A JVM Does That???

Cliff Click
www.azulsystems.com/blogs

AZUL SYSTEMS®

A JVM Does That???

- Been a JVM Engineer for over a decade
- I'm still amazed at what goes in a JVM
- Services have increased over time
- Many new services painfully "volunteered" by naive change in specs

Some JVM Services

- High Quality GC
  - Parallel, Concurrent, Collection
  - Low total allocation cost
- High Quality Machine Code Generation
  - Two JITs, JIT'd Code Management, Profiling
  - Bytecode cost model
- Uniform Threading & Memory Model
  - Locks (synchronization), volatile, wait, notify
- Type Safety

Some JVM Services

- Dynamic Code Loading
  - Class loading, Deoptimization, re-JIT'ing
- Quick high-quality Time Access
  - System.currentTimeMillis
- Internal introspection services
  - Reflection, JNI, JVMTI, JVMDI/JVMPI, Agents
- Access to huge pre-built library
- Access to OS
  - threads, scheduling, priorities, native code

Too Many Services?

- Where did all this come from?
- Mostly incrementally added over time
- The Language, JVM, & Hardware all co-evolved
  - e.g. incremental addition of finalizers, JMM, 64-bits
  - Support for high core-count machines

Why Did We Add All These Services?

- Because Illusions Are Powerful Abstractions

The 'V' in JVM

- "Virtual" – Its a Great Abstraction
- Programmers focus on value-add elsewhere
- JVM Provides Services
- The selection of Services is ad-hoc
  - Grown over time as needed
  - Some services are unique to Java or the JVM
  - Many services overlap with existing OS services
    - But sometimes have different requirements

Agenda

- Introduction (just did that)
- **Illusions We Have**
- Illusions We Think We Have or Wish We Had
- Sorting Our Illusions Out

Illusion: Infinite Memory

- Garbage Collection – The Infinite Heap Illusion
  - Just allocate memory via 'new'
  - Do not track lifetime, do not 'free'
  - GC figures out What's Alive and What's Dead
- Vastly easier to use than malloc/free
  - Fewer bugs, quicker time-to-market
- Enables certain kinds of concurrent algorithms
  - Just too hard to track liveness otherwise

Illusion: Infinite Memory

- GC have made huge strides in the last decade
  - Production-ready robust, parallel, concurrent
  - Still major user pain-point
    - Too many tuning flags, GC pauses, etc
  - Major Vendor point of differentiation, active dev
  - Throughput varies by maybe 30%
  - Pause-times vary over 6 orders of magnitude
    - (Azul GPGC: 100's of Gig's w/10msec)
    - (Stock full GC pause: 10's of Gig's w/10sec)
    - (IBM Metronome: 100's Megs w/10microsec)

Illusion: Bytecodes Are Fast

- Class files are a lousy way to describe programs
- There are better ways to describe semantics than Java bytecodes
  - But we're stuck with them for now
  - Main win: hides CPU details
- Programmers rely on them being "fast"
- It's a big Illusion: Interpretation is slow
- JIT'ing brings back the "expected" cost model

Illusion: Bytecodes Are Fast

- JVMs eventually JIT bytecodes
  - To make them fast!
  - Some JITs are high quality optimizing compilers
    - Amazingly complex beasties in their own rights
  - i.e. JVMs bring "gcc -O2" to the masses
- But cannot use "gcc"-style compilers directly:
  - Tracking OOPs (ptrs) for GC
  - Java Memory Model (volatile reordering & fences)
  - New code patterns to optimize

Illusion: Bytecodes Are Fast

- JIT'ing requires Profiling
  - Because you don't want to JIT everything
- Profiling allows focused code-gen
- Profiling allows better code-gen
  - Inline whats hot
  - Loop unrolling, range-check elimination, etc
  - Branch prediction, spill-code-gen, scheduling
- JVMs bring Profiled code to the masses!

Illusion: virtual calls are fast

- C++ avoids virtual calls – because they are slow
- Java embraces them – and makes them fast
  - Well, mostly fast – JIT's do Class Hierarchy Analysis
  - CHA turns most virtual calls into static calls
  - JVM detects new classes loaded, adjusts CHA
    - May need to re-JIT
  - When CHA fails to make the call static, *inline caches*
  - When IC's fail, virtual calls are back to being slow

Illusion: Partial Programs Are Fast

- JVMs allow late class loading, name binding
  - i.e. classForName
- Partial programs are as fast as whole programs
  - Adding new parts in (e.g. Class loading) is "cheap"
  - May require: deoptimization, re-profiling, re-JIT
  - Deoptimzation is a hard problem also

Illusion: Consistent Memory Models

- ALL machines have different memory models
  - The rules on visibility vary widely from machines
  - And even within generations of the same machine
  - X86 is very conservative, so is Sparc
  - Power, MIPS less so
  - IA64 & Azul very aggressive
- Program semantics depend on the JMM
  - So must match the JMM
  - Else *program meaning* would depend on hardware!

Illusion: Consistent Memory Models

- Very different hardware memory models
- None match the Java Memory Model
- The JVM bridges the gap -
  - While keeping normal loads & stores fast
  - Via combinations of fences, code scheduling, placement of locks & CAS ops
  - Requires close cooperation from the JITs
  - Requires detailed hardware knowledge

Illusion: Consistent Thread Models

- Very different OS thread models
  - Linux, Solaris, AIX
  - But also cell phones, iPad, etc
- Java just does 'new Thread'
  - On micro devices to 1000-cpu giant machines
  - and *synchronized*, *wait, notify, join,* etc, all just work

Illusion: Locks are Fast

- Contended locks obviously block and must involve the OS
  - (Expect fairness from the OS)
- Uncontended locks are a dozen nano's or so
  - Biased locking: ~2-4 clocks (when it works)
  - *Very* fast user-mode locks otherwise
- Highly optimized because *synchronized* is so common

Illusion: Locks are Fast

- People don't know how to program concurrently
  - The 'just add locks until it works' mentality
  - i.e. Lowest-common-denominator programming
  - So locks became common
  - So JVMs optimized them
- This enabled a particular concurrent programming style
- And we, as an industry, learned alot about concurrent programming as a result

Illusion: Quick Time Access

- System.currentTimeMillis
  - Called *billions* of times/sec in some benchmarks
  - Fairly common in all large java apps
  - Real Java programs expect that:
    if T1's Sys.cTM < T2's Sys.cTM
    then T1 <<<*happens_before* T2
- But cannot use, e.g. X86's "tsc" register
  - Value not coherent across CPUs
  - Not consistent, e.g. slow ticking in low-power mode
  - Monotonic per CPU – but not per-thread

Illusion: Quick Time Access

- System.currentTimeMillis
  - Switching from fastest linux gettimeofday call
    - (mostly-user-mode atomic time struct read)
    - gettimeofday gives *quality* time
  - To a plain *load* (updated by background thread)
  - Was worth 10% speed boost on key benchmark
- Hypervisors like to "idealize" tsc :
  - Means: uniform monotonic ticking
  - Means: slows access to tsc by 100x?

Agenda

- Introduction (just did that)
- Illusions We Have
- **Illusions We Think We Have or Wish We Had**
- Sorting Our Illusions Out

Illusions We'd Like To Have

- Infinite Stack
  - e.g. Tail calls. Useful in some functional languages
- Running-code-is-data
  - e.g. Closures
- 'Integer' is as cheap as 'int'
  - e.g. Auto-boxing optimizations
- 'BigInteger' is as cheap as 'int'
  - e.g. Tagged integer math, silent overflow to infinite precision integers

Illusions We'd Like To Have

- Atomic Multi-Address Update
  - e.g. Software Transactional Memory
- Fast alternative call bindings
  - e.g. invokedynamic

Illusions We Think We Have

- This mass of code is maintainable:
  - HotSpot is approaching 15yrs old
  - Large chunks of code are fragile
    - (or very 'fluffy' per line of code)
  - Very slow new-feature rate-of-change
- Azul Systems has been busy rewriting lots of it
  - Many major subsystems are simpler, faster, lighter
  - >100K diffs from OpenJDK

Illusions We Think We Have

- Thread priorities
  - Mostly none on Linux without *root* permission
  - But also relative to entire machine, not JVM
  - Means a low-priority JVM with high priority threads
    - e.g. Concurrent GC threads trying to keep up
  - ...can starve a medium-priority JVM
- Write-once-run-anywhere
  - Scale matters: programs for very small or very large machines are different

Illusions We Think We Have

- Finalizers are Useful
  - They suck for reclaiming OS resources
    - Because no timeliness guarantees
    - Code "eventually" runs, but might be never
    - e.g. Tomcat requires a out-of-file-handles situation trigger a FullGC to reclaim finalizers to recycle OS file handles
- What other out-of-OS resources situations need to trigger a GC?
- Do we really want to code our programs this way?

Illusions We Think We Have

- Soft, Phantom Refs are Useful
  - Again using GC to manage a user resource
  - e.g. Use GC to manage Caches
- Low memory causes
  rapid GC cycles causes
  soft refs to flush causes
  caches to empty causes
  more cache misses causes
  more application work causes
  more allocation causes
  rapid GC cycles

Agenda

- Introduction (just did that)
- Illusions We Have
- Illusions We Think We Have or Wish We Had
- **Sorting Our Illusions Out**

Services Summary

- Services provided by JVM
  - GC, JIT'ing, JMM, thread management, fast time
  - Hiding CPU details & hardware memory model
- Services provided below the JVM (OS)
  - Threads, context switching, priorities, I/O, files, virtual memory protection,
- Services provided above the JVM (App)
  - Threadpools & worklists, transactions, cypto, caching, models of concurrent programming
  - Alt languages: new dispatch, big ints, alt conc

Move to OS: Fast Quality Time

- JVM provides *fast quality* time
  - Fast not quality from X86 'tsc'
  - Quality not fast from OS gettimeofday
- This should be an OS service
  - Tick memory word 1000/sec
    - Update with kernel thread or timer
  - Read-only process-shared page
  - This CTM is a coherent across CPUs on a clock-cycle basis

Move to OS: Thread Priorities

- OS provides thread priorities at the process level
  - Higher priority JVMs can/should starve lower ones
- JVM also needs thread priorities within-process
  - GC threads need cycles before mutator threads
    - Or else that concurrent GC will won't be concurrent
    - And the mutator will block for a GC cycle
  - JIT threads need cycles
    - Or else the 1000-runnable threads will starve the JIT
    - And the program will always run interpreted

Move to OS: Thread Priorities

- Right now Azul is faking thread priorities
  - With duty-cycle style locks & blocks
  - Required for a low-pause concurrent collector
- Per-process Thread Priorities belong in the OS
  - OS already does process priorities & context switches
  - Also, cannot raise thread priorities without 'root'
  - Lowering mutator priorities changes behavior wrt non-Java processes

Keep Above JVM: Alternative Concurrency

- JVM provides thread management, fast locks
- Many new langs have new concurrency ideas
  - Actors, Msg-passing, STM, Fork/Join are a few
  - JVM too big, too slow to move fast here
  - Should experiment 'above' the JVM
  - ...at least until we get some concensus on **The Right Way To Do Concurrency**
  - Then JVM maybe provides building blocks
    - e.g. park/unpark or a specific kind of STM

Move to JVM: Fixnums

- Fixnums belong in the JVM, not language impl
- JVM provides 'int' & 'long'
  - Many languages want 'ideal int'
  - Obvious java translation to infinite math is inefficient
    - Really want some kind of tagged integer
    - Requires JIT support to be really efficient
  - I think "64bits ought to be enough for anybody"
    - You (app-level programmer) know if you might need more
    - Don't make everybody else pay for it

Keep in JVM: GC, JIT'ing, JMM, Type Safety

- JIT'ing (by itself) belongs above the OS and below the App – so in the JVM
- GC requires deep hooks into the JIT'ing process
  - And also makes sense below the App
- The JMM requires deep hooks into the JIT also
  - And again (mostly) makes sense below the App
  - Some alternative concurrency models
    would expose weaker MMs to the App,
    would enable faster, cheaper hardware
    – but this is still going require close JIT cooperation

Move Above JVM: OS Resource Lifetime

- Move outside-the-JVM resource lifetime control out of Finalizers
  - Make the app do e.g. ref-counting or 'arenas' or other lifetime management
  - Do not burden GC with knowledge that more of resource 'X' can be had running finalizers
- Move weak/soft/phantom refs to the App
  - GC should not change application semantics

Summary

OS	JVM	Application
Virtual-Memory	Type-Safe Memory GC JIT'ing & Code Management	Alternative-Concurrency STM / FJ
Context-Switches	JMM Fast Locks Thread Management Thread Priorities Fast Time	Fixnums
Files, I/O	OS Resource Management	Tail Calls Closures

Cliff Click
http://www.azulsystems.com/blogs

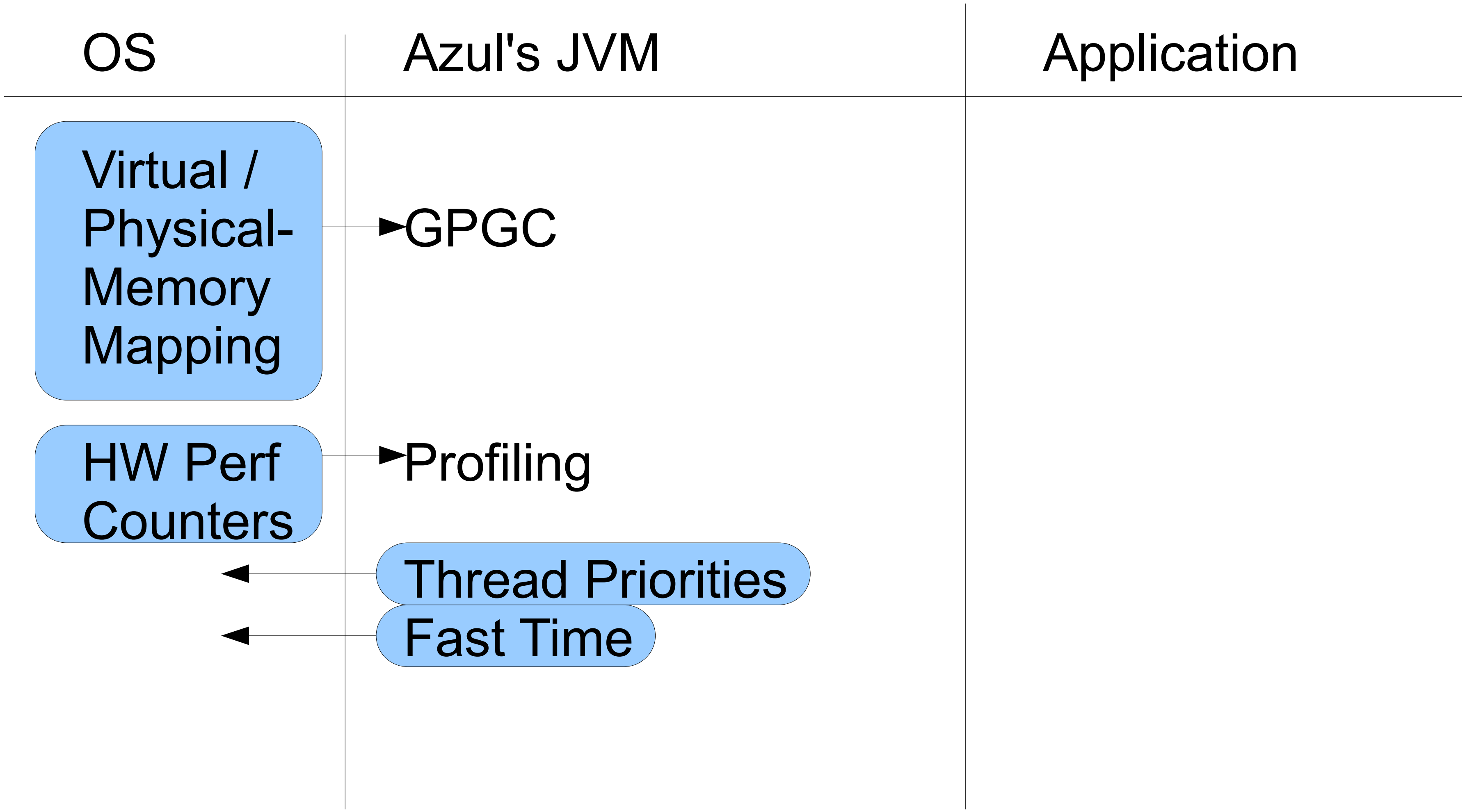
Move To JVM (Azul): Virtual / Physical Mappings

- Azul's GPGC does aggressive virtual-memory to physical-memory remappings
  - Tbytes/sec remapping rates
  - **`mmap()`** & friends too slow
- Need OS hacks to expose hardware TLB
  - Still safe across processes
  - But within process can totally screw self up

Move To JVM (Azul): Hardware Perf Counters

- JVM is already doing profile-directed compilation
  - Natural consumer of HW Perf Counters
- JVM can map perf counters to bytecodes
  - JIT's code, manages JIT'd code
  - "hotcode" mapped back to user's bytecodes
- Want quickest & thin-est way to expose HW perf counters to JVM

Summary (Azul)



Cliff Click
http://www.azulsystems.com/blogs

Summary

There's Work To Do

(full employment contract for JVM engineers)

PS: Azul is hiring compiler & runtime engineers

Cliff Click
http://www.azulsystems.com/blogs