Pointer-Chase Experiments Summary

One Slide per Experiment: Purpose, Results, and Analysis

February 11, 2026

Goals

- Measure pointer-chase performance across Java HotSpot and C baselines.
- Compare intrinsic vs explicit-null-check (guarded) vs contract-style chase.
- Study working-set size effects and implicit-null-check behavior.
- Evaluate JIT bundling and MLP (multiple independent chains).
- Assess feasibility of offload ideas like Intel DSA for pointer chasing.
- Answer the core question: what actually limits pointer-chasing performance on modern CPUs and JVMs?

Big Picture: What Could Be Limiting Us?

- Pointer chasing is a serial dependency chain: each load depends on the previous address.
- Potential bottlenecks: branches/null checks, control-flow structure, JIT optimizations, object layout/GC, offload engines, or raw memory latency.
- The experiments are ordered to eliminate each suspect until only the true limiter remains.

Narrative Links Between Experiments

- After Exp 1: with a stable baseline, we can trust later deltas.
- After Exp 2: control-flow differences are tiny once memory dominates.
- After Exp 3: branch counts change a lot, but runtime barely moves.
- After Exp 4: native C confirms in-cache wins; DRAM hides them.
- After Exp 5/6: JIT bundling cannot break true dependence chains.
- After Exp 7: MLP is the reliable performance lever.
- After Exp 8: offload engines do not remove dependency latency.

Pointer-Chase Modes: What Do We Mean?

- **Intrinsic:** call into the HotSpot-provided `PtrChase.chase8` intrinsic; the JVM expands it into an 8-hop chase with implicit null checks.
- **Guarded:** explicit null checks after each hop; every step is a branch that can be predicted or mispredicted.
- **Contract:** unrolled dereferences with no explicit checks; a `try/catch` collapses control flow by relying on a null dereference to terminate.

Code: Intrinsic Variant (HotSpot-Provided)

```
static PtrChase.Node runIntrinsic(PtrChase.Node p, int iters) {
  PtrChase.Node x = p;
  for (int i = 0; i < iters; i++) {
    x = PtrChase.chase8(x);  // HotSpot intrinsic (unrolled chase)
  }
  return x;
}
```

Intrinsic delegates the chase to the JVM; it typically relies on implicit null checks rather than explicit branches.

Code: Guarded Variant (Explicit Checks)

```
static PtrChase.Node chase8_guarded(PtrChase.Node p) {
  if (p == null) return null;
  p = p.next; if (p == null) return null;
  p = p.next; if (p == null) return null;
  p = p.next; if (p == null) return null;
  p = p.next; if (p == null) return null;
  p = p.next; if (p == null) return null;
  p = p.next; if (p == null) return null;
  p = p.next; if (p == null) return null;
  p = p.next;
  return p;
}
```

Guarded is the most explicit form: every hop includes a null check branch.

Code: Contract Variant (Collapsed Control-Flow)

```
static PtrChase.Node chase8_contract(PtrChase.Node p) {
  try {
    p = p.next; p = p.next; p = p.next; p = p.next;
    p = p.next; p = p.next; p = p.next; p = p.next;
    return p;
  } catch (NullPointerException e) {
    return null;
  }
}
```

Contract removes explicit checks and relies on the exception path to terminate; the goal is fewer branches/guards.

Experiment Setup

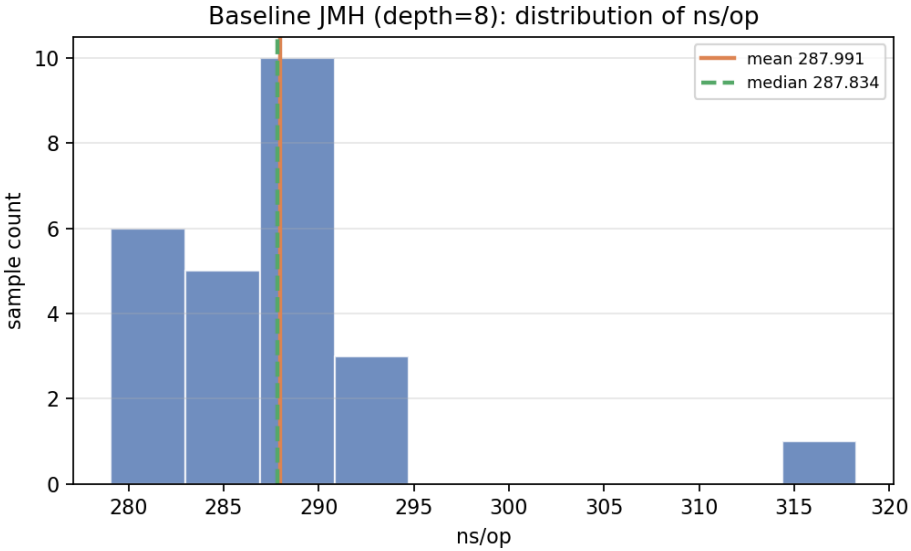
Date	2026-02-11 03:59 UTC
OS	Ubuntu 24.04.1, Linux 6.14.0-37-generic
CPU	Intel Core i9-9980XE (18C/36T, 3.0 GHz)
Cache	L3 24.8 MiB, L2 18 MiB, L1d 576 KiB
Memory	62 GiB RAM, 1 NUMA node
Baseline JDK	21.0.11-internal (JMH run)
Sweep/Nullcheck JDK	Built from openjdk-src/jdk-21.0.9+10 (per run scripts)

Experiment 1: Baseline JMH (Purpose, Results, Analysis)

- **Purpose:** establish a clean Java baseline (depth=8) so later changes can be judged against a trusted reference instead of noise.
- **Method:** JMH average-time mode with multiple forks and warmup to stabilize JIT and GC; ns/op is the average time to complete one pointer-chase operation.
- **Results:** 287.991 ns/op $\pm$ 5.601 (CI); min/max 279.069/318.285 ns/op, showing a tight distribution.
- **What it isolates:** JVM steady-state pointer-chase cost and benchmark variance.
- **Analysis:** this anchors the rest of the study; if we move only a few ns/op later, it is likely measurement noise, not a real effect.

Parameters	depth=8, n=1,000,000
Mode	Average time (ns/op)
Warmup/Measure	3x1s warmup, 5x1s measure, 5 forks

Experiment 1: Baseline JMH (Graph)



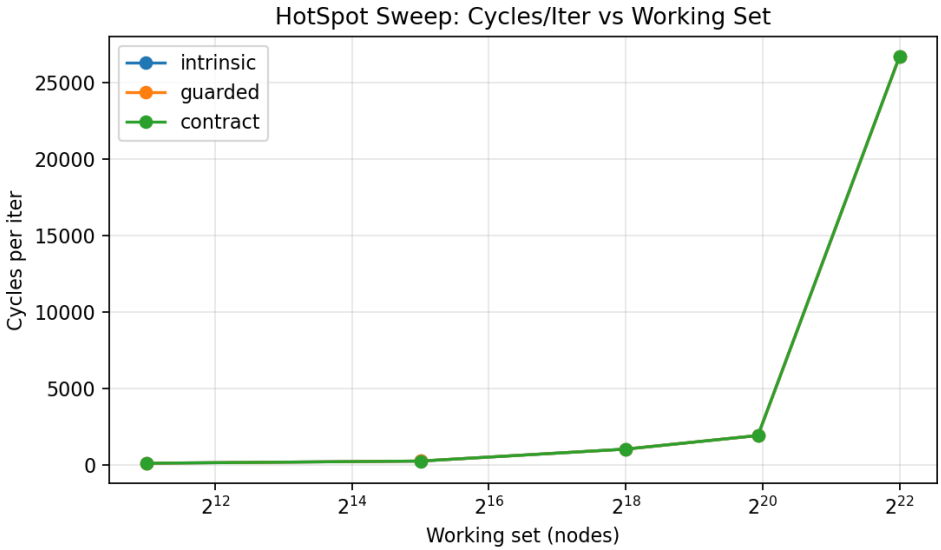
Experiment 1: Graph Analysis

- **Variables:** x-axis is ns/op; y-axis is sample count; the histogram shows the distribution of all fork/iteration samples (n=25).
- **Results:** mean 287.991 ns/op, median 287.834 ns/op; min/max 279.069/318.285 ns/op.
- **Behavior:** a single tight cluster indicates stable performance and low run-to-run noise.

Experiment 2: HotSpot Working-Set Sweep

- **Purpose:** isolate the effect of control-flow style while sweeping working-set size, so we can see when cache vs DRAM latency dominates.
- **Method:** pinned core, perf stat -r10 to collect cycle counts; cycles/iter is total cycles divided by iterations.
- **Results:** cycles/iter for intrinsic/guarded/contract stay within $\approx 0.5\%$ of each other, while the absolute cost explodes from ≈ 88 to ≈ 26.7k as the working set grows past cache.
- **What it isolates:** the memory-hierarchy effect (L1/L2/L3/DRAM) and whether control-flow changes remain visible under cache misses.
- **Analysis:** this already points at the core answer: once you miss cache, memory latency dwarfs branch or control-flow costs.

Experiment 2: HotSpot Working-Set Sweep (Graph)



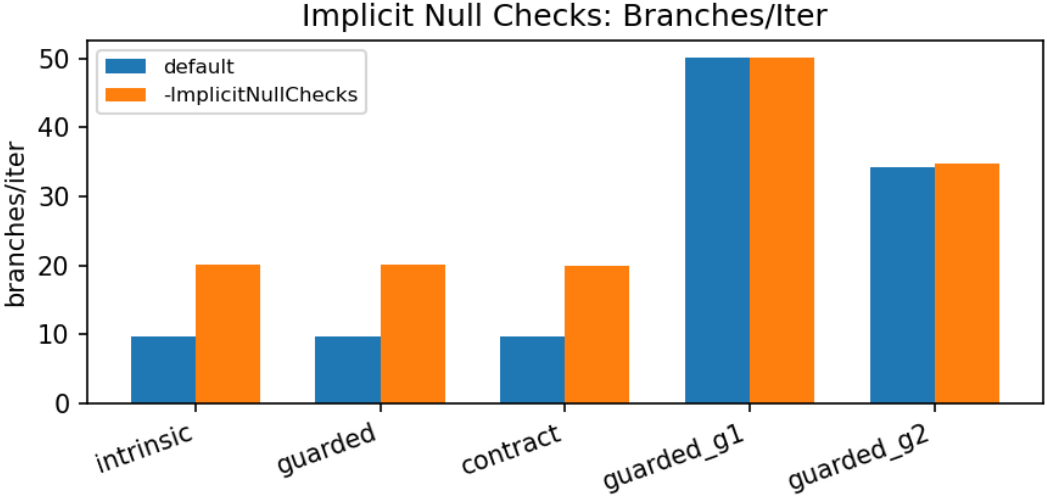
Experiment 2: Graph Analysis

- **Variables:** x-axis is working set size (nodes, log scale); y-axis is cycles per iter (perf cycles / iters), which directly reflects CPU time per chase.
- **Results:** cycles/iter rises from $\approx$ 88 at the smallest set to $\approx$ 26.7k at the largest; the three curves are nearly on top of each other.
- **Behavior:** once the working set spills out of cache, the chase is latency-bound and control-flow style stops being a meaningful lever.

Experiment 3: Implicit Null Checks

- **Purpose:** determine whether HotSpot's implicit null checks (using faults instead of branches) secretly help performance.
- **Method:** compare default vs `-ImplicitNullChecks` across two node sizes; summarize wall time and branches/iter from perf.
- **Results:** branches/iter roughly double for intrinsic/guarded/contract (about 2.08x), but total time changes are near 0%.
- **What it isolates:** branch-heavy null checking versus implicit null checking, and whether branch pressure affects end-to-end time.
- **Analysis:** another suspect eliminated: even large changes in branch count do not matter when memory stalls dominate.

Experiment 3: Implicit Null Checks (Graph)



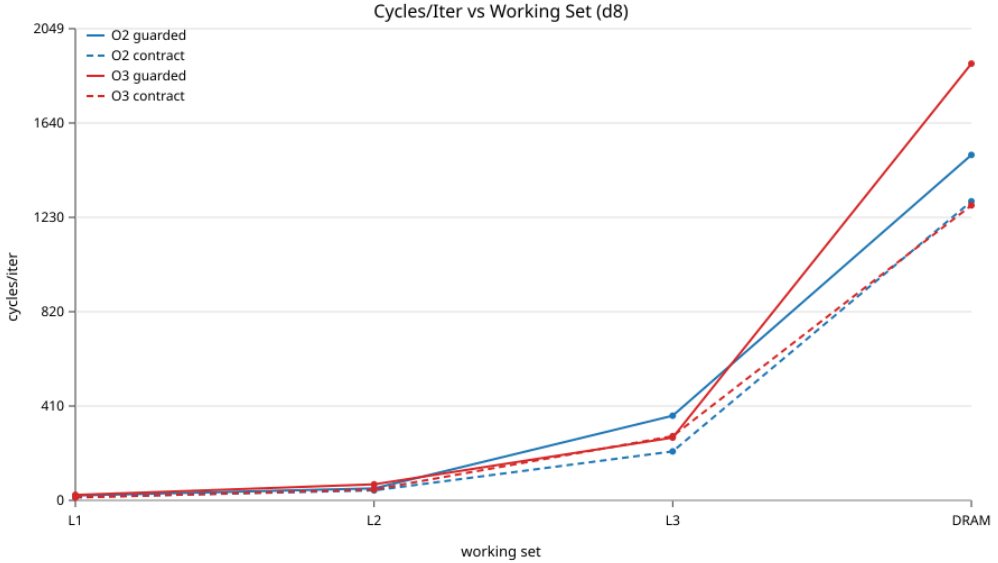
Experiment 3: Graph Analysis

- **Variables:** x-axis is mode; y-axis is branches/iter; bars compare default vs `-ImplicitNullChecks`, which forces explicit null checks.
- **Results:** intrinsic/guarded/contract increase from ≈ 9.6 to ≈ 20 branches/iter (about 2.08x); guarded_g1/g2 stay flat.
- **Behavior:** branch count rises sharply without implicit checks, but end-to-end time remains nearly unchanged, reinforcing that memory latency dominates.

Experiment 4: C Contract vs Guarded (Depth=8)

- **Purpose:** test contract-style control-flow collapse in a low-level C baseline (no JVM), so we can see if the idea is fundamentally beneficial before JIT complexity.
- **Method:** O2/O3 builds, depth=8, L1/L2/L3/DRAM working sets, measuring cycles/iter directly from perf.
- **Results:** contract lowers cycles/iter in L1/L2; results are mixed in L3/DRAM where latency dominates.
- **What it isolates:** pure compiler + hardware effects without JVM/GC, and the boundary where cache ends and latency begins.
- **Analysis:** this confirms the pattern in native code: in-cache, instruction overhead matters; out-of-cache, memory latency dominates.

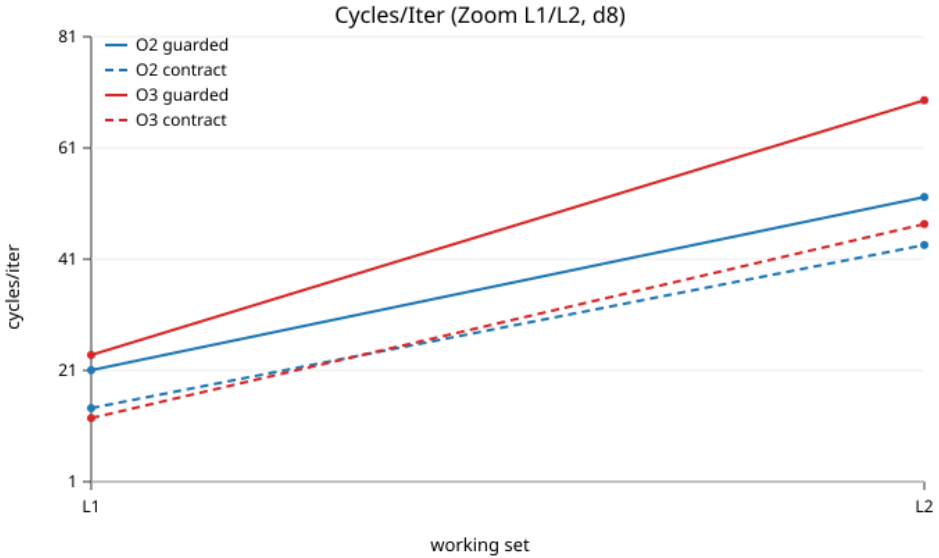
Experiment 4: C Contract vs Guarded (Graph)



Experiment 4: Graph Analysis

- **Variables:** x-axis is workset (L1/L2/L3/DRAM); y-axis is cycles/iter; lines show O2/O3 and guarded/contract.
- **Results:** contract is lower in L1/L2 for both O2/O3 (L1: $\approx$ 33–48%, L2: $\approx$ 17–32%).
- **Behavior:** L3/DRAM are mixed (O3 L3 regresses slightly), which is expected when the critical path is dominated by long-latency memory accesses.

Experiment 4: C Contract vs Guarded (Graph, Zoom L1/L2)



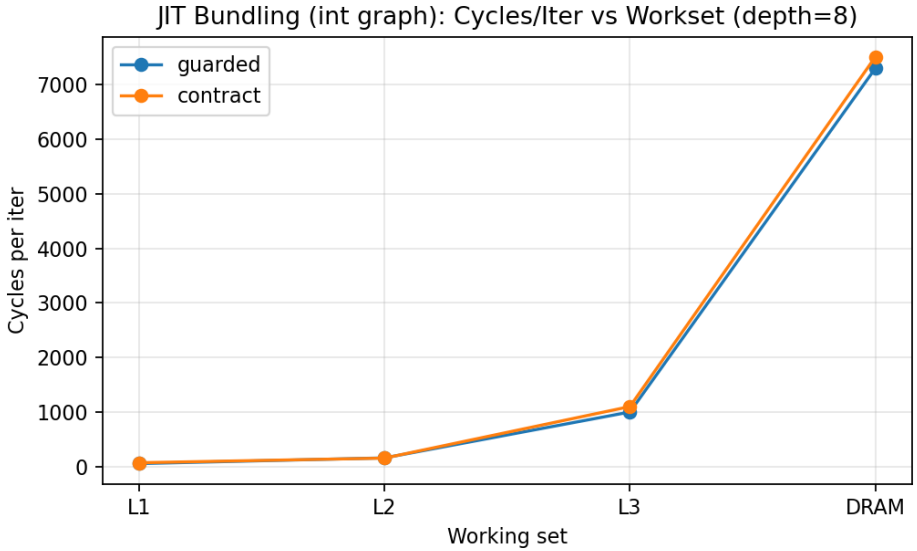
Experiment 4: Zoom Graph Analysis

- **Variables:** same as the main plot, zoomed to L1/L2 only to highlight small-cycle differences.
- **Results:** contract is consistently below guarded in L1/L2, with $\approx$ 33–48% lower cycles/iter in L1 and $\approx$ 17–32% in L2.
- **Behavior:** clear separation confirms control-flow reduction helps when data is cache-resident and the CPU is not stalled on DRAM.

Experiment 5: JIT Bundling (int graph)

- **Purpose:** test whether the JIT can bundle dependent pointer loads in an index graph (best-case for scalarization and tight loops).
- **Why bundle:** reduce per-step overhead such as checks/guards/branches and enable tighter codegen, ideally keeping the chase in registers.
- **Method:** HotSpot with tiered compilation disabled to reduce variability; depth=8 across L1/L2/L3/DRAM worksets.
- **Results:** contract cycles/iter are higher in L1/L3/DRAM and slightly lower in L2 (depth=8).
- **What it isolates:** whether the JIT can restructure a true dependency chain in a best-case representation.
- **Analysis:** bundling does not overcome dependent-load latency; the critical path still waits on each load before the next address is known.

Experiment 5: JIT Bundling (int graph) (Graph)



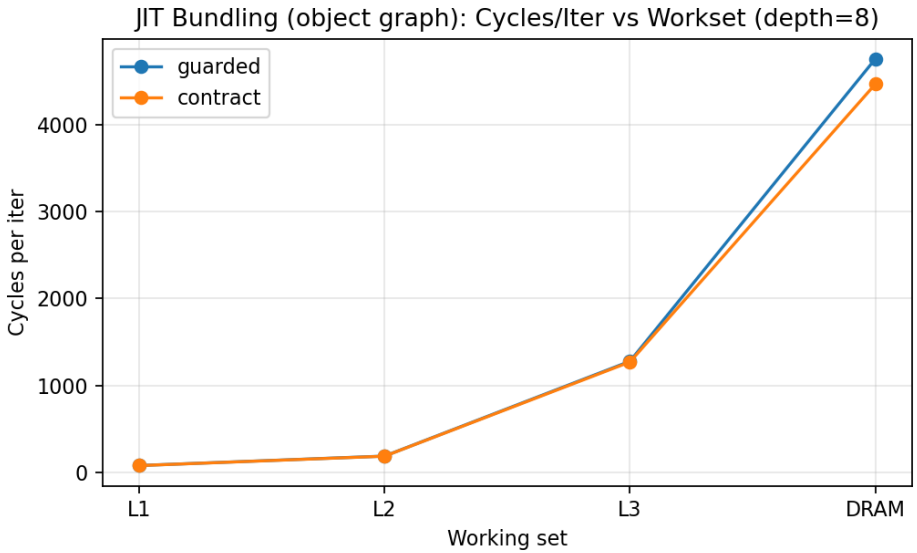
Experiment 5: Graph Analysis

- **Variables:** x-axis is workset; y-axis is cycles per iter (depth=8), so lower is better.
- **Results:** contract is +26% in L1, +10% in L3, +3% in DRAM, and ≈ -2.5% in L2 vs guarded.
- **Behavior:** bundling does not reduce cycles/iter for dependent int-graph loads; the dependency chain prevents overlap of load latency.

Experiment 6: JIT Bundling (object graph)

- **Purpose:** repeat the bundling test on real object graphs to capture GC barriers, object headers, and layout effects.
- **Method:** `dontinline` enforced for chaseGuarded/chaseContract to isolate JIT behavior and avoid inlining side effects.
- **Results:** contract cycles/iter are slightly lower across worksets (depth=8), but differences are small.
- **What it isolates:** whether object layout and GC mechanics change the bundling story.
- **Analysis:** even with realistic object layout overhead, the dependency chain remains the limiter, so gains are small.

Experiment 6: JIT Bundling (object graph) (Graph)



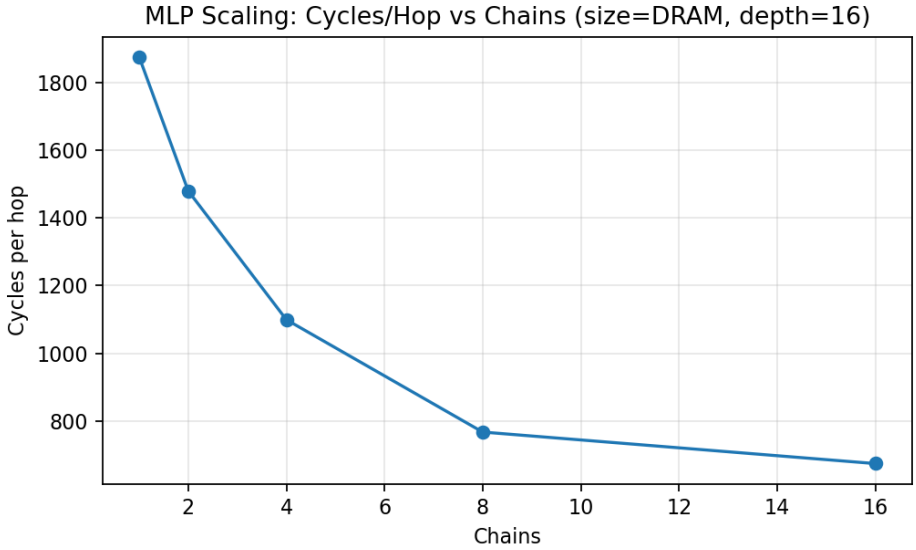
Experiment 6: Graph Analysis

- **Variables:** x-axis is workset; y-axis is cycles per iter (depth=8) on object graphs.
- **Results:** contract is about 0.6–6.1% lower cycles/iter than guarded, which is a small but measurable shift.
- **Behavior:** object layout and GC barriers do not change the latency-limited conclusion; differences are small and inconsistent.

Experiment 7: JIT MLP (DRAM, depth=16)

- **Purpose:** measure how much throughput improves if we explicitly increase memory-level parallelism (multiple independent chains).
- **Method:** DRAM working set, depth=16, chains = 1..16 (independent lists), so the CPU can overlap multiple outstanding misses.
- **Results:** cycles/hop drop as chains increase, from $\approx$ 1875 (1 chain) to $\approx$ 674 (16 chains).
- **What it isolates:** how much latency can be hidden by overlapping multiple misses (hardware MLP).
- **Analysis:** this is the most reliable lever we found; more chains allow the CPU to hide latency with overlap.

Experiment 7: JIT MLP (Graph)



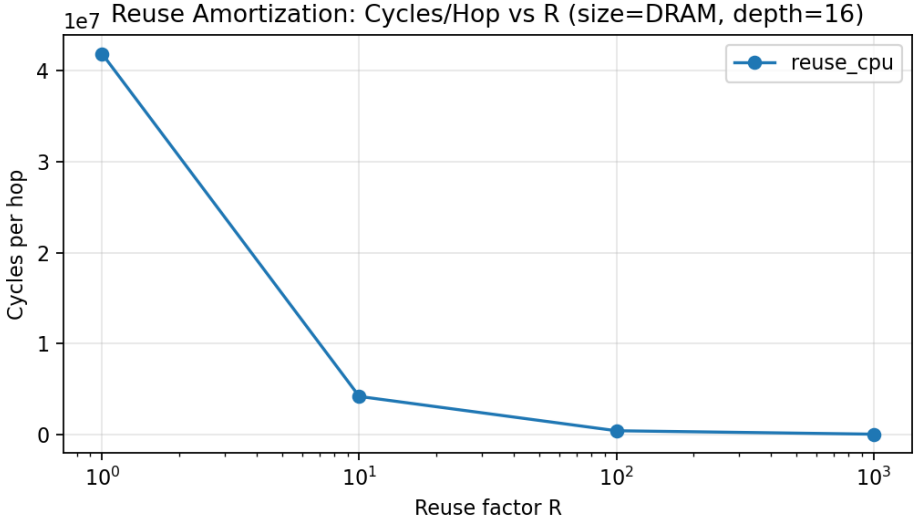
Experiment 7: Graph Analysis

- **Variables:** x-axis is number of chains; y-axis is cycles per hop (perf cycles / total hops), which reflects average time per pointer hop.
- **Results:** cycles/hop decreases steadily to ≈ 674 at 16 chains, showing real overlap of memory misses.
- **Behavior:** diminishing returns indicate saturation of available memory-level parallelism.

Experiment 8: Intel DSA Feasibility

- **Purpose:** assess whether a hardware offload (DSA) can help pointer chasing despite the dependency chain.
- **Method:** reuse-factor benchmark in the suite; DSA path is defined but was unavailable on this host.
- **Results:** reuse_cpu cycles/hop drop roughly 10x per decade of reuse (R=1 to 1000), but DSA data path was unavailable here.
- **What it isolates:** whether offload or reuse changes the fundamental dependency constraint.
- **Analysis:** offload engines help with copies/scans, but cannot break a serial load-to-use dependency chain.

Experiment 8: Intel DSA Feasibility (Graph)



Experiment 8: Graph Analysis

- **Variables:** x-axis is reuse factor R (log scale); y-axis is cycles per hop (perf cycles / total hops), so lower is better.
- **Results:** cycles/hop fall from $\approx$ 4.19e7 (R=1) to $\approx$ 4.19e4 (R=1000); reuse_dsa is absent here.
- **Behavior:** reuse amortizes fixed overhead per hop, but dependency latency still sets the lower bound.

Synthesis: What Each Experiment Eliminates

Experiment	What it rules out or confirms
Baseline	Measurement noise and unstable JVM warmup
Working set	Control-flow tricks matter once memory is slow
Null checks	Branch overhead as the primary limiter
Native C	JVM-specific artifacts as the root cause
JIT bundling	Instruction fusion defeating dependency latency
Object graphs	Layout/GC effects as the dominant factor
MLP	Shows the only consistent lever: parallel misses
DSA offload	Hardware offload breaking serial dependencies

Conclusion: Real Bottleneck

- The dominant bottleneck is dependent memory latency (load-to-use), not computation or branch overhead.
- Across C and HotSpot variants, changing checks/bundling rarely moves cycles/iter once the working set exceeds cache.
- MLP tests show the only consistent gains come from increasing outstanding misses and hiding latency.
- These experiments separate control-flow costs from the core latency bound, making the dependency chain the clear limiter.
- Practical implication: focus on prefetch/MLP or data-structure changes that reduce dependency depth, not extra instruction fusion.