ORACLE

Flat objects — in memory — safely accessed

questions, sample solutions

John Rose, JVM Architect

March 24, 2022

POST-MEETING VERSION: Includes slides omitted from meeting or added as follow-up.

Flat objects — in memory — safely accessed

⇒ Vector atomics?

OpenJDK Project Valhalla, a very very short summary

- Java objects today are accessed indirectly, via pointer-to-header
  - The “payload” of an object lives in a different D$ block than the reference
  - Reference word sits in a “container” — payload is displaced elsewhere.
  - Typical costs of pointer-chasing, from reference in D$[i] to payload in D$[k]
- With Valhalla, the JVM “flattens” the object, like an inlined C/C++ struct
  - The whole object sits in its “container” (array/field)
  - Good density too: Both the reference word and header word(s) disappear
  - Goal: cut out up to half of the D$ traffic, which was due to pointer-chasing.
  - Our early testing supports this as a realistic goal! (Faster sorting…)
- **Problem: Flattened data is subject to “struct tearing”.**

Problem: Flattened data is subject to “struct tearing”

- Struct tearing is what happens when 2 words inside one object “diverge”
  - (By “diverge” I mean 2 visible writes are “mixed up” from 2 threads.)
  - (By “mixed up” I mean that class invariants require 2 writes be from 1 thread.)
  - “Just a SW problem?” (Mixing writes from 2 threads is a HW expense too!)
- With Java/JVM, an object is entitled to define and protect multi-word invariants
  - (Simple example invariant: Two fields are never both zero at the same time.)
- A non-flattened object can “synchronize” (header mutex); a flattened one cannot.
- A non-flattened object can make itself immutable, so **one write per field**.
  - Valhalla objects are logically immutable, so problem solved?
- **Problem.next: All fields of a flattened object must rewritten simultaneously.**

Problem: All fields of a flattened object must rewritten simultaneously.

- The object is logically immutable, but *the container can update the whole object*.
  - (This was trivial with non-flat pointer+header objects: Just update the pointer.)
  - Updating the whole object, in a mutable container field, must be atomic.
- “Atomic” is the hard part here. Flat-object update is a transaction! (Ouch.)
  - Software transactional memory is our fallback, our slow path.
  - But what is our fast path?? That’s where we need guidance.
- Note: Guidance at this point is probably the same as for C++ multi-word atomics!
  - (What you advise us today will be helpful in the C++ ecosystem also.)

- **Problem.last: We need a current or future hardware fast path for R/W.**

More on Valhalla trade-offs [followup slide from verbal discussion]

- Assumption: Almost all Valhalla flattened objects will fit in a D$ line.
- “Sweet spot” = inline flattened objects of ≈ 1/2/3/4/?5 words. (Mask odd sizes.)
  - (Per usual observations of typical object sizes as coded by programmers.)
  - It is OK to optimize smaller ones (1/2) better than larger ones (3/4/5).
  - Desired fully-optimized size ≈ 192-256 bits (3-4 words); 512 must mask.
- Note: Power-of-two sizing works against the Valhalla benefits of flattening
  - Adding a wasted 64-bit word to round up a 192-bit structure dilutes memory.
- Very large flattened objects (equal to or larger than a D$ line) will be rare.
  - They do not require full optimization, and/or will use different techniques.
  - A large object can “afford” an embedded monitor and/or an indirection.

SLIDE MISSING/OMITTED FROM MEETING

Valhalla trade-off Implications [followup slide from verbal discussion]

- 128-bit atomics are great; they handle the 2-word case.
  - 3/4/5 word objects will require much slower handling (buffer/version/lock)
- Please consider 3-word objects, masked in 256-bit container
  - not aligned to 256-bit address boundary
  - really masked: no accidental “tearing write” to unused memory
  - byte-wise masking not needed here
  - D$ line crosses not needed here (would be nice, but can avoid)
- Similar “ask” for 5-word objects, but somewhat less important. Also 6/7/8.
- AVX-2 4-word 256-bit objects are part of 3/4/5 word size sequence.
  - not aligned; implicitly masked to avoid “tearing write” to unused memory

SLIDE MISSING/OMITTED FROM MEETING

Hardware fast path for atomic update of flat objects: Candidates that fail

- MOVDQ[A/U]: Normal vector-wise, masked as necessary: Not guaranteed
  - Testing suggests that 99.9+% of vector R/Ws are atomic; 0.1-% = problem.
- TSX: Too much unpredictable power: Solves bigger “many-location” problems.
  - Flattened object update touches 1 D$ line (maybe two, if bad alignment…)
- HLE: Would require an extra “mutex word” somewhere near the container
  - (Maybe the containing object’s header, but that may not be on same D$ line)
- LOCK CMPXCHG16B: Works on aligned 64-bit word pairs. Improvable?
  - Could possibly help with 2-word value types. Tests show it’s very slow.
- MOVDIR[64B/I]: Direct cache-bypassing store for 512 bits (for journalling?).
  - Slow; requires fence; skips D$. Not for Java object computing

So, how to update a Java flat object? (And a multi-word C++ atomic?)

- SW problem only? (Just write portable code and forget HW tricks…)
  - Use a mutex or SeqLock: Maybe tune up with HLE?
  - Use object versioning: Pointer swapping; GC cleans the old versions later.
  - These options are expensive relative to primitive scalar (int64) read/write.
  - Valhalla aspires to make flat objects perform close to primitive scalars.
- Add guarantees to MOVDQ[A/U]? — No, disruptive and always-expensive
- Tune TSX for Java flat objects (and C++ atomics)? — Still needs STM fallback.
- Faster LOCK CMPXCHG16B? — Only reaches 25% of D$ line; want %zmm.
- MOVDIR[64B/I] variant? — Needs a cached, masked version for true density.
- **Or some new SW/HW combination?**

Naive sample proposal: LOCK MOVDQA vector move instructions.

- Locked vector load, unmasked
  - Allocate LSU queue resources, lock one D$ line (or two if unaligned?)
  - Transfer indicated VPU lanes into value tracking registers; release D$ line
- Locked vector load, masked — similar to unmasked
  - Might lock only one cache line where unmasked would lock two
- Locked vector store, unmasked
  - RTO (read-to-own) D$ line(s), wait for all VPU lanes ready
  - Allocate LSU queue resources, lock one D$ lines (or two if unaligned?)
  - Drain LSU queue entries under lock; release D$ line
- Locked vector store, masked — similar to unmasked
  - Might lock only one cache line depending on mask?

Naive sample proposal #2: CMPXCHGDQA vector CAS instruction.

- Inputs: Two (xyzmm) vectors A, B, one mask K, one address M
  - Output: A vector C (ignored for the write use case)
- Operation (omitting narrative about LSU and D$):
  - Load C=(M){K}, compare to C and A under mask K
  - Store (M){K}=B under mask, but only if C{K}=A{K}

- Possible restriction: test mask K is limited in size/shape
- Memory transaction is always one D$ line (or 2 if DQU not DQA?)
- Not a general TSX-like DCAS/CAS2; more like a “whole cache line” CAS
- Scales CMPXCHG16B to a whole D$ line; *might be slow as well.*

Naive sample proposal #2b: CMPXCHGDQA2M vector CAS instruction.

- Inputs: Two (xyzmm) vectors A, B, *two masks* K, K2, one address M
  - Output: A vector C (ignored for the write use case)
- Operation (omitting narrative about LSU and D$):
  - Load C=(M){K}, compare to C and A under mask K
  - Store (M){K2}=B under mask, but only if C{K}=A{K}
- Same possible restrictions or extensions as #2a.
- This #2b version is not necessary for Valhalla or C++ atomics
  - But it looks useful as a STM building block, e.g., queue management
  - User-space multiprocessor queues are hard and useful at the same time!
  - Could be the basis for an enhanced SeqLock mechanism (HLE for SeqLock)

The underlying realities of memory operations (an educated guess)

- Memory travels in D$ blocks (with update masks)
- The CPU decomposes D$ operations into smaller units, tracking data as scalars (words, bytes, …)
  - This happens in the LSU (load-store unit).
- Existing locking operations briefly pin D$ blocks and suppress decomposition
  - Sometimes they pin *adjacent* blocks when a locked operation crosses a D$ line

- Presumably there are difficulties with suppressing decomposition of vectors
  - Decomposing memory operations supports value numbering and reordering
  - Suppressing decomposition breaks those optimizations
  - Decomposing operations also allows more flexible fit to internal queue limits
  - Suppressing decomposition requires more resource allocation for internal queues

- Still, it is possible, *in principle,* to consider cache-locked versions of vector operations

SLIDE MISSING/OMITTED FROM MEETING

While we are here, some other queries: Sort/unsort cross-lane ops

- Need fuller conversation later about the design space vector permutations
  - **compress** : **expand** :: sort : **permute**(="unsort") :: summarize : parse
  - I think there is processing potential here to be unlocked
  - …By reasonable (incremental) modifications of existing HW function
- Key thoughts:
  - Compress is (1/2 of) radix-sort on 1-bit key
  - Expand is the inverse of compress
  - Permute is *inverse* of radix-sort on n-bit keys (n = lg(#lanes))
  - Radix sort on lane numbers is multi-bucket compress
  - Routing hardware for all of the above is in a common family

more on cross-lane motion [followup slide from verbal discussion]

- A butterfly network with suitable routing logic can sort/unsort lanes.
  - There are many ways to use this, if the “hooks” are right.
  - (I learned this in the ‘80s working on the Connection Machine.)
- When sorting, key-collisions need to be handled usefully.
  - Many possible collision actions: Pick-first, keep-all, add-values, etc.
  - The primitive is probably “keep-all” in stable order.
  - But that requires a segmented reduction to do “add-values”, etc.
- Segmented reduction or scan is a fundamental primitive. (The C.M. again.)
  - Use a mask register to define segments, then add/xor/mul/… in each.
  - “Scan” means accumulate partials along each segment.
  - Compress can collect the final value from each segment if desired.
  - For numbering applications, you want *all* the partials (scan values).

SLIDE MISSING/OMITTED FROM MEETING

While we are here, some other queries: HW support for advanced GC

- HotSpot Java-JVM Garbage Collection engineers ask…
  - What is the best way to work with “colored” pointers?
  - (Colored pointer is a machine address with a few extra bits for SW.)
  - Key operations: 1. color, 2. un-color, 3. test and branch on (unexpected) color
- LAM (linear address masking) is a long-term goal (handles 2. un-color)
- One current or potential technique is to use shifting for 1./2. and bit-test for 3.
- Today’s ask: Can this be made a fused op please?
  - SHRQ %RAW_PTR, #1; JA SLOW_PATH
  - Note that JA tests the carry bit that was produced as a flag by SHRQ (ugh!)
  - This handles de-color (ptr>>1) and test-color (ptr&1) in one frequent idiom

mixing LAM into JVM GC [followup slide from verbal discussion]

- JVM needs fast instruction idiom for test-color + remove-color
  - Best case: A fused test-branch (single micro-op)
- Without LAM, fast SHRQ %RAW_PTR, #1; JA SLOW_PATH
  - Delivers both a condition code and an un-colored PTR value.
- With LAM, fast BT %PTR, #61; JC SLOW_PATH (or JNC or #62…)
  - This is because with LAM the un-color operation is implicit
- Full disclosure: It's better for the JVM if long offsets are supported
  - If only short offsets, then it's a *frequent* jump-around-long-jump
- Alternative in some ISAs might be a trap-on-condition w/ fast-fast traps.
  - That would avoid the whole question of branch offsets.
  - But the SLOW_PATH *is not rare enough* to use regular traps.

SLIDE MISSING/OMITTED FROM MEETING

non-crypto compute with AES [followup slide from verbal discussion]

- Many algorithms (both JVM and user) feature non-crypto hashes
  - Requirements 1,2,3: Low latency, high throughput, very good dispersion.
- Requirement 4: Parametric (“saltable”) family of hash functions.
  - Random parameter (64b-128b) is used to prevent hash-prediction attacks.
  - Salting is also used to search for and instantiate perfect hash functions.
  - Zero-cost if salt is just a multiplier; can also be “whitening” pattern to XOR
- Portable solution is 64-bit [I]MUL, followed by 1-2 SHR+XOR.
  - MUL is uneven; SHR+XOR required to cascade MSB effects to LSB positions
- Better portable solution is 1-2 AES steps, “salted” by XOR patterns.
  - Faster and more even mixing across 128 bits than 64-bit MUL+XOR+XOR

SLIDE MISSING/OMITTED FROM MEETING

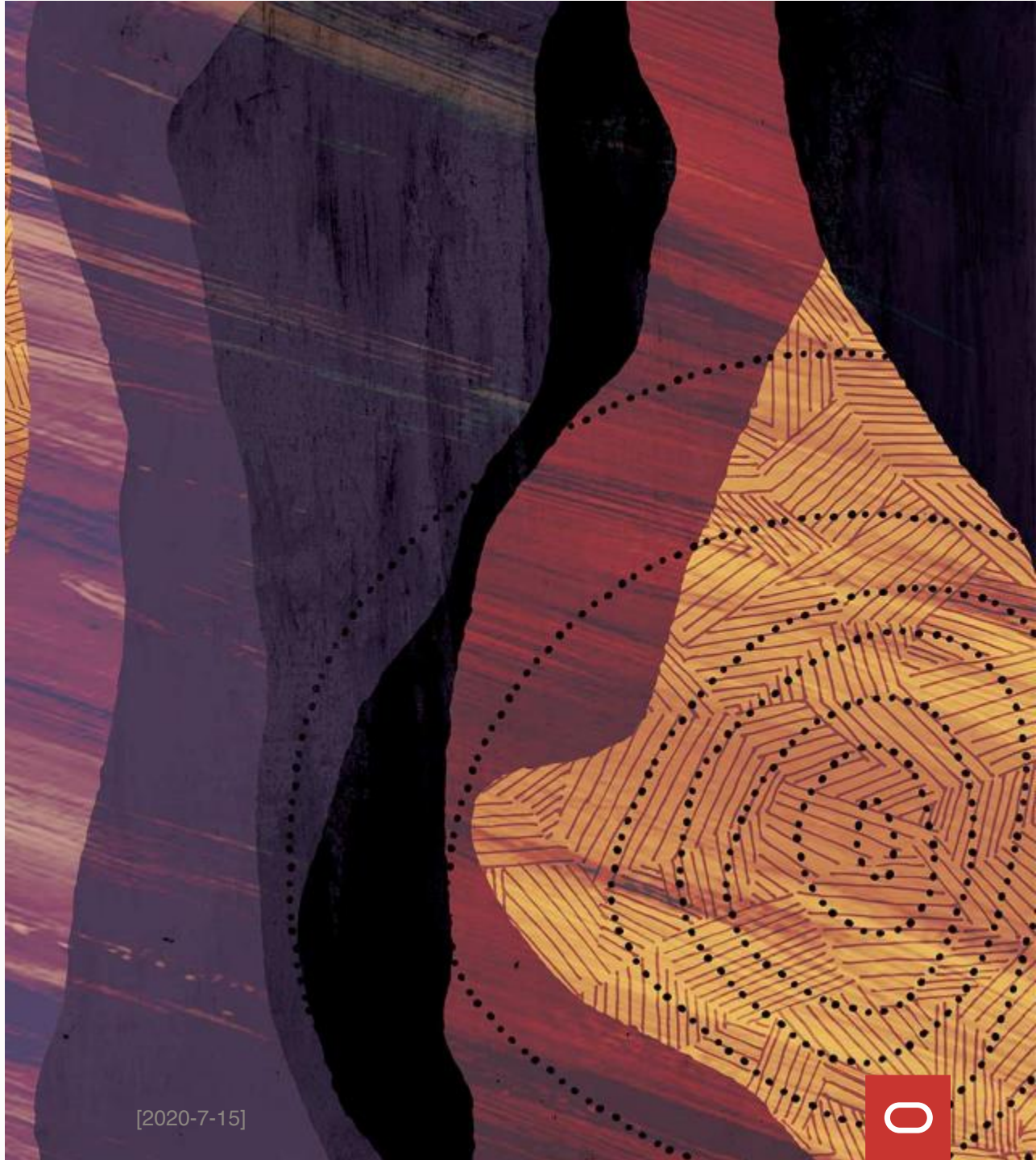
non-crypto compute with AES #2 [followup slide from verbal discussion]

- AES is non-linear (has S-boxes) compared with MUL or GF2P8AFFIN
  - (AES = either AESENC/DEC… AESKEYGENASSIST not so good)
  - Easy to salt, since ASEENC takes a second input to XOR (the “salt”)
- AES cascade (bit-per-bit, out-from-in) is 1-in-32-out in a regular pattern
  - This means a second AES step is needed for a full 128b cascade
  - By comparison MUL is 1-in-32.5-out (avg.) in a skewed pattern
  - GF2P8AFFIN is 1-in-8 out in a regular pattern (localized to bytes)
- To hash 128 bits of source material, 2 x AES is faster/better than MUL, GF2.
  - Suggestion: Maybe fuse back-to-back AESENC pairs?
  - VAES is useful to hash array input: Independently hash each chunk.

SLIDE MISSING/OMITTED FROM MEETING

Thank You

Questions? Comments?



Our mission is to help people
see data in new ways, discover insights,
unlock endless possibilities.