

Bits of Advice For the VM Writer

Cliff Click

H₂O.ai

Fast, Scalable In-Memory Machine and Deep Learning
For Smarter Applications

Who Am I?



Cliff Click

CTO, Co-Founder H2O.ai

cliffc@h2o.ai

PhD Computer Science

1995 Rice University

HotSpot JVM Server Compiler

“showed the world JITing is possible”

40 yrs coding

35 yrs building compilers

30 yrs distributed computation

20 yrs OS, device drivers, HPC, HotSpot

10 yrs Low-latency GC, custom java hardware,

NonBlockingHashMap

20 patents, dozens of papers

100s of public talks

Some History of HotSpot

- HotSpot started as a **Self** VM ported to Java
 - Simple GC, simple JIT, single-threaded, X86 only
- I've been working on HotSpot for ~20 yrs now
- Watched HotSpot become 'thread-robust'
- Many ports: X86, Sparc, IA64, X86-64, MIPS, Azul
- Pick up 2 new compilers (C1 & C2, -client & -server)
- Pick up many new GC's (serial, train, parallel, CMS, G1, Azul's Pauseless)
- Other Cool Stuff: reflection as bytecodes, thin locks, JMM, JNI, JVM{DI, TI, PI},

Nature of This Talk

- Much hard experience learned building HotSpot
 - Too much info for one talk!
- Many topics are crucial for
 - Ultimate speed, power, or footprint
 - Engineering complexity & cost
- Interlock badly, in non-obvious ways
- I'm gonna breeze through! (limits of time)
 - Happy to stop for questions
 - ASK: because if your confused,
so is the next person



Agenda

- **Some Choices To Make**
- Native Calls – a Deep Dive
- Things that Worked Well
- Hard Things but Worth Doing
- Things I Won't Do Again
- Q&A

(if time, A Deep Dive into Code Unloading)

VMs are Big Complex Beasties

- Or Really Really Small (e.g., Squawk, OOVm)
- 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, Endianess
 - Threads, POSIX, stacks
- Footprint
 - Embedded (1G), desktop (32G+), server (16G-256G)
- X86 vs RISC (ARM? GPU? DSP?)
- JIT or Interpret (or no Interpreter!)
- Multi-threaded
 - Cooperative vs preemption
- Multi-CPU



Interpreter Choices...

- 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, pico-Java?)
- Fancy: inline & schedule dispatch
 - Seen at least 2 different ways to slice this
- Requires stack oriented layout, bad for JITs
- or none at all...



Some JIT choices to make...

- JIT
 - None? (gives up peak performance)
 - stage0, 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-final methods?



More JIT choices to make...

- Stage-0 JIT + No Interpreter
- 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 generation, 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? (really⁴ hard!)



More GC choices...

- 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?)
 - Polling in software vs hardware?
 - Preempt at wrong place?
 - Roll forward vs step forward vs interpret?



Threading Issues

- Multi-threading costs
 - All operations no longer atomic
 - May be preempted inconveniently
 - Need locking (never needed it before!)
- Threads can block for I/O, or deadlock
- GC requires SP (& usually PC) for stack roots
- Hard to find the SP+PC of remote thread
 - Common problem of OS's
 - ! **Very** surprising 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



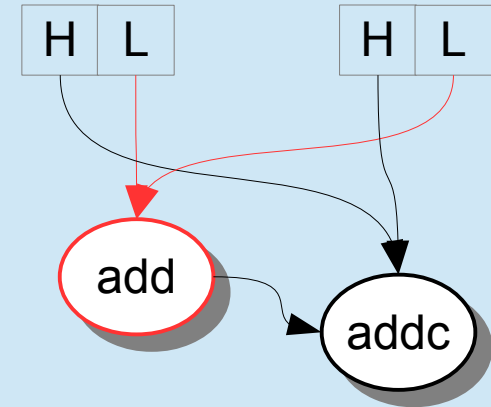
Multiple CPU Issues

- Multi-CPU?
 - Now need Atomic operations
 - Coherency, memory fences
 - Low-frequency data race bugs
- Need *scale-able* locks in JVM
 - Not just correctness locks
- Need *scale-able* 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



64-bit Math Choices...

- Long (64bit int) math vs JIT
 - Major user is BigInteger package
 - Called by crypto routines
 - Called by web services everywhere
- BigInteger 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” 64b code on 64-bit machines



Agenda

- Some Choices To Make
- **Native Calls – a Deep Dive**
- Things that Worked Well
- Hard Things but Worth Doing
- Things I Won't Do Again
- Q&A

(if time, A Deep Dive into Code Unloading)

Native Calls: A Deep Dive

- Ideally: simple call
 - Only true for *fixed application set*
 - Often holds true for cell phones, small embedded
- Reality: Too complex, wants 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
```

The Argument Shuffle

- First problem: JIT and Native calling 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
# 'this' pointer is in o0
ldw  [sp+64],o1    # misaligned, need 2 loads
ldw  [sp+68],o2    # double now in o1/o2
```

Handlizing OOP Arguments

- 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 OOP-map 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:

Synchronized Native

- 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)
```

Allowing Stack Crawls

- 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,10
sethi #ret_pc,10 # set 32-bit PC before allow GC
stw   10,[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)
```

Returning while GC-in-Progress

- 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,10
sethi #ret_pc,10 # set 32-bit PC before allow GC
stw   10,[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,10 # CAS does not take an offset
      cas   g0,sp,[10] # if SP still there, store 0 to disable gc
      bne  gc_in_progress
```

Odds-N-Ends

- 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],13
stw    g0,[13+&top]
.....
call   foo
nop                    # delay slot
stw    g0,[13+&top]
```

- Profiling tags (optional)

A Sample Native Call

```
save 64          # register window push
add  g7,&jnienv_offset,o0 # JNIEnv* in o0
set  0,o1        # assume null arg
beq  i0,skip    # handle of null is null
stw  i0,[sp+72] # handlize 'this'
add  sp,72,o1   # ptr to 'this' passed in o1

skip:
std  d0,[sp+64] # float args passed in int regs
ldd  [sp+64],o2 # double now in o2/o3
setlo #ret_pc,10
sethi #ret_pc,10 # set 32-bit PC before allow GC
stw  10,[g7+&pc] # Rely on Sparc TSO here (IA64!)
stw  sp,[g7+&sp] # Enable GC from here on
call foo        # The Native Call

ret_pc: add  g7+&sp,10 # CAS does not take an offset
cas  0,sp,[10] # if SP still there, store 0 to disable gc
bne  gc_in_progress
set  0,i0      # assume null result
beq  o0,is_null
ldw  [o0],i0   # de-handlize return value

is_null:
return #
restore # pop sparc window
```

} Handles,
GC-unsafe natives

} Arg-shuffle
JIT vs native convention

} Stack crawl
GC lock

The Call

} GC unlock

} De-Handlize

Agenda

- Some Choices To Make
- Native Calls – a Deep Dive
- **Things that Worked Well**
- Hard Things but Worth Doing
- Things I Won't Do Again
- Q&A

(if time, A Deep Dive into Code Unloading)

Safepoints, Preemption & Polling

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

Heavy weight JIT Compiler

- Heavy weight compiler
 - Needed for peak performance
 - 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



Portable Stack Manipulation

- Portable stack crawling code
 - Need SP & PC
 - Need notion of 'next' frame, 'no more frames'
 - Frame iterator
 - Works for wide range of CPUs and OSs
- Less-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 reorganizing call args
 - Really cheap, once you figure it out

The Code Cache, Debugging

- CodeCache notion
 - All code in same 4Gig space
 - Only need a 32-bit PC everywhere
 - All calls use 'cheap' local call
 - Big savings on X86 (& Sparc, RISC) vs 'far' call
- 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

Thin Locks

- Thin-lock notion
 - HotSpot'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
- JMM worked out nicely as well

Agenda

- Some Choices To Make
- Native Calls – a Deep Dive
- Things that Worked Well
- **Hard Things but Worth Doing**
- Things I Won't Do Again
- Q&A

(if time, A Deep Dive into Code Unloading)

Porting to Many CPUs+OSs

- Portable
 - Sparc (windows, RISC)
 - X86 (CISC, tiny register set)
 - Both endianness
- System more robust for handling all flavors
 - Requires better code discipline
 - Separates out idea from implementation better
- No middle compilation tier
 - HS has needed a middle tier for years

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

Self-Modifying Code

- Code Patching
 - Inline-caches
 - “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

HLL Assembler

- 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

64b Object Header

- Single word on 64-bit VM
- Large memory savings plus speed in caches
- Needs dense KlassID vs 64b KlassPtr
- Thread ID for locking, speculative locking
- HashCode – want all 32bits when hashing more than a few billion objects

Dense Thread ID

- Align stacks on 2Mb boundary
 - Stack overflow/underflow: TLB 4K page protection
 - Protect whole stack for various GC asserts
- Mask SP to get Thread ptr; shift for Thread ID
 - Plus Thread Local Storage for VM
- TID in Object headers for locking
- Thread ptr **very** common in core VM code
 - Must be *fast*

Safepointing Single Threads

- Software polling of single TLS word
 - 1-clock on X86, predicted branch, L1 cache hit load
- Set a bit to stop a thread at safepoint
- Thread does many self-service tasks
 - Crawl self stack for GC roots, GC phase flip
 - Install remote exception, or stack overflow
 - Revoke biased lock
 - Debugger hooks; stop/start/conditional breakpoints
 - Clean inline caches
- Cooperative preemption at safepoint

Agenda

- Some Choices To Make
- Native Calls – a Deep Dive
- Things that Worked Well
- Hard Things but Worth Doing
- **Things I Won't Do Again**
- Q&A

(if time, A Deep Dive into Code Unloading)

Things I won't do again...

- Write a VM in C/C++
 - Java plenty fast now
 - Mixing OOPS in a non-GC language 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
 - Adds an extra indirection in the JIT engineering

Things I won't do again...

- Patch & roll-forward 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
 - Real mess to crawl stacks and track
 - X86 doesn't need them
 - Register windows work fine, both variable and fixed
 - Only PPC & ARM 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 & JIT'd 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 instructions 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 due to contention,
 - or moving during concurrent GC

Agenda

- Some Choices To Make
- Native Calls – a Deep Dive
- Things that Worked Well
- Hard Things but Worth Doing
- Things I Won't Do Again
- **Q&A**

(if time, A Deep Dive into Code Unloading)

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!*

Open Questions for a Big JVM

- Graph-coloring vs linear-scan allocator?
 - I can argue strongly both ways
- 'Right size' for an object header?
 - Azul HotSpot: 1 (64bit) word, not two
- Interpreter vs stage0 JIT?
 - Tracing stage1 JIT vs Heavy-weight vs multi-tier
- OS Threads vs Roll-Your-Own (e.g. Green)?
- High-Throughput vs Low-Pause GC?

Bits of Advice - Deep Secrets

Cliff Click

H₂O.ai

Fast, Scalable In-Memory Machine and Deep Learning
For Smarter Applications

Inline Caches

- Wonderful – for fixing a horrible problem
 - Virtual calls: both **very** common and SLOW
- 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:
 - $3 + 3 + 25 = 31\text{clks}$
 - 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:
 - $\text{Max}(3, 1 + 1) = 3$
- 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 guarantee 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 data structures
- Very heavy with multi-threaded issues
 - Hence full of subtle bugs
- More than one active per top-level Java 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 OOP-Maps
- Code is patched
 - Inline-caches are VERY common, VERY hot
 - Direct calls from one NMethod to another
- Code contains OOPs

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
- OOP-Maps at Safepoints
 - OOPs in code – roots for GC, maybe changed
- Other info
 - Deopt maps, exception handling tables, inlining info

NMethods - Lifetime

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

NMethods – Make & 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 OOP-Maps, 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!