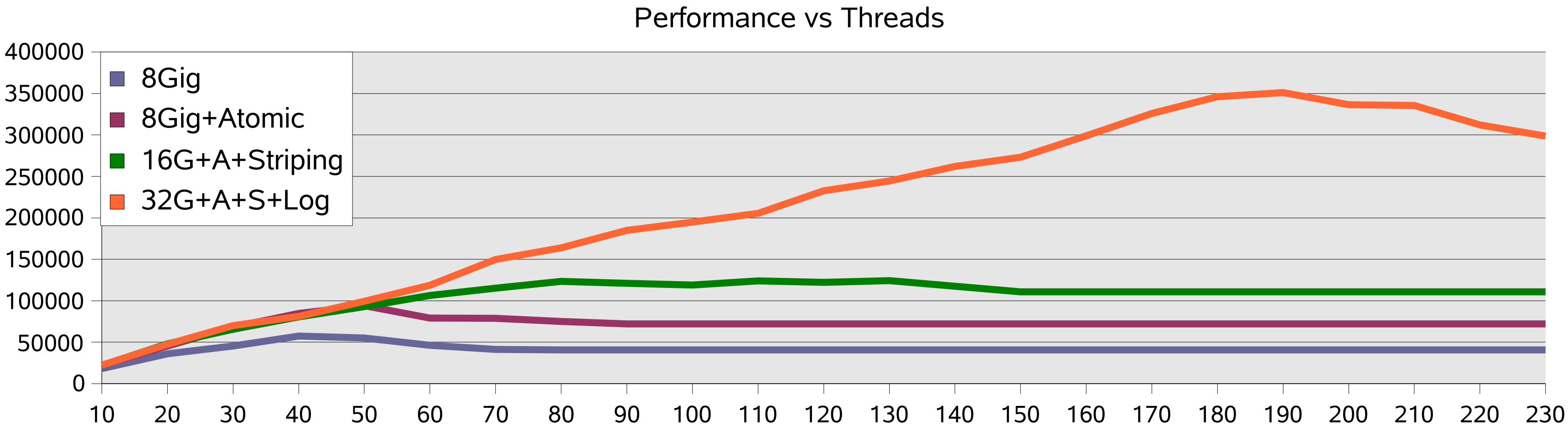
- Some logging is clearly useful
  - But maybe not 1Gig!
- Looking back—
  - I saw tlc.tool.DiskStateQueue in the HotLocks list
  - I saw this code:

```
this.theStateQueue = new DiskStateQueue(this.metadir);
// this.theStateQueue = new MemStateQueue(this.metadir);
this.theFPSet = new MultiFPSet(1);
// this.theFPSet = new DiskFPSet(-1);
// this.theFPSet = new MemFPSet();
```

Just edit and recompile…

Logging Off

- New performance curve:



- Peak is still higher
  - And needs yet more threads and heap

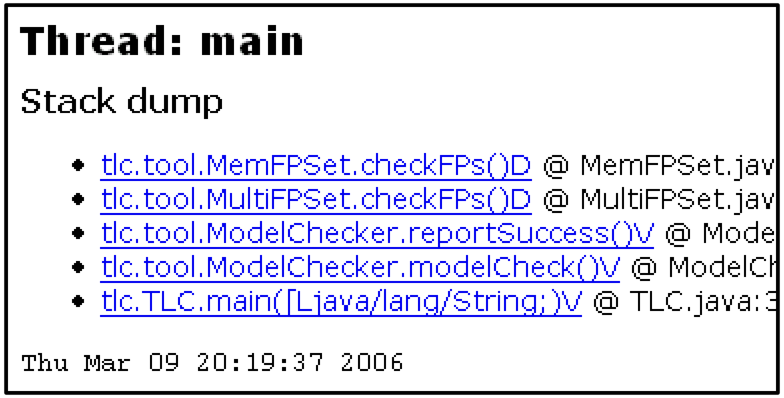
45x Speedup: The Journey

Benchmarking	14,000 ops/sec
Easy Stuff	60,000 ops/sec
Cracking Hot Lock #1	94,000 ops/sec
Cracking Hot Lock #2	115,000 ops/sec
Turn Down Logging	350,000 ops/sec
Using java.util.concurrency	
Cracking Hot Lock #3—Chunking	
Wrap-up	

Next Bottleneck

New single-threaded work phase

- Been doing "short runs"
    - So I don't have to wait a week to check results
- Hit a new end-game phase
    - After a few minutes, goes single-threaded
- Back to RTPM
    - Check thread stack of the one busy thread



Thread: main

Stack dump

- tlc.tool.MemFPSet.checkFPs()D @ MemFPSet.jav
- tlc.tool.MultiFPSet.checkFPs()D @ MultiFPSet.jav
- tlc.tool.ModelChecker.reportSuccess()V @ Mode
- tlc.tool.ModelChecker.modelCheck()V @ ModelCh
- tlc.TLC.main([Ljava/lang/String;)V @ TLC.java:3

Thu Mar 09 20:19:37 2006

java.util.concurrent

Making it parallel!

- Thread is in MemFPSet.checkFPs
  - Called from MultiFPSet
- But MemFPSet is now striped!
- “Should be” trivial to parallelize w/j.u.concurrent
- Compile w/ javac 5.0
  - err…fix uses of new enum keyword
- Slap down some boilerplate parallel iteration*
- And voilà!—instant parallel end-phase

*Goetz: Java Concurrency in Practice

Making an Iterator Parallel

- Old:

```
for (int i = 0; i < sets.length; i++)
 res = Math.max(res, sets[i].checkFPs());
```

- New*:

```
Executor exec = Executors.newFixedThreadPool(numThreads);
CompletionService<Double> cs = new ...<Double>(exec);
// Submit the jobs in parallel
for (int i = 0; i < sets.length; i++) {
 cs.submit(new Callable<Double>() {
  public Double call() throws IOException {
   return sets[i].checkFPs();
  }
 });
}
for (int i = 0; i < this.sets.length; i++)
 res = Math.max(res, cs.take().get());
```

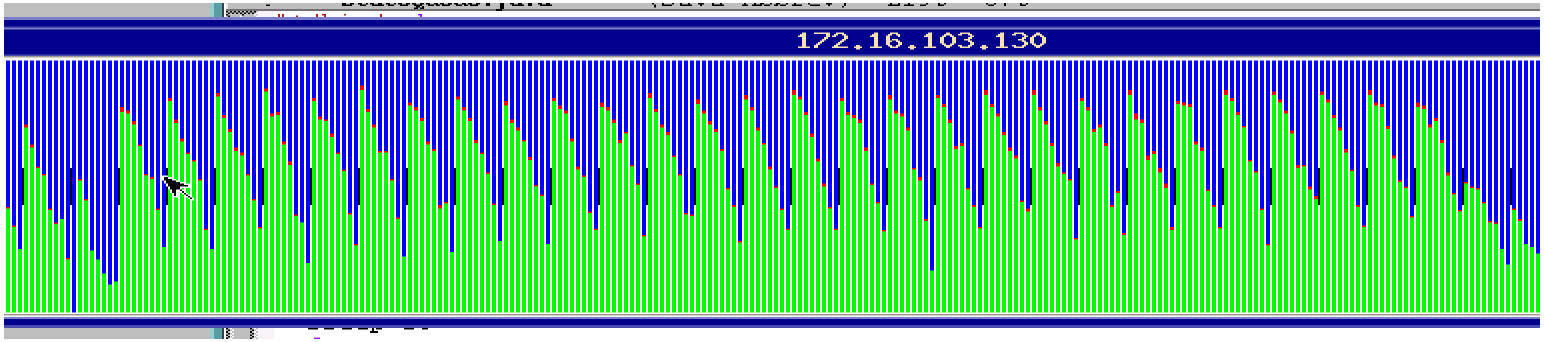
*Goetz: Java Concurrency in Practice

45x Speedup: The Journey

Benchmarking	14,000 ops/sec
Easy Stuff	60,000 ops/sec
Cracking Hot Lock #1	94,000 ops/sec
Cracking Hot Lock #2	115,000 ops/sec
Turn Down Logging	350,000 ops/sec
Using java.util.concurrency	
Cracking Hot Lock #3—Chunking	
Wrapup	

Cracking Hot Lock #3

- Revert back to full size problem
  - Short problem now much too short to benchmark
  - End phase is perfectly parallelizable (and short)
- Main phase still does not scale past 200 cores
- 'perfbar' shows sawtooth pattern again
  - Back to the “Hot Locks” list



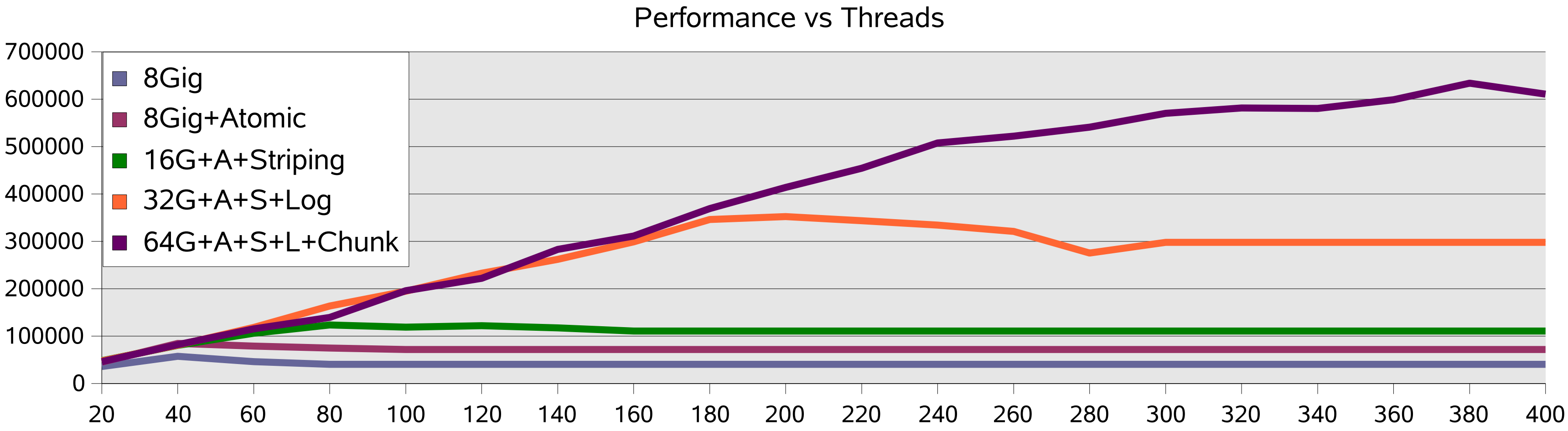
tlc.tool.MemStateQueue

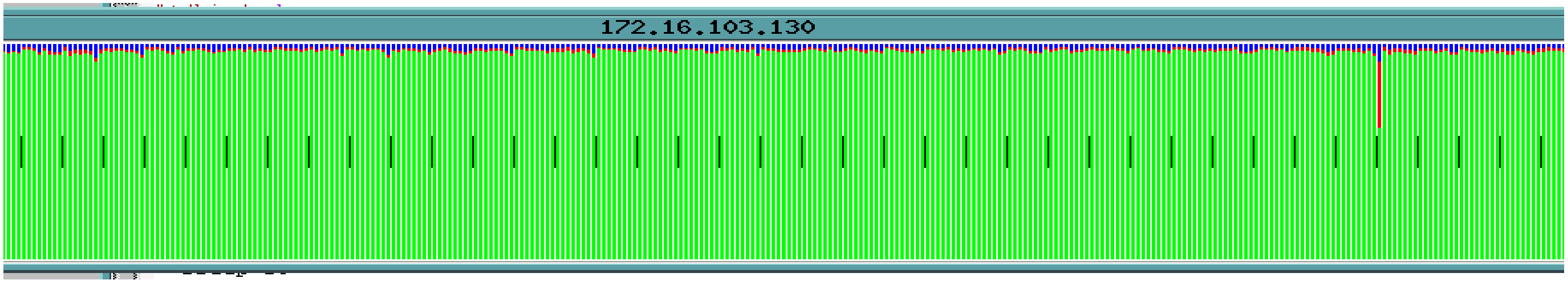
Cracking Hot Lock #3

- It’s a worklist, a double-ended ring buffer
- Contention when 200+ threads try to access
- Worklist lock is hot
  - So take it less often!
- Idea: grab more than 1 piece of work at a time
- Take (or put) 32 units of work
  - Or just take 1 (or put all) if worklist is low on work
- Includes small thread-local worklist

Chunked Worklist

- New performance curve:





45x Speedup: The Journey

Benchmarking	14,000 ops/sec
Easy Stuff	60,000 ops/sec
Cracking Hot Lock #1	94,000 ops/sec
Cracking Hot Lock #2	115,000 ops/sec
Turn Down Logging	350,000 ops/sec
Using java.util.concurrency	
Cracking Hot Lock #3	630,000 ops/sec
Wrapup	

Summary

- Get a stable setup
  - Need repeatable results!
  - Need decent self-checking
- Get an initial value
  - So you can tell progress!
- Try the easy stuff: thread pools and heap size
- Locks:
  - Cracking them makes the code complex, subtle
  - Crack only what you need, clearly, carefully
  - Leave the rest alone

Summary

- After cracking a lock:
  - Recheck thread-pools and heap sizes
  - Oh yeah, check correctness (race conditions)
- Some techniques for cracking locks:
  - Use AtomicLong instead of synchronized counters
  - Using java.util.concurrent
  - Striping locks
  - Chunking work
- Might need to crack several locks to see any gain
  - The last bottleneck isn't gone until CPUs are busy ;-)

Postlude: 54x Speedup

- Peak now: 790,000 ops/sec
- Found an instance of non-chunked worklist
  - New task was being directly inserted into global list
  - Using the chunked interface gives better locality
- Bumped hashtable striping to 4096
- Parallelized I/O better
- More GC tuning: app uses 60Gig in 10 sec
  - GC still 20–30% of runtime
- Further improvements possible

For More Information

- www.azulsystems.com
    - See the webinar at our website
- See our booth # 622
- Also BOF-0377—Confessions of a JVM Writer

Q&A

Scaling Up a Real Application on Azul

Dr. Cliff Click

Distinguished Engineer
Azul Systems
www.azulsystems.com

TS-5354