Pointer Chasing as a Semantic Mega-Instruction

AOT + HotSpot Evidence and the Atomic Chase Prototype

Pointer-Chase Study

February 12, 2026

Agenda

1. Motivation and hypothesis
2. AOT evidence (GCC/Clang)
3. HotSpot evidence (JVM)
4. Combined view
5. Atomic chase prototype
6. Status and next steps

Motivation

Pointer chasing forces a serial dependency chain:

$$p_{i+1} = *(p_i + \Delta_i), \quad i = 0..(d-1)$$

This typically expands to:

$$\mathcal{O}(d) \text{ loads} + \mathcal{O}(d) \text{ guards} + \mathcal{O}(d) \text{ control-flow}$$

Goal: justify representing the entire chase as a single semantic unit.

Why This Approach?

- We need structural evidence, not only timing results.
- AOT shows what compilers emit; JIT shows guards and deopts.
- If both retain $\mathcal{O}(d)$ structure, a single semantic unit is justified.

AOT Experimental Matrix

Depths: 2, 3, 5, 8
Variants:

- Plain dereference: `**p, ***p, ...`
- Constant offsets: `*(p + c1) -> *(... + c2) -> ...`
- Dynamic offsets: offsets passed as arguments
- Explicit checks: per-level guard calls

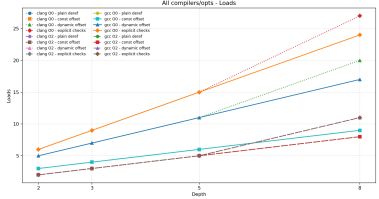
Metrics: loads, branches, basic blocks, total instructions.

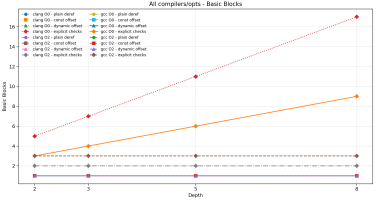
AOT Snapshot (GCC -O2)

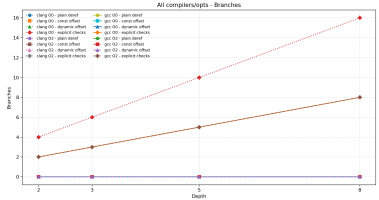
Pattern	Loads d2	Loads d8	Branches d2	Branches d8
plain	2	8	0	0
const	2	8	0	0
dyn	2	11	0	0
checks	2	11	2	8

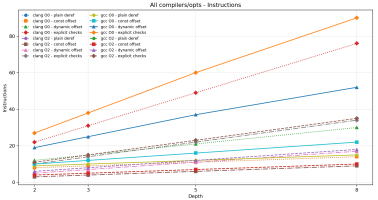
- Loads scale with depth for all patterns.
- Branches appear primarily with explicit checks.

AOT Results: Combined (All Compilers)









AOT Takeaways

- Loads scale linearly with depth across compilers and optimizations.
- Explicit checks induce $\mathcal{O}(d)$ branches and basic blocks.
- Optimization shrinks constants, but $\mathcal{O}(d)$ scaling remains.

HotSpot (JVM) Measurement

We extended the same matrix to HotSpot and extracted:

deref ops, guards (uncommon_trap), deopts, IR proxy counts

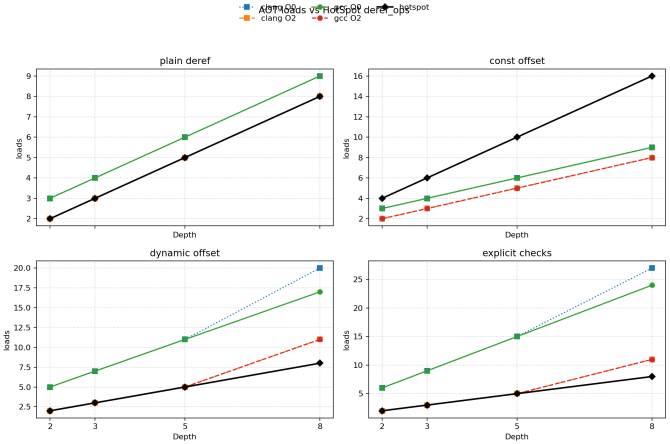
Note: Product JVMs do not expose C2 IR nodes. We use bytecode parse statistics as proxies and log all `uncommon_trap` entries.

HotSpot Snapshot (Proxies)

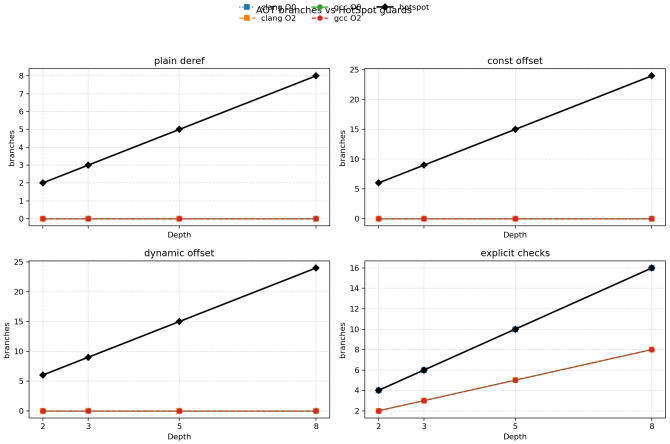
Pattern	Deref d2	Deref d8	Guards d2	Guards d8
plain	2	8	2	8
const	4	16	6	24
dyn	2	8	6	24
checks	2	8	4	16

- Deref ops scale with depth.
- Guards scale roughly with depth, mirroring AOT.

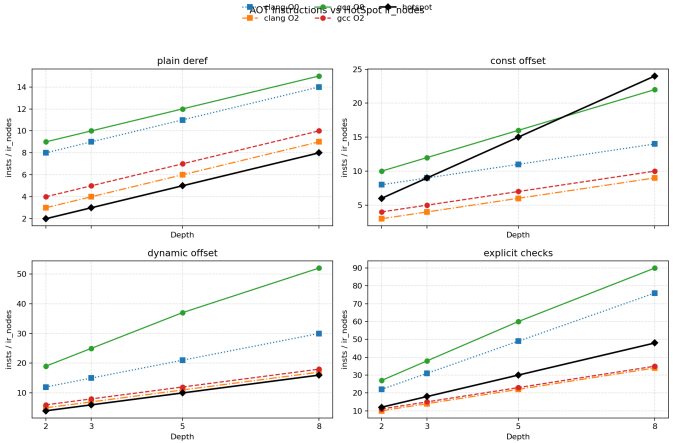
AOT vs HotSpot: Loads / Deref Ops



AOT vs HotSpot: Branches / Guards



AOT vs HotSpot: Instructions / IR Proxy



HotSpot Takeaways

- Deref operations scale linearly with depth.
- Guards and deopt points also scale roughly $\mathcal{O}(d)$.
- This mirrors the AOT structure and motivates a single semantic unit.

Atomic Chase: Semantic Mega-Instruction Model

Naive (current practice):

```
for (i = 0; i < depth; i++) {
    p = p->next;
}
return p->value;
```

Atomic (mega-instruction model):

```
if (!p || !p->next || ... ) return -1;
Node *q = p->next->next->...;
return q->value;
```

Checks are upfront; dataflow is straight-line with a single return.

Code Example: Naive Chase (C)

```c
int chase_naive(Node *p, int depth) {
    for (int i = 0; i < depth; i++) {
        p = p->next;
    }
    return p->value;
}
```

Code Example: Atomic Chase (C)

```
int chase_atomic(Node *p, int depth) {
    if (!p || !p->next || !p->next->next) return -1;
    Node *q = p->next->next;
    return q->value;
}
```

(Actual implementation is unrolled for depths 2,3,5,8.)

Code Example: Atomic Chase (Switch Excerpt)

```
switch (depth) {
  case 5:
    if (p && p->next && p->next->next &&
        p->next->next->next &&
        p->next->next->next->next &&
        p->next->next->next->next->next) {
      result = p->next->next->next->next->next->value;
    }
    break;
}
```

This models upfront checks + straight-line dataflow per depth.

Code Example: Benchmark Harness (C)

```
static volatile int sink;
for (int i = 0; i < ITERS; i++) {
    acc ^= chase_naive(head, depth);
}
for (int i = 0; i < ITERS; i++) {
    acc ^= chase_atomic(head, depth);
}
sink = acc;
```

Why Atomic Chase Now?

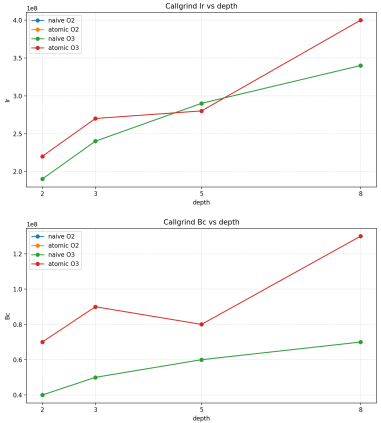
- The AOT/JIT evidence shows persistent $\mathcal{O}(d)$ structure.
- Atomic chase isolates the effect of control-flow collapse.
- We avoid changing algorithms to keep comparisons honest.

Atomic Chase Benchmark Harness

- Depths: 2, 3, 5, 8
- Tight loops ($\geq 10^7$ iters), identical inputs
- Metrics via `perf stat`: instructions, cycles, branches, branch-misses

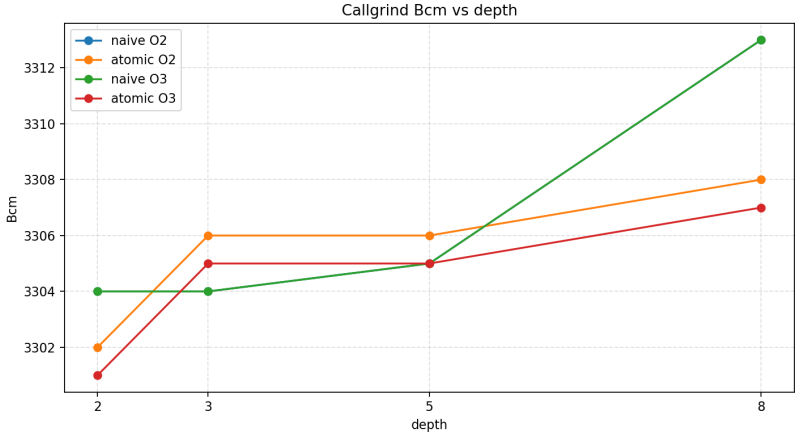
Note: hardware counters are unavailable in WSL; perf runs must be done on a native Linux machine.

Callgrind (WSL-safe) Results



- Ir = dynamic instruction refs (simulated)
- Bc = conditional branches executed
- Atomic has *more* branches due to explicit null checks

Callgrind: Branch Mispredicts



Mispredicts are low and nearly flat; branches are predictable in this synthetic setup.

Current Direction

1. Run perf on native Linux for atomic vs naive.
2. Prioritize branches and branch-misses as the primary signal.
3. If the structure collapses, formalize the semantic contract.

Status

- **Done:** AOT tables + graphs (GCC/Clang, -O0/-O2)
- **Done:** HotSpot tables + graphs (bytecode/guard proxies)
- **Done:** Atomic chase code + benchmark harness
- **Done:** Callgrind (WSL-safe) branch/instruction estimates
- **Pending:** Native Linux perf run + hardware counters

Next Steps

1. Run perf on native Linux (collect `instructions`, `branches`, `branch-misses`)
2. Produce atomic-vs-naive graphs and summary table
3. (Optional) Use fastdebug JVM to extract real C2 IR node counts

Conclusion

The data show a persistent $\mathcal{O}(d)$ structural cost in both AOT and JIT pipelines. The atomic chase model provides a concrete semantic target for collapsing this structure into a single unit.