Proof-Guided Dereference Collapse

Loop-Invariant Hoisting of Guarded Pointer Chains in LLVM

January 28, 2026

- C kernels use checked pointer operations (no undefined behavior, UB)
- LLVM pass builds a guarded heap-access graph (proof of purity)
- Collapse pass hoists invariant deref chains into a preheader (SSA)

1. Motivation

- Pointer dereference chains inside loops are expensive and branchy
- Guard checks make safety explicit but add repeated work
- If inputs are invariant, repeating guarded chains is pure waste

Goal

Replace repeated guarded dereferences in a loop with a single cached SSA value.

Scope: Checked-pointer API

- The LLVM pass only recognizes explicit `ck_*` calls
- Raw C pointer ops are not analyzed or rewritten
- Safety is encoded in `ck_*` guards + loads (no UB)
- Whole-program use would require a separate source-to-source rewrite

2. Baseline C code

Baseline

```c
for (uint64_t k = 0; k < iters; ++k) {
  Eval r = triple_deref(heap, p, 0);
  sum += r.value;
}
```

- Each iteration performs guards + 3 loads
- Work scales with loop trip count
- No reuse across iterations

Analysis: Baseline C

- Each iteration re-enters the guarded dereference chain.
- All guards and loads are repeated even when inputs are invariant.
- The loop body cost scales with chain length $\times$ trip count.

3. Optimized C (conceptual)

Hoisted

```
Eval cached = triple_deref(heap, p, 0);
for (uint64_t k = 0; k < iters; ++k) {
  Eval r = cached;
  sum += r.value;
}
```

- Compute once in preheader
- Loop uses SSA value only
- Eliminates repeated guards + loads

Analysis: Hoisted C

- The guarded chain is evaluated once in the preheader.
- The loop consumes a single SSA value per iteration.
- Repeated guards and loads are removed from the hot path.

4. Guarded semantics (baseline vs optimized)

Baseline (inside loop)

```
vp   = guard_ptr(p)
v1   = guard_nonnull(vp)
v2   = load_ptr(v1)
v3   = guard_ptr(v2)
v4   = guard_nonnull(v3)
v5   = load_ptr(v4)
v6   = guard_ptr(v5)
v7   = guard_nonnull(v6)
v8   = load_ptr(v7)
return v8
```

Optimized

```
// preheader
vp   = guard_ptr(p)
...
cached = load_ptr(v7)

// inside loop
return cached
```

Analysis: Guarded Semantics

- Baseline repeats a pure chain of guards + loads.
- Optimized version preserves the same checks but runs them once.
- Semantic proof: the guarded chain is side-effect free.

5. LLVM IR (baseline)

```
loop:
  %val = call { i64, i32 } @triple_deref(...)
  %v   = extractvalue { i64, i32 } %val, 1
  %sum = add i64 %sum, %v
  br label %loop
```

- `call @triple_deref` sits inside the loop body
- Repeated guard+load chain every iteration

Analysis: LLVM IR Baseline

- The call is inside the loop, so costs multiply by trip count.
- The loop depends on a pointer-chasing call each iteration.
- No reuse of the computed value.

6. LLVM IR (optimized, SSA)

```
preheader:
  %hoisted = call { i64, i32 } @triple_deref(...)
  br label %loop

loop:
  %v = extractvalue { i64, i32 } %hoisted, 1
  %sum = add i64 %sum, %v
  br label %loop
```

- One call in preheader, no loads from cache
- SSA value dominates all loop uses

Analysis: LLVM IR Optimized

- The call is hoisted to the preheader and dominates the loop.
- The loop uses a single SSA value – no memory loads.
- This is loop-invariant code motion with a proof of purity.

7. Assembly intuition

Baseline

```
loop:
  call triple_deref    ; guard+load chain
  add  s0, s0, a0
  j    loop
```

Optimized

```
preheader:
  call triple_deref
loop:
  add  s0, s0, a0      ; cached SSA value
  j    loop
```

Analysis: Assembly Intuition

- Baseline spends cycles in a call + dependent loads per iteration.
- Optimized loop is reduced to arithmetic on a cached value.
- Pointer chasing is removed from the hot loop body.

8. Correctness argument

- **Loop invariance:** inputs to `triple_deref` are invariant in the loop.
- **Purity:** guarded deref graph contains only guards + loads, no side effects.
- **Dominance:** preheader dominates loop body, so SSA value is available.

Key reasoning

If f is pure and x is loop-invariant, then $f(x)$ is loop-invariant. Hoisting preserves semantics.

9. Benchmark methodology

- **Unoptimized benchmark:** volatile sink inside loop (noisy, hides benefit).
- **SSA benchmark:** register accumulator + single compiler barrier.
- Same heap layout and `triple_deref` implementation for comparability.
- Run config: 50,000,000iterations, 7runs.

10. Benchmark results

Variant	Base mean (ms)	Base ns/iter	Opt mean (ms)	Opt ns/iter	Speedup
Unoptimized	1632.35	32.647	95.75	1.915	17.30
SSA	1642.58	32.852	61.39	1.228	26.88

Table: Mean timings over 7runs at 50,000,000iterations.

- Microbenchmarks amplify savings (loop body becomes very small)
- SSA benchmark isolates the optimization effect and lowers variance

11. Generalization

- Extend beyond `triple_deref` to any linear guarded dereference chain
- Recognize patterns at IR level (guards + loads + fields)
- Hoist when alias + memory stability are proven (MemorySSA/AA)

12. From demo to production optimizer

- **Production-ready:** SSA hoist, dominance checks, preheader insertion
- **Missing for real C/C++:** MemorySSA + alias analysis for heap stability
- **Volatile/atomic handling:** must avoid speculation across side effects
- **Profitability:** TTI cost models, loop trip counts, reg pressure
- **Robustness:** remove dependence on runtime JSON graphs
- **Still valid:** proof-guided LICM at IR level, target-independent