Bits of Advice For the JVM Writer

THE ERA OF UNBOUND COMPUTE IS NOW

Dr Click, Distinguished Engineer

JVMs are Big Complex Beasties

- Or Really Small
- Depends on the Feature Set
- Many features interact in Bad Ways
  - Usually interactions not obvious
- I went the Big Desktop/Server route
  - Very different choices from the cell-phone guys
  - Must solve a different set of problems
- I suspect most of this applies to .Net as well

Some choices to make...

- Portable or not
  - Native code vs JIT'd code
  - Calling conventions
  - Threads, POSIX, stacks
- Footprint
  - Phone (2M), desktop (2G), server (16G+)
- X86 vs RISC
- JIT or Interpret (or no Interpreter!)
- Multi-threaded
  - Cooperative vs preemption
- Multi-CPU

Some JIT choices to make...

- JIT
  - None? (gives up peak performance)
  - 'stage0' or template style?
  - 'stage1', light optimization, linear-scan allocator?
  - 'stage2', all the optimizations, graph-coloring?
- Portable or targeted (usual X86 + Vendor)?
- Intercalls with native?
- Mixed-mode with Interpreter?
  - Also messes with calling convention
- Class loading vs inlining-non-finals?

More choices to make...

- Interpreter
- Simple, software only
  - Pure C
  - Gcc label vars (about 2x faster than pure C)
  - Pure asm (about 2x faster again)
- Hardware support (ARM jazelle)
- Fancy: inline & schedule dispatch
  - Seen at least 2 different ways to slice this
- Requires stack oriented layout, bad for JITs
- or none at all...

More JIT choices to make...

- Stage-0 JIT
- Template generation, very fast low quality code
- No funny stack layouts
- Easy calling conventions for all
- Still slower than just interpreting run-once code
- Lots of run-once code at startup
- Lots of bulky code to startup
  - Big footprint
  - Can throw away & regenerate

Some GC choices to make...

- GC
  - Simple? (single gen, StopTheWorld?)
  - Fast? (Throughput? Low pause?)
  - Exact? (allows moving objects, compaction, de-frag, bump pointer allocation)
  - Conservative? (No need to track all objects)
  - Tried & true? (e.g. Mark/Sweep)
  - Fancy new algorithm?
- Parallel? (really hard!)
- Concurrent? (really really hard!)
- Both? (really^4 hard!)

More choices to make...

- Stop-Anywhere vs Safepoints
- Stop-Anywhere
  - Oop Maps at each PC? (bulky data)
  - Interpret instructions? (hard/slow on X86)
  - Fixed oop registers / stack areas? (bad register usage)
- Safepoints
  - Cooperative? (polling costs?)
  - Preempt at wrong place?
  - Roll forward vs step forward vs interpret?
  - Polling in software vs hardware?

More choices to make...

- Multi-threading costs
  - All operations no longer atomic
  - May be preempted inconveniently
  - Need locking (never needed it before!)
- Hard to find the SP+PC of remote thread
  - Common problem of OS's
    - ! **Very** suprising to me, until I tried it !
    - Fails (w/low frequency) on many “robust” OSes!
    - Caught in page fault handler, nested in tlb handler, nested in stack overflow handler nested in ....
- GC requires SP (& usually PC) for stack roots
- Threads can block for I/O, or deadlock

More choices to make...

- Multi-CPUs?
  - Now need Atomic operations
  - Coherency, memory fences
  - Low-frequency data race bugs
- Need *scalable* locks in JVM
  - Not just correctness locks
- Need *scalable* locks for Java/JIT'd code
  - Spinning & retries
  - 'fair' locks under super high contention
- GC can be in-progress when a thread awakens
  - Threads now need to take a form of GC lock to run

More choices to make...

- Long (64bit int) math vs JIT
  - Major user is BigInteger package
  - Called by crypto routines
  - Called by web services everywhere
- BigInt usage is as a 'pair of ints w/carry'
  - Lots of masking to high or low halves
  - Lots of shift-by-32
- Optimizes really well as a pair of ints
  - All those mask/shifts turn into simple register selection
  - Better code on 32-bit machines by far (X86!)
  - Almost tied with “good” code on 64-bit machines
    - (add+addc vs add8)

Example: calling native from JIT'd code

- Ideally: simple call
  - Only true for ***fixed application set***
- Reality: Too complex, needs own frame
- Example:

  native Object foo(double d);

```
        save 64             # register window push
        mov   i0,o0         # Move incoming arg
# o0 holds live OOP at call
        call  foo           # The Native Call
        nop                 # (really fill delay slot)
        mov   o0,i0         # Move outgoing arg
        return              #
        restore             # pop sparc window
```

Example: Native Call from JITed Code

- First problem: JIT and Native conventions
  - Eg: SparcV8 passes float in int regs
  - Java normally floats passed in float regs
  - No Java varargs or prototype-less code
- Want fast argument shuffle
  - Auto-generate asm shuffle code from sig

```
std  d0,[sp+64]   # float args passed in int regs
ldw  [sp+64],o1   # misaligned, need 2 loads
ldw  [sp+68],o2   # double now in o1/o2
```

Example: Native Call from JITed Code

- Cannot pass in oops to Native
  - Lest GC be unable to find them
  - (only for moving collector)
  - Handlize oops – part of arg shuffle code
  - Need an OopMap as well: [sp+72] live across call

```
        set  0,o0          # assume null arg
        beq  i0,skip       # handle of null is null
        stw  i0,[sp+72]    # handlize 'this'
        add  sp,72,o0      # ptr to 'this' passed in o0
skip:
```

- Reverse for returning an oop after the call

```
        beq   o0,is_null
        ldw   [o0],o0      # de-handlize return value
is_null:
```

Example: Native Call from JITed Code

- May need locking, i.e. synchronized keyword

```
        ldw  [i0+0],l1     # Load 'this' header word
        or   l1,2,l2       # set locked-bit
        stw  l2,[sp+76]    # save header on stack
        cas  [i0],l1,l2    # Attempt to fast-lock
        cmp  l1,l2         # success or fail
        bne  slow_lock     # CAS failed? Recursive lock?
# not shown: inline recursive-lock handling

# Both [sp+72] and i0 hold a live OOP across call
# [sp+76] holds 'displaced header' - needed for inflation
        call foo           # The Native Call
        nop                # (really: fill in delay slot)

        cas  [i0],l2,l1    # Attempt to fast-unlock
        cmp  l1,l2         #
        bne  slow_unlock   # (awake waiting threads)
```

Example: Native Call from JITed Code

- Next problem: Native code may block
  - We must allow a GC (if multi-threaded)
  - But GC needs SP/PC to crawl stack
  - Which portable OS + native compiler won't tell us
- So store SP/PC before calling native
  - Storing SP is also trigger that allows GC

```
        setlo #ret_pc,l0
        sethi #ret_pc,l0   # set PC before allow gc
        stw   l0,[g7+&pc]  # Rely on sparc TSO here (IA64!)
        stw   sp,[g7+&sp]  # Enable GC from here on
        call  foo          # The Native Call!!!
        nop                # (really fill delay slot)
ret_pc:
```

Example: Native Call from JITed Code

- Next: Native code returns while GC active
  - Must block until GC completes
  - No good to spin (eats CPU)
- Similar to 'lock acquire'
- Requires real Atomic operation

```
        setlo #ret_pc,l0
        sethi #ret_pc,l0   # set 32-bit PC before allow gc
        stw   l0,[g7+&pc]  # Rely on sparc TSO here (IA64!)
        stw   sp,[g7+&sp]  # Enable GC from here on
        call  foo          # The Native Call!!!
        nop                # (really fill delay slot)
ret_pc: add   g7+&sp,l0    # CAS does not take an offset
        cas   g0,sp,[l0]   # if sp still there, store 0 to disable gc
        bne   gc_in_progress
```

Example: calling native from JIT'd code

- Need JNIEnv* argument, really offset from thread

```
# JNIEnv* is 1st arg, shuffle others to o1,o2,etc...
        add   g7,&jnienv_offset,o0
```

- Reset temp handles before and after

```
        ldw   [g7+&gjni_sp],l3
        stw   g0,[l3+&top]
....
        call  foo
        nop           # delay slot
        stw   g0,[l3+&top]
```

- Profiling tags (optional)

Things that worked suprisingly well...

- Safepoint notion
  - Easy for Server compiler to track, optimize
  - Good optimization
- Polling for Safepoints
  - Software polling not so expensive
    - Free for Azul
  - Threads stop at 'convenient' spots
  - Threads do many self-service tasks
    - Self-stack is hot in local CPU cache
- Cooperative Preemption (Azul only)
  - most stopped threads already Safepointed

Things that worked suprisingly well...

- Heavy weight compiler
  - Needed for peak scores
  - Can be **really** heavyweight and still OK
  - Loop optimizations (unrolling, peeling, invariant motion)
    - Actually plenty cheap and payoff well
- C2's Graph IR
  - ***Very*** non-traditional
  - But very fast & light
  - And very easy to extend
- C2's Graph-Coloring Allocator
  - Robust in the face of over-inlining

Things that worked suprisingly well...

- Portable stack crawling code
  - Need SP & PC
  - Need notion of 'next' frame, 'no more frames'
    - Frame interator
  - Works for wide range of CPUs and OSs
- Non-portable bits:
  - Flush register windows
    - Must lazy flush & track flushing
  - Two kinds of stack on IA64, Azul
- Frame adapters for mixing JIT, interpreter
  - Custom asm bits for reorganzing call args
  - Really cheap, once you figure it out

Things that worked suprisingly well...

- CodeCache notion
  - All code in same 4Gig space
  - All calls use 'cheap' local call
  - Big savings on SparcV9 vs 'far' call
  - Only need a 32-bit PC everywhere
- BlahBlahBlah-ALot debugging flags
  - SafepointALot, CompileALot, GCAlot, etc...
  - Stress all sorts of interesting things
  - Easy for QA to run big apps long time w/flags
  - Catches zillions of bugs, usually quite easily

Things that worked suprisingly well...

- Thin-lock notion
  - HS's, Bacon-bits, whatever
  - Key: single CAS on object word to own lock
  - CAS on unlock hardly matters; it's all cache-hot now
- Actually, want a thinner-lock:
  - Thread speculative 'owns' lock until contention
  - No atomic ops, ever (until contention)
  - Pain-in-neck to 'steal' to another thread
  - BUT toooo many Java locks never ever contend
- Actually, JMM is working out nicely as well

Hard but useful things...

- Portable
  - Sparc (windows, RISC)
  - X86 (CISC, tiny register set)
  - Both endianess
- System more robust for handling all flavors
  - Requires better code discipline
  - Seperates out idea from implementation better
- No middle tier
  - HS is working, but has needed a middle tier for 3 yrs

Hard but useful things...

- Deoptimization
- No runtime cost to inline non-finals
- No runtime cost if you DO override code
  - Must recompile of course
- Must flip compiled frame into interpreted frame
- Not Rocket Science
  - But darned tricky to get right
  - Only HS does it
- Others pay a 'no-hoisting' cost at runtime

Hard but useful things...

- Code Patching
  - Inline-caches
  - Deoptimization (not-entrant code)
- Patch in the face of racing Java threads, of course
  - Must be legit for Java threads to see partial patches
- Almost easy on RISC
  - Still must do I-cache shoot-down
- Pain on X86
  - Variable-size (does it fit?)
  - Instructions span cache-lines
  - No atomic update

Hard but useful things...

- Hand ASM in HLL
  - Turns out need lots of hand ASM
- Want tight integration to runtime & VM invariants
  - External ASM doesn't provide this
- Fairly easy to make ASM 'look' like ASM
  - But actually valid HLL code (C, Java)
  - Which, when run, emits ASM to a buffer
  - And proves invariants, and provides other support

Things I won't do again...

- Write a VM in C/C++
  - Java plenty fast now
  - Mixing OOPS in native code a total pain
  - Forgetting 'this' is an OOP
    - Across a GC-allowable call
  - Roll-your-own malloc pointless now
- C2's BURS patterning-matching
  - Harkens back to VAX days
  - Never needed on RISC
  - Not needed on X86 for a long time now

Things I won't do again...

- Patch & roll-foward Safepoints
  - Hideously complex
  - Very heavy-weight to 'safepoint' a single thread
  - Multiple OS suspend/resumes
  - Patching non-trivial
  - Required duplicate code
    - And so required dup PC handling throughout VM
- Generic callee-save registers
  - X86 doesn't need them
  - Real mess to crawl stacks and track
  - Register windows work fine, both variable and fixed
  - Only PPC common+many regs+no windows

Things I won't do again...

- Adapter frames
  - For intercalling JIT'd code and interpreter
  - Contrast to 'frame adapters'
  - These leave a frame on stack
  - Extra frame screws up all kinds of stack crawls
  - Occasionally end up with unbounded extra frames
- No adapter frames means: interpreter & JITd code must agree on return value register
- Only an issue for SparcV8 long values

Things I won't do again...

- Constant oops in code
  - Looks good on X86 as 32-bit immediate
  - Split on Sparc, PPC, other RISC
  - Moving collector requires patching the constant
    - Multi-instruction patch?
  - Must patch with all threads stopped
  - Requires Stop-The-World pause in GC
- Better answer: pay to load from table every time
  - Easy to schedule around
  - Concurrent GC
  - Fewer instructions, especially for 64-bit VMs
  - No patching, no tracking of where oop is

Things I won't do again...

- Lock'd object header in stack
  - Means no tracking recursion count
  - Means total pain when inserting hashcode,
  - or inflating lock,
  - or moving during concurrent GC

Summary

- The need to run any code, including native
- Run it fast, well, defensively
- Handle extremes of threading, GC, code volume
- ***---- Be General Purpose ----***
- Forces difficult tradeoffs
- Things that are cheap & easy ....
  - *when you own the whole stack*
- *Are hard when you don't!*

Summary

- I “know too much” about this space
  - And yet – so much is unknown
  - And I'm in no way an expert in small JVM
- No good forum for the sum of it
  - Book maybe?
- Open questions for a Big JVM...
  - Graph-coloring vs linear-scan – I can argue strongly both ways
  - 'Right size' for an object header
  -

Thank You

Dr Click, Distinguished Engineer

Bits of Advice For the JVM Writer (part 0, the prequel)

THE ERA OF UNBOUND COMPUTE IS NOW

Dr Click, Distinguished Engineer

Inline Caches

- Wonderful – for fixing a horrible problem
- Horrible – for causing a wonderful? problem
- All major JVMs (Sun, IBM, BEA) do it
- “Makes Java as fast as C”
- Or at least, make virtual calls as cheap as static
  - *A **key** performance trick*

vs Virtual Calls

- What's wrong with v-calls?
- OFF by default in C++
- ON by default in Java
- So Java sees a zillion more of 'em than C++
- And so MUST be fast or “Java is slow”
- Typical (good) implementation:
  - LD / LD / JR
- Dependent loads (no OOO), so cost is:
  - 3 + 3 + 25 = 31clks
  - Actually 1st load probably misses in L1 cache 50%

A Tiny Fast Cache

- But – 95% of static v-calls are to Same Class 'this'
- Means jump to same target address
- Predict 'this' class with 1-entry cache
- On a hit, use static call – NOT jump-register
- Cost is:
  - LD / (build class immediate) / CMP / TRAP / CALL
- cmp predicts nicely, OOO covers rest:
  - Max(3,1 + 1) = 3
- Now 95% of v-calls nearly same cost as C calls!

But ....

- On prediction failure, patch code to Do It Right
  - (that LD / LD / JR sequence)
- Actually, start with cache empty & patch to fill
  - Target chosen lazily – important!
  - First call fails prediction
  - Patches to 1st caller's target
- Patching must be safe in presence of racing CPUs
  - Always a multi-instruction patch
  - CPUs can be pre-empted between any 2 instructions
  - Sleep during patching, then awaken and...
  - See any 'cut' of the patch sequence

Code patching

- Fairly easy to patch -
  - Empty to static (no prediction needed)
  - Empty to predicted (install key/value into cache)
  - Predicted to full-lookup
- Impossible to patch:
  - (predicted or full-lookup) to empty!
  - Must guarentee no thread is mid-call!
- Without a full stop-all-threads at safepoints
  - Or some other roll-forwards / roll-backwards scheme
  - Very expensive
  - So don't clean caches

Inline Caches

- Means inline caches hold on to busted predictions
  - Until we bother to stop and clean
  - Or a thread stumbles across it and fails prediction
- Means one piece of JIT'd code holds a pointer to another – indefinitely!
  - At any time a running CPU might load the address
  - Keep it in his PC register
  - Get context-switched out
  - Wake up sometime later....
- Means I cannot trivially remove dead JIT'd code!

NMethods

- HotSpot jargon for “JIT'd code”
- Complex datastructures
- Very heavy with multi-threaded issues
  - Hence full of subtle bugs
- More than one active per top-level Method
- Very complex lifetimes
  - Constantly produced (as new classes loaded) so...
  - Must be constantly culled as well

NMethods

- Generated code can be in active use
  - CPUs are actively fetching instructions
  - Might be OS hard-preempted at ANY point
  - Live code in I$
- Return PCs active in frames for long times
  - GC must find PCs to find NMethods to find OopMaps
- Code is patched
  - Inline-caches are VERY common, VERY hot
  - Direct calls from one NMethod to another
- Code contains OOPs
  - GC is actively moving objects

NMethods - Data

- The Code
  - And relocation info
- Class dependencies
  - e.g. invalidate this code when a subclass of X is loaded
- Inline caches
  - direct calls from one NMethod to another
- OopMaps at Safepoints
  - OOPs in code – roots for GC, maybe changed
- Other info
  - Deoptimization maps, exception handling tables, inlining info

NMethods - Lifetime

- Construction & Installation
- Publish
- Actively Used
- NotEntrant
- Deoptimized
- Zombie
- Flush

NMethods – Construction & Installation

- During construction
  - embedded OOPs must be kept alive
  - NMethod can not do it
  - Need “hand-off” with compiler
- Race during Install with invalidating class load
  - Compiler has made assumptions about Classes
  - If class gets loaded, must invalidate NMethod
- Maybe need frame adapters
  - To map call signature from interpreter to JIT'd layout

NMethods – Publish

- Make a strong ref from owning Method
- Other running threads can instantly find
  - And start executing
  - And instantly need GC info
- Must fence appropriately

NMethods – Active

- Running threads find via calls to owning Method
- Inline-caches directly reference
- PC register in CPUs directly reference
- Stack frame return PCs (hardware stacks!) ref
- GC info needed
- Deoptimization info needed
- Exception handling tables needed

NMethods – NotEntrant

- No new calls can be made
  - But old calls are legit, can complete
- Happens when compiler makes poor (not illegal) decisions
  - And wants to try again after profiling
- Patch entry point with a trap
  - Callers hit trap & fixup
  - Generally fixup calling code to not call again and
  - Find another way to execute
- All data remains alive

NMethods – Deoptimized

- No OLD calls either
  - Class-loading makes existing calls illegal to continue
- Wipe Code out with NOPs
- Existing nested calls run full speed till return
  - And fall thru NOPs into a handler
- No more OOPs in code, no exception handlers
  - No OopMaps, inline caches, dependency info
- But need deopt info (which may include OOPs)
- Inline caches, I$, CPU PCs still point here!

NMethods – Zombie

- No more pending Deoptimizations remain
  - Generally found by GC at next full Safepoint
- But Inline-caches still point here
  - So CPU PCs still point to 1st instruction
- Must sweep all existing NMethods
  - Could be slow, megabytes of JIT'd code
  - But done concurrently, parallel
- After sweep, must flush all I$'s
  - Could be slow, impacts all running CPUs

NMethods – Flush

- Remove Code from CodeCache
- Reclaim all other datastructures
- It's Dead Jim, Finally!