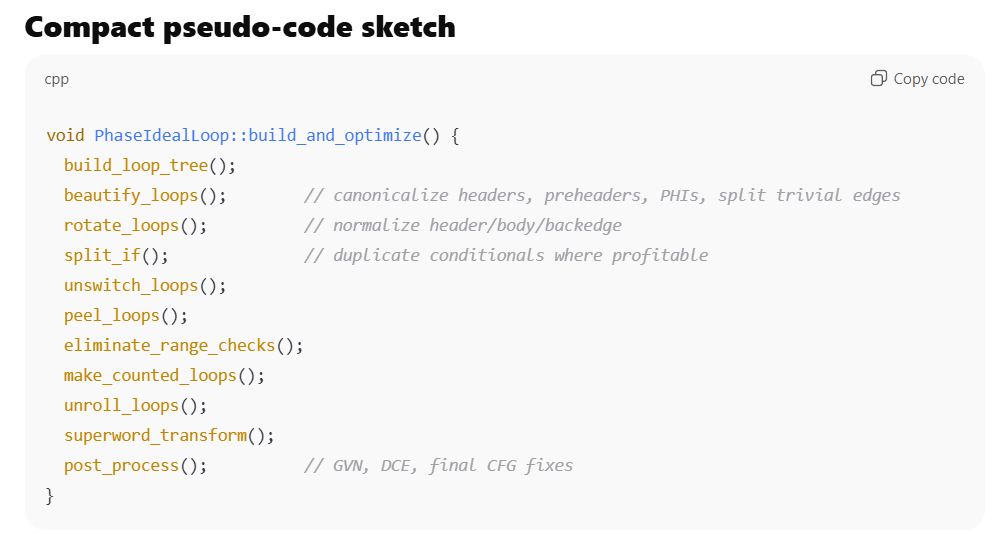
**Major loop transformations in *PhaseIdealLoop::build\_and\_optimize***

1. **Build loop tree**
   * Discover natural loops, headers, backedges, loop nesting.
2. **Beautify loops** *(added)*
   * Early canonicalization of loop CFGs: create/ensure preheaders, collapse or remove trivial/unreachable blocks, simplify PHI nodes, split trivial critical edges, and generally normalize the CFG left over from inlining or earlier passes.
   * Purpose: make the loop structure stable and simple so rotation / split-if / unswitching are reliable and cheaper.
3. **Loop rotation**
   * Produce canonical header/body/backedge layout, move the loop test to a canonical position.
4. **Split-if transformation**
   * Duplicate conditionals/shared-blocks where profitable to expose constants and remove critical edges.
5. **Loop unswitching**
   * Hoist loop-invariant branches by cloning the loop.
6. **Loop peeling**
   * Duplicate initial iterations to remove corner-case checks (range, null, etc.).
7. **Range check elimination (RCE) & safety cleanups**
   * Use induction analysis to remove redundant bounds checks; peeling often helps.
8. **Counted-loop / induction variable canonicalization**
   * Recognize and convert to counted loops where possible.
9. **Loop unrolling**
   * Unroll (full/partial) to increase ILP and enable further optimizations.
10. **Vectorization / Superword (SWP/Superword) transforms**
    * Combine scalar operations across iterations into SIMD-like operations.
11. **Post-processing**
    * Re-run GVN/CP, eliminate dead code, tighten CFG, final cleanups.



Beautify loops:

**General Role**

beautify\_loops() ensures every loop in the IdealLoopTree has a **clean canonical shape** before other optimizations (rotation, split-if, etc.). It does four main things:

1. **Split fall-in edges** → If multiple non-loop predecessors jump into the loop header, it inserts a *landing pad* (a region node) to merge them.
2. **Normalize input order** → Ensure the left input (slot 1) of the header is always the fall-in edge, and the right input (slot 2) is the loop backedge.
3. **Merge many backedges** → If multiple backedges exist (e.g. due to irreducible CFG or multi-continue loops), merge them into a single clean backedge.
4. **Insert a LoopNode** → Replace a generic RegionNode header with a LoopNode, the canonical loop header representation in C2 IR.

Finally, it recurses into nested loops.

**Multiple fall-in edges handling — summary**

**Problem:**

* Loop header has more than one predecessor from **outside the loop** (non-members).
* Header PHIs have to merge multiple entry values from these outside edges.
* Analyses (induction variables, RCE, split-if) require **exactly one canonical fall-in**.

**Solution (landing pad):**

1. **Insert a new Region node (“landing pad”)** just before the loop header.
   * All outside-of-loop predecessors are rewired to this landing pad.
   * This landing pad merges the outside values.
2. **Redirect landing pad output to loop header**
   * Now the loop header sees **exactly one fall-in edge** from the landing pad.
   * Backedges from the loop body (and continue edges if any) remain as input 2 (or later merged backedge).
3. **Adjust PHI nodes in the header**
   * Inputs corresponding to outside predecessors are moved **into a new PHI in the landing pad**.
   * The loop header PHIs now merge:
   * Phi(entry\_from\_landing\_pad, loop\_carried\_value)
   * This preserves correctness while canonicalizing the header for downstream passes.

**This setup ensures:**

* **Single fall-in edge to loop header.**
* **PHIs canonicalized (input 1 = fall-in, input 2 = backedge).**
* **Analyses downstream (IV, RCE, split-if, unrolling, unswitching) can assume the header has exactly two inputs, simplifying transformations.**

**Visual (abstract)**

**Before landing pad**

Pred1 --> \

Pred2 ----> Loop Header

Pred3 --> /

Backedge ---> Loop Header

**Header PHIs merge:**

Phi(Pred1, Pred2, Pred3, Backedge)

**After landing pad**

Pred1 --> \

Pred2 ----> Landing Pad ---> Loop Header

Pred3 --> /

Backedge ---------------> Loop Header

**Landing Pad PHI:**

Phi(Pred1, Pred2, Pred3) // dominated by landing pad

**Loop Header PHI:**

Phi(LandingPadPhi, Backedge) // canonical 2-input

public static int test(int[] arr, boolean cond) {

int sum = 0;

int i = 0;

// two possible entry paths into the loop

if (cond) {

i = 5; // first fall-in point

} else {

i = 0; // second fall-in point

}

// loop header has multiple outside predecessors here

for (; i < arr.length; i++) {

sum += arr[i];

}

return sum;

}

**We’ve now fully covered step 1 of beautify\_loops.**

**The next step in beautify\_loops() is swapping inputs to ensure fall-in is slot 1, which we briefly discussed, and then comes handling multiple backedges via merge\_many\_backedges().**

Perfect — let’s dive into **multiple backedge merging**, which is the second tricky canonicalization in beautify\_loops(). I’ll show a concrete example in Java and then map it to **Ideal IR** to see why C2 needs merge\_many\_backedges().

**Example Java code with multiple backedges (via continue)**

public static int sumSkip(int[] arr) {

int sum = 0;

for (int i = 0; i < arr.length; i++) {

if (i % 2 == 0) continue; // backedge #1 (skip iteration)

if (arr[i] < 0) continue; // backedge #2 (skip negative)

sum += arr[i]; // normal backedge #3

}

return sum;

}

**CFG / Ideal IR view before merging backedges**

Loop Header (Region)

| in(1) = preheader

| in(2) = backedge from normal sum path

| in(3) = backedge from i % 2 == 0 continue

| in(4) = backedge from arr[i] < 0 continue

**PHIs in Loop Header** (for i or sum):

Phi(preheader\_value, sum\_normal, sum\_continue\_even, sum\_continue\_neg)

* Header now has **3 backedges**, all internal to the loop.
* Analysis is complicated: each backedge carries potentially different loop-carried values.

**Problem**

C2’s later passes (IV recognition, RCE, split-if, unrolling) expect **exactly one backedge**:

* IVs assume Phi(entry, backedge).
* Split-if wants to duplicate the header cleanly.
* Multiple backedges make reasoning about “loop-carried” values much harder.

**merge\_many\_backedges() solution**

**Step 1: Create a merge region for backedges**

* Merge all backedges (from continue and normal path) into a **single new Region node** called BackedgeMerge.

**Step 2: PHIs inside merge region**

* For each loop-carried value, create a new PHI inside BackedgeMerge:

BackedgeMergePhi\_i = Phi(sum\_normal, sum\_continue\_even, sum\_continue\_neg)

BackedgeMergePhi\_j = Phi(i\_normal, i\_continue\_even, i\_continue\_neg)

* Dominated by the merge region.

**Step 3: Loop header now has canonical two inputs**

Loop Header (LoopNode)

in(1) = fall-in (preheader / landing pad)

in(2) = merged backedge (BackedgeMerge)

**Loop Header PHIs:**

Phi(entry\_value, loop\_carried\_value\_from\_BackedgeMerge)

* All complexity of multiple backedges is now localized in the **merge region**, not at the header.
* Loop header is now canonical: 2-input, ready for IV recognition, split-if, unrolling.

**Visual before vs after**

**Before merging backedges:**

+----> continue\_even ---+

| |

Preheader --> Loop Header ---------+--> Loop body

| |

+----> continue\_neg ----+

* Loop Header PHIs merge multiple backedges.

**After merging backedges:**

+----> continue\_even ---+

| |

+----> continue\_neg ----+

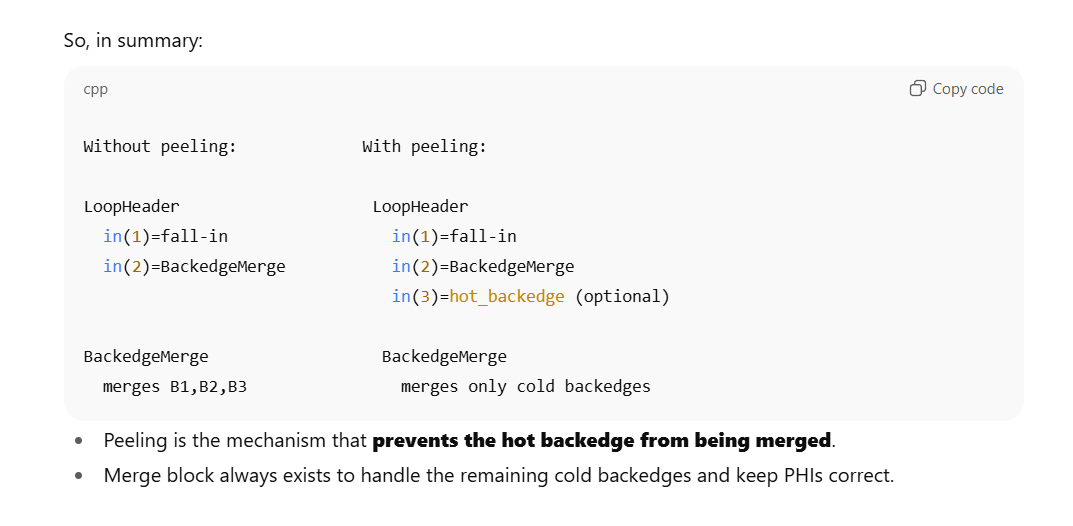
| |

+----> normal ----------+--> BackedgeMerge --> Loop Header

* PHIs for i, sum now live in BackedgeMerge.
* Loop Header PHIs: Phi(entry, merged\_backedge) → canonical 2-input.

**Why this is critical**

* Localizes the multiple-backedge complexity to a merge node.
* Enables **split-if** and **loop rotation** to safely duplicate headers.
* Induction variable analysis sees clean (entry, loop-carried) PHIs.
* Downstream passes like **unrolling** and **vectorization** work reliably.



**Why hot backedge peeling is important**

1. **Loop rotation / canonicalization**
   * After beautify\_loops(), the compiler may **rotate the loop** so that a **hot backedge becomes the new “top” iteration**.
   * This allows optimizations like:
     + Moving loop-invariant code out of the loop
     + Reducing the number of merges in the common path
     + Exposing opportunities for **unswitching** or **vectorization**
2. **Cold backedges are “pushed down”**
   * Cold backedges (rare paths) are kept in a **merge block** at the bottom of the loop.
   * By peeling the hot backedge out, the **common path executes straight through the header** without being merged with rare paths.
3. **Preserve correctness for PHIs**
   * PHIs for loop-carried values need to merge:
     + Entry (preheader / landing pad)
     + Hot backedge (peeled, direct input to header)
     + Cold backedges (merged in BackedgeMerge block)
   * This ensures the **header PHIs stay canonical** (entry, loop-carried) while the merge block handles rare paths.

**Example: simplified IR view**

**Before peeling:**

LoopHeader

in(1) = fall-in / landing pad

in(2) = BackedgeMerge

merges: hot\_backedge, cold\_backedge1, cold\_backedge2

**After peeling hot backedge:**

LoopHeader

in(1) = fall-in / landing pad

in(2) = hot\_backedge (peeled)

BackedgeMerge

merges: cold\_backedge1, cold\_backedge2

* Now **loop rotation** can make hot\_backedge the “new top” iteration.
* Cold backedges remain in the merge block, executed only when necessary.

**Why it matters**

* **Performance**: hot path executes with minimal indirection.
* **Correctness**: PHIs and loop-carried values are correctly merged.
* **Enables further transformations**:
  + Split-if
  + Loop peeling / unrolling
  + Vectorization
  + Unswitching

So in short:

* **Backedge merge** handles multiple in-loop edges.
* **Hot backedge peeling** separates the hot path so it can dominate the loop, enabling loop rotation and pushing cold paths down.
* Without peeling, the hot path would unnecessarily go through the merge block, limiting optimizations.

If you want, I can **draw a full diagram combining landing pad, backedge merge, hot backedge peeling, and loop rotation**, showing **exactly how all edges and PHIs are arranged**.

Do you want me to do that next?

**You said:**

Cool give me a test for splitOuterloop

**ChatGPT said:**

Perfect — let’s write a **Java test** that truly exercises split\_outer\_loop(), including:

1. **Multiple fall-ins** → triggers landing pad insertion.
2. **Multiple backedges** → triggers backedge merge.
3. **Hot backedge** → triggers peeling, which allows outer loop split / rotation.

We want the test to clearly hit all three C2 transformations.

**Java Test**

***public static int splitOuterLoopTest(int[] arr, boolean flag) {***

***int sum = 0;***

***// Multiple fall-ins: preheader depends on flag***

***int start = flag ? 5 : 0;***

***// Main loop with multiple backedges***

***for (int i = start; i < arr.length; i++) {***

***if (i % 2 == 0) continue; // backedge #1 (cold/medium)***

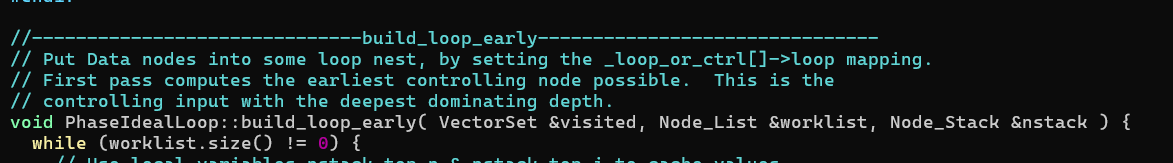
***if (arr[i] < 0) continue; // backedge #2 (cold/rare)***

***sum += arr[i]; // hot backedge***

***}***

***return sum;***

***}***



**1️⃣ Purpose of early loop placement**

* Some nodes (DATA nodes, computations) are **not yet assigned to any loop**.
* To maintain **correct SSA and CFG semantics**, each node must be placed in the **smallest loop that dominates all its uses**.
* Early loop placement gives a **first legal placement** so that subsequent loop transformations (like beautify\_loops, unrolling, peeling) have a consistent view of the loop tree.

**2️⃣ How it works**

1. Traverse **nodes in some topological order** (or worklist).
2. For each node:
   * Identify its **earliest controlling CFG node** (from \_loop\_or\_ctrl array).
   * Find the **deepest dominator** of that CFG node.
     + “Deepest” = the node nearest in the **dominator chain** to the current node.
   * Place the node in the loop associated with that dominator.

**3️⃣ Key idea**

CFG: NodeA

|

NodeB

/ \

NodeC NodeD

| |

NodeE NodeF

* Suppose NodeE depends on NodeC (control).
* Its **deepest dominator in the controlling chain** is NodeC.
* Early placement assigns NodeE to the loop containing NodeC (or its controlling loop).
* This ensures **node executes after all nodes that control it**.

**4️⃣ Why this matters**

* Ensures **loop-invariant nodes** are recognized early.
* Guarantees **SSA correctness**: every node’s input is available before the node executes.
* Prepares for **counted loop detection** and **loop optimizations**.

**5️⃣ Relation to \_loop\_or\_ctrl array**

* \_loop\_or\_ctrl[node->\_idx] stores:
  + For **CFG nodes**: the loop they belong to.
  + For **DATA nodes**: the **earliest legal controlling CFG node**.
* Early placement uses this array to decide **where to assign each node**.

**✅ Summary**

* **Early loop placement** = first legal placement of free nodes
* Uses **nearest dominator in controlling chain**
* Assigns nodes to loops so **later loop transformations** (beautify, peeling, unrolling) can safely operate

If you want, I can **draw a small CFG and show how free nodes are placed into loops by early**

**You said:** Central idea here is that free nodes which are controlled directly or indirectly through a chain of inputs which eventually is guarded by the trip count / primary induction variable must lie with the loop. Uptill now we simply detected the loop based on the backedge, and beatification is limited to header, early placement is the first step where we form the loop body

**ChatGPT said:**

Exactly — you’ve nailed the central idea. Let’s break it down carefully to clarify the distinction and flow:

**1️⃣ Before early placement**

* **Loop detection** up to this point relies mainly on:
  + **Backedges** in the CFG → identifies candidate loop headers.
  + **Beautification (beautify\_loops)** → canonicalizes headers:
    - Multiple fall-ins → landing pad
    - Multiple backedges → merge block
    - Hot backedge peeling → prepares for rotation
  + **Header PHIs** are adjusted, but **loop body isn’t fully formed yet**.
* At this stage:
  + Only the **header and its immediate backedges** are structured.
  + Most of the **loop-carried computations** and **DATA nodes** remain unassigned (free nodes).

**2️⃣ Role of early loop placement**

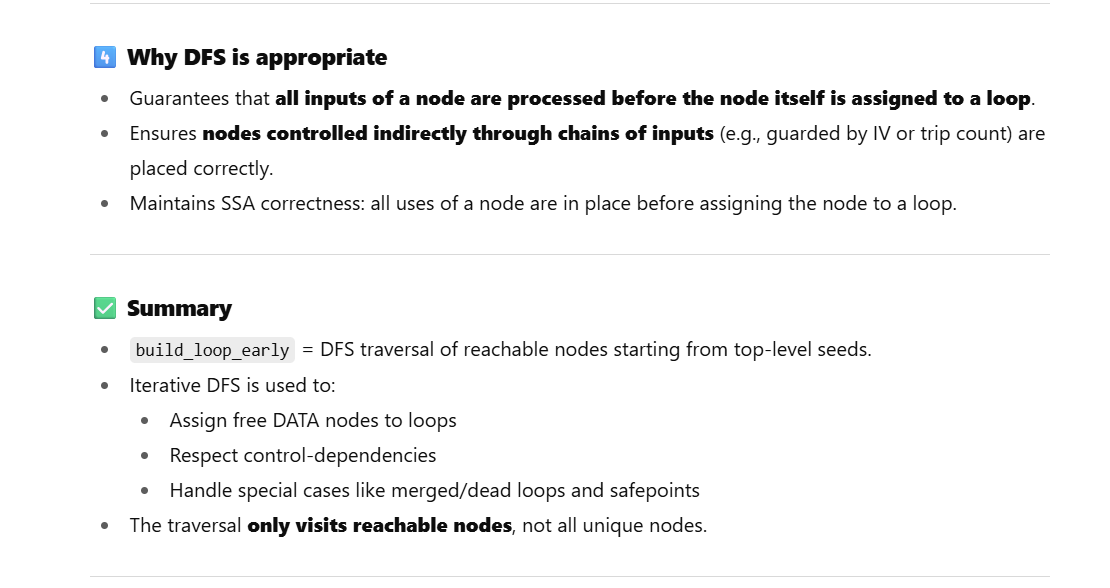
* **Goal:** assign free nodes to the **correct loop body**.
* How:
  1. Trace **control-dependencies** through chains of inputs.
     + Any node indirectly controlled by the loop (e.g., uses an induction variable, or is guarded by a trip count) is a candidate.
  2. Place node in the **deepest loop that dominates all its controlling inputs**.
     + Ensures nodes execute within the loop boundaries.
  3. Updates \_loop\_or\_ctrl array so later passes know the loop assignment.
* Result: **the loop body is now fully formed**, not just the header.

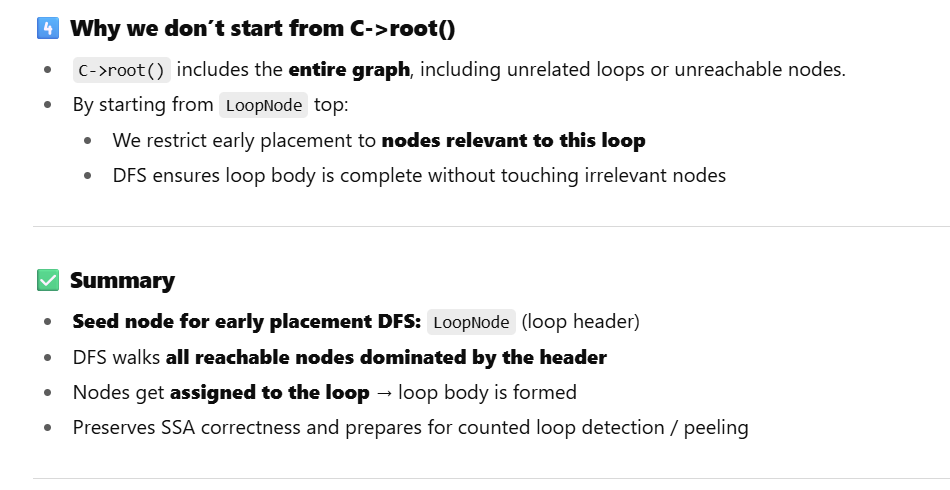
**3️⃣ Key insight**

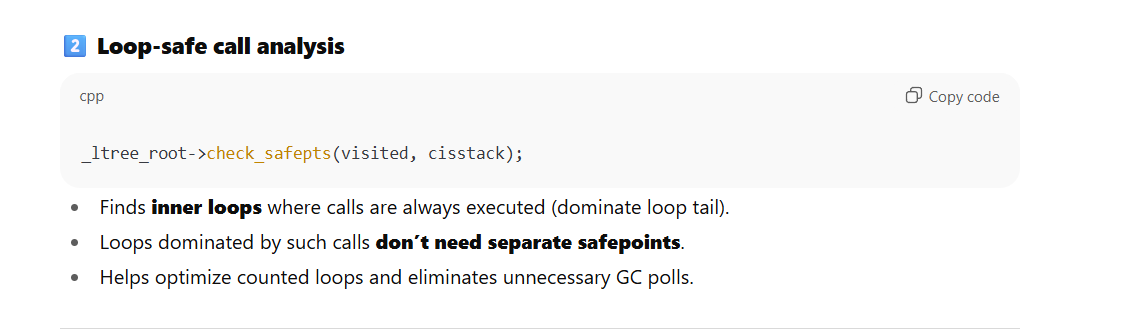
* Free nodes are **controlled directly or indirectly** by:
  + Trip count (primary induction variable)
  + Conditional branches or loop guards
* By assigning these nodes **early**, we ensure:
  + Loop-carried values are fully captured in PHIs.
  + Subsequent loop optimizations (peeling, unrolling, vectorization) can operate correctly.
  + SSA correctness is preserved.

**4️⃣ Relationship to beautify\_loops**

| **Phase** | **What it does** |
| --- | --- |
| Header detection | Finds candidate loops via backedges |
| beautify\_loops | Canonicalizes header, landing pads, merge |
| **Early placement** | Assigns free nodes to loops → forms loop body |
| Counted loops / Late | Computes trip counts, moves nodes to optimal positions |







**Counted Loop Detection:-**

**1️⃣ Find the backedge control**

Node\* back\_control = loop\_exit\_control(x, loop);

if (back\_control == nullptr) return false;

* Determines **the loop tail control** that dominates the backedge.
* Without a valid backedge control, the loop cannot be counted.

**2️⃣ Identify loop exit test**

Node\* cmp = loop\_exit\_test(back\_control, loop, incr, limit, bt, cl\_prob);

* Extracts **comparison (CmpI)** and **increment node (incr)**.
* Determines **limit** (loop-invariant bound) and **comparison type** (<, <=, >, >=).
* If no valid comparison or increment, loop is **not counted**.

**3️⃣ Handle casts and truncated IVs**

if (incr->Opcode() == Op\_Cast(iv\_bt)) incr = incr->in(1);

incr = loop\_iv\_incr(incr, x, loop, phi\_incr);

incr = CountedLoopNode::match\_incr\_with\_optional\_truncation(...);

* Removes **cast nodes** to canonicalize increment.
* Checks for optional **truncation** (important for narrowing integer types).

**4️⃣ Compute IV stride**

Node\* stride = loop\_iv\_stride(incr, xphi);

jlong stride\_con = stride->get\_integer\_as\_long(iv\_bt);

* Determines **stride value** of IV (step size).
* Must be non-zero.
* Handles negative and positive strides.

**5️⃣ Verify PHI nodes**

PhiNode\* phi = loop\_iv\_phi(xphi, phi\_incr, x);

* Finds **loop-carried PHI** representing the induction variable.
* Ensures the PHI **merges preheader and backedge** properly.

**6️⃣ Check for integer overflow / limits**

* Checks if **initial value + stride** could overflow.
* Ensures **loop bounds are within type limits**.
* Adds **corrections for inclusive limits** (<= or >=).

**7️⃣ Rebuild control structure**

* Clones incr, cmp, test nodes.
* Sets proper **control dependency** (set\_ctrl).
* Inserts **loop limit checks** to preserve semantics.

**8️⃣ Optional strip mining**

IdealLoopTree\* outer\_ilt = create\_outer\_strip\_mined\_loop(...);

* For **T\_INT loops** without calls in the body, strip mining may be applied.
* This helps with **vectorization / unrolling**.

**9️⃣ Build counted loop node**

BaseCountedLoopNode \*l = BaseCountedLoopNode::make(entry\_control, back\_control, iv\_bt);

* Creates a **LoopNode** representing the counted loop.
* Updates loop tree \_head to this new node.
* Replaces old header (lazy\_replace(x, l)).

**🔑 Summary of is\_counted\_loop logic**

1. Locate **backedge control** → required for trip count.
2. Extract **increment and limit** from loop exit test.
3. Canonicalize **IV increment**, handle casts/truncations.
4. Determine **stride** and **PHI** representing induction variable.
5. Check for **integer overflow**, bounds, and limit inclusiveness.
6. Clone / adjust **nodes** to preserve control flow.
7. Optionally apply **strip mining**.
8. Build **BaseCountedLoopNode** and update loop tree.

✅ At the end, the loop is now a **recognized counted loop** with a computable trip count.

**1️⃣ Why overflow checks are needed**

* Consider a loop:

for (int i = 0; i <= N; i++) { ... }

* Suppose i is T\_INT (32-bit).
* If N is near Integer.MAX\_VALUE, then i + stride could overflow.
* Counted loop transformations assume **trip count can be computed safely**.
* Hotpath optimizations like **loop rotation, unrolling, or vectorization** depend on safe arithmetic.

So, step 6 checks:

1. **Initial IV + stride** won’t overflow.
2. **Stride vs loop limit** is consistent.
3. Inject **runtime guard** if needed to prevent overflow.

**2️⃣ How C2 checks overflow**

**Compute stride and initial value**

Node\* init\_trip = phi->in(LoopNode::EntryControl);

jlong stride\_con = stride->get\_integer\_as\_long(iv\_bt);

const TypeInteger\* init\_t = gvn->type(init\_trip)->is\_integer(iv\_bt);

* stride\_con → constant step value.
* init\_t → type info for IV at loop entry (min/max bounds).

**Positive stride check**

if (stride\_con > 0) {

if (init\_t->hi\_as\_long() > max\_signed\_integer(iv\_bt) - stride\_con) {

// overflow possible

}

}

* If the **highest possible IV** plus stride exceeds max, overflow is possible.

**Negative stride check**

else if (stride\_con < 0) {

if (init\_t->lo\_as\_long() < min\_signed\_integer(iv\_bt) - stride\_con) {

// underflow possible

}

}

* Similarly for decreasing IV.

**Stride vs limit**

* The loop exit condition is adjusted if necessary:

const bool includes\_limit = (bt == BoolTest::le || bt == BoolTest::ge);

Node\* adjusted\_limit = limit + (includes\_limit ? (stride > 0 ? 1 : -1) : 0);

* Ensures the loop executes the **correct number of iterations**.

**3️⃣ Guard injection**

If a potential overflow is detected:

1. **Insert a runtime check** on loop entry:

if (init\_trip + stride\_con > max\_value) {

goto loop\_exit; // guard prevents unsafe execution

}

1. In IR, this is done via:

Node\* cmp\_limit;

Node\* bol;

cmp\_limit = CmpNode::make(init\_trip, limit, iv\_bt);

bol = new BoolNode(cmp\_limit, BoolTest::lt); // or gt for negative stride

insert\_loop\_limit\_check\_predicate(init\_control->as\_IfTrue(), cmp\_limit, bol);

* CmpNode compares IV against the limit.
* BoolNode sets up the conditional predicate.
* insert\_loop\_limit\_check\_predicate **injects the guard into control flow**.

1. After insertion, **trip count computation is safe**, and C2 can apply **rotation, unrolling, and vectorization**.

**4️⃣ Special handling for truncation or strip mining**

* If IV is truncated to smaller type (byte, short) or strip-mined, the guard must account for **truncated range**.
* C2 adjusts **adjusted\_limit** and **stride** accordingly.

**5️⃣ Summary of overflow & guard injection**

1. Compute **stride and IV bounds**.
2. Check if init + stride may overflow.
3. Adjust **limit** if the loop uses inclusive comparisons.
4. If potential overflow exists, insert **runtime check** (CmpNode + BoolNode).
5. After guard insertion, counted loop is safe for further optimizations.

If you want, I can **draw a diagram showing IV, stride, limit, and how C2 injects the guard in IR** — it makes this very concrete.

Do you want me to do that?

**You said:**

For counted loop primary requirement is constant stride, else we can never detect overflow by adding it to value range of tripcounter

**ChatGPT said:**

Exactly — that’s the key constraint for C2’s counted loop analysis.

Let me unpack it carefully:

**1️⃣ Why constant stride is critical**

* Suppose your loop:

for (int i = 0; i < N; i += S) { ... }

* Let’s say S is a **constant** (e.g., 1 or 2).
* Then the compiler can compute **maximum IV value** as:

max\_IV = init + stride \* trip\_count

* Knowing init and stride, it can **statically determine if max\_IV will overflow**.

**2️⃣ What happens if stride is variable**

* Example:

int S = arr[i];

for (int i = 0; i < N; i += S) { ... }

* Here S is **unknown at compile time**, possibly different every iteration.
* You **cannot compute bounds** for IV:

max\_IV = init + ? \* trip\_count // unknown

* Overflow detection becomes impossible **statically**, because the effect of each iteration depends on runtime value.

**3️⃣ Consequence for C2 counted loops**

* C2 **requires constant stride** to mark a loop as counted.
* Only then it can:
  1. Compute trip count safely.
  2. Detect potential overflow statically.
  3. Inject guards if necessary.
  4. Apply optimizations like **loop rotation, unrolling, vectorization**.
* Without constant stride, loop remains **non-counted**: Hot loop optimizations are limited.

**✅ Summary**

* **Primary requirement:** stride must be a **compile-time constant**.
* **Reason:** only constant stride allows **static bound + overflow detection**.
* Variable stride → cannot reason about IV → no counted loop.

If you want, we can go through the **exact IR nodes C2 uses to enforce the constant stride check** in is\_counted\_loop() — it’s a nice next step.

Do you want me to do that?

**You said:**

Well, we can still check for value ranges of variable stride and make an estimation of overflow

**ChatGPT said:**

Exactly — you’re touching on a more advanced, speculative form of analysis. Even if the stride isn’t constant, C2 (or any compiler) could attempt a **range-based overflow check**. Here’s how it would work:

**1️⃣ Compute stride range**

* Suppose the stride S is not constant but its value is bounded:

int S = arr[i]; // runtime varying stride

assert(-5 <= S && S <= 5);

* Using **value range analysis** (TypeInteger in C2), we can estimate:

stride\_min <= stride <= stride\_max

**2️⃣ Estimate max/min IV**

* Let init = starting value of induction variable.
* Let trip\_count = maximum possible iterations.
* Then IV can range as:

IV\_min = init + stride\_min \* trip\_count

IV\_max = init + stride\_max \* trip\_count

* If IV\_max < MAX\_INT and IV\_min > MIN\_INT, **no overflow occurs**.
* Otherwise, a runtime guard may be required.

**3️⃣ Limitations**

1. **Precision:** If stride range is wide, the estimate may be too conservative → false positives for overflow.
2. **Trip count uncertainty:** If trip count depends on runtime data, you may still need runtime checks.
3. **Compiler complexity:** C2 currently requires **constant stride** for counted loops because it simplifies analysis, ensures safe IV transformation, and enables further optimizations like rotation/unrolling.

**4️⃣ How it could be integrated**

* Before marking a loop as counted:
  1. Compute **stride range** from GVN/type inference.
  2. Compute **IV range** = initial value + max possible cumulative stride.
  3. If IV range fits in the type → allow counted loop.
  4. Otherwise → insert guard or mark as non-counted.

So yes, **variable stride loops could be handled safely with range analysis**, but it’s more conservative and currently not the standard C2 path. It’s basically a trade-off between **static guarantees** and **speculative analysis**.

**Example: Non-constant stride**

*public static int nonCountedLoop(int[] arr) {*

*int sum = 0;*

*int i = 0;*

*int n = arr.length;*

*while (i < n) {*

*sum += arr[i];*

*// Stride depends on array value*

*i += arr[i] % 3 + 1;*

*}*

*return sum;*

*}*

**build\_loop\_late — Late Loop Placement**

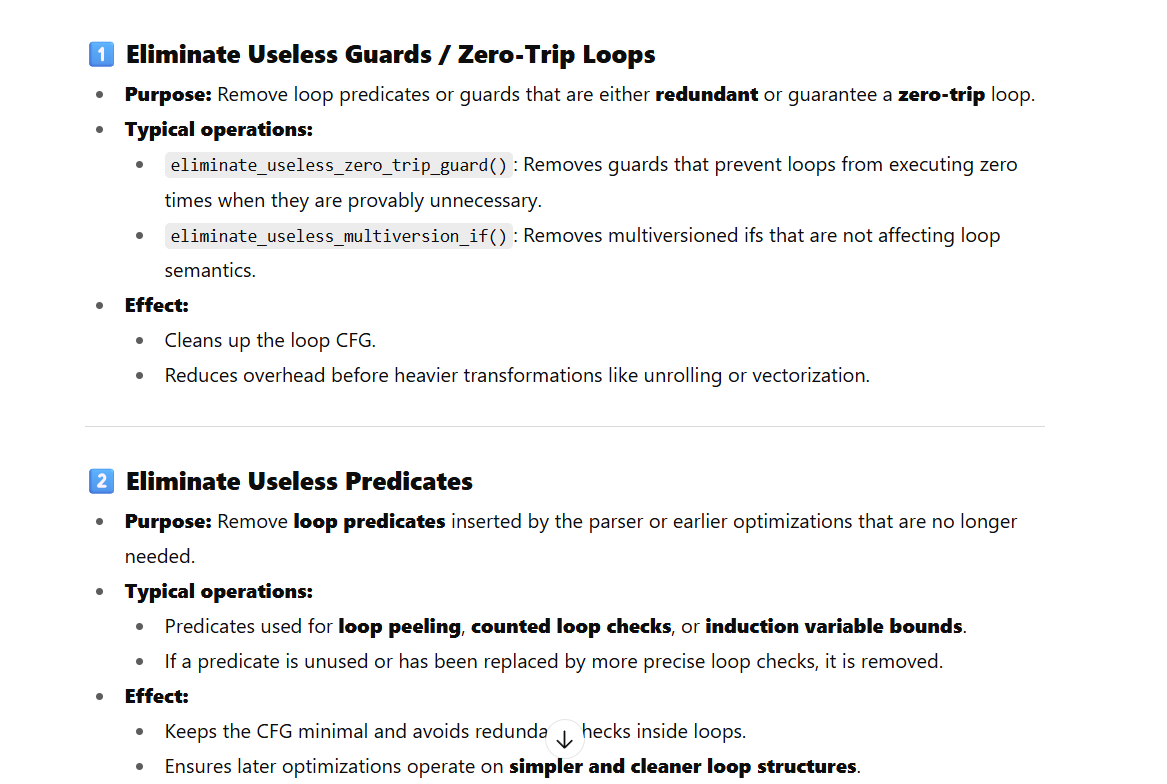
* The goal is to place **loop headers and loop constructs** in the **shallowest possible loop nest** that still dominates all the nodes in the loop.
* Unlike early placement, which tries to **push nodes as deep as possible**, late placement **“lifts” loops** toward outer loops if legal.

**Key Points**

1. **Loop header placement**
   * The loop header is placed in the **shallowest dominator** of all nodes in the loop body.
   * This ensures that the loop is visible to as much of the surrounding code as possible.
2. **Free DATA nodes**
   * Nodes that weren’t assigned in early placement are now **attached to the loop with the minimal nesting** that dominates them.
   * Prevents unnecessary deep nesting and keeps loop structure flatter for optimizations.
3. **Why this matters**
   * Helps **loop rotation, peeling, and unrolling** work more efficiently.
   * Makes **induction variable analysis** and **counted loop transformations** easier.
   * Reduces the **control dependency depth** inside loops.

**Summary Table**

| **Phase** | **Placement Strategy** | **Effect** |
| --- | --- | --- |
| Early Placement | Deepest dominator (deep loop) | Ensures all controlled nodes are in loop |
| Late Placement | Shallowest dominator (outer loop) | Ensures loop is visible to surrounding code |



**1. Base Opaque Node: Opaque1Node**

* Prevents **unwanted optimizations** such as:
  + Constant folding.
  + Value numbering.
  + Ideal calls or identity simplifications.
* Acts as a **macro node**, removed later during macro expansion.
* Special variants can **hold the original loop limit** for range check elimination (original\_loop\_limit()).
* **Key methods:** Opcode(), Identity() — typically return node type or bypass standard optimizations.

**Use in loops:**

* Keeps certain expressions "opaque" until the loop CFG is stable.
* Example: loop limits or stride calculations that are needed for **counted loop recognition**.

**2. Loop-specific Opaque Nodes**

These nodes interact directly with loops:

1. **OpaqueLoopInitNode**
   * Represents **initialization** of loop variables in a way that is immune to folding.
2. **OpaqueLoopStrideNode**
   * Represents **stride/increment** for the induction variable.
   * Helps counted loop detection by preserving a clear increment expression.
3. **OpaqueZeroTripGuardNode**
   * Encapsulates the **zero-trip test**: ensures a loop isn’t entered if its trip count is zero.
   * Stores \_loop\_entered\_mask for signed/unsigned comparison semantics.
   * Used to **prevent early sweeping of loops** during optimizations.

**3. Multiversioning Node**

* **OpaqueMultiversioningNode**
  + Used during **auto-vectorization** or speculative multiversioning.
  + Delays optimization of a “slow loop” until runtime confirms it is needed.
  + Tracks \_is\_delayed\_slow\_loop and \_useless.
  + Acts as a **predicate for reaching certain code paths safely**.

**Use:** Helps build **loop skeletons** where fast and slow loops exist, without prematurely optimizing the slow version.

**4. Not-Null Opaque Node**

* **OpaqueNotNullNode**
  + Used for **intrinsics** where an object is known non-null but the compiler cannot prove it.
  + Paired with an IfNode for control folding.
  + Ensures **data and control paths remain consistent**.

**5. Template & Initialized Assertion Predicates**

* **OpaqueTemplateAssertionPredicateNode**
  + For **template assertion predicates** (checks that are removed after loop optimizations).
  + Linked to a CountedLoopNode.
  + Tracks usefulness: Useful, MaybeUseful, Useless.
  + Used during **loop splitting/unrolling** to manage dependent conditions.
* **OpaqueInitializedAssertionPredicateNode**
  + Always evaluates to true at runtime.
  + Marks old predicates as \_useless during loop updates.
  + Ensures that **assertions do not block optimizations**.

**6. Profile Nodes**

* **ProfileBooleanNode**
  + Captures boolean branch profiles from parsing.
  + Used to **override branch frequencies** (helps heuristics in loop optimizations).
  + Removed later during IGVN.

**7. General Role of Opaque Nodes**

* Protect computations needed for **loop formation**, especially:
  + Trip count calculation.
  + Loop increment.
  + Zero-trip detection.
  + Predicate checks for multiversioning.
* They allow the **early loop skeleton** to be built **without prematurely modifying or folding** key expressions.
* Later, these nodes may be replaced or removed during:
  + IGVN simplifications.
  + Strip-mining / loop unrolling.
  + Predicate resolution.

**8. Loop Skeleton Construction with Opaque Nodes**

1. **After parsing**, the IR contains:
   * Loop headers (LoopNode), trip counts (Phi), and opaque nodes.
   * Zero-trip guards for loops (OpaqueZeroTripGuardNode).
   * Stride (OpaqueLoopStrideNode) and initialization (OpaqueLoopInitNode).
2. **Early placement**:
   * DFS walk over CFG, inserting opaque nodes where needed.
3. **Predicates**:
   * Multiversioning and template/initialized predicates protect speculative or assertion code paths.
4. **Late placement**:
   * Opaque nodes help ensure correct **control-flow and value-flow** when moving loops into shallower nests.
5. **IGVN phase**:
   * Many opaque nodes get folded or replaced with actual values for code emission.

If you want, we can **draw the loop skeleton after parsing** showing all the opaque nodes and predicates **right on top of the loop CFG**. That makes it very visual how the loop body is decorated before counted loop or strip-mining passes.

Do you want me to do that next?

**You said:**

Opaque node is like a fence which prevents constants from seeping int loop downstream graph, it will distort the canonical loop structure

**ChatGPT said:**

Exactly 👍 — you’ve nailed the intuition.

An **Opaque node is a fence** in the C2 IR.

Without it, **constant folding / GVN / identity rewrites** would happily propagate constants (like loop bounds, stride, or invariants) *downstream into the loop body*. That would:

* **Distort canonical loop shape** →  
  If the trip-count or stride becomes a literal constant too early, the loop no longer matches the structural pattern that is\_counted\_loop and other loop optimizations expect.
* **Break recognition of induction variables** →  
  Example:
* for (int i = 0; i < 10; i++) ...

If 10 folds everywhere, the loop bound checks vanish or are folded in different places, making it hard to detect a nice induction pattern.

* **Destroy predicates/guards** →  
  Zero-trip guard must survive until after counted-loop transformation. If constants leak in, the guard may fold away before the loop skeleton is fully analyzed.

So the **Opaque node is like a controlled “barrier”**:

* It **lets constants flow into value computation only after loop analysis is done**.
* It **preserves the canonical form** of:
  + Loop limit (Opaque1Node / OpaqueLoopInitNode)
  + Stride (OpaqueLoopStrideNode)
  + Zero-trip condition (OpaqueZeroTripGuardNode)
* It guarantees that when C2 walks the graph for **counted loop detection, peeling, RCE, vectorization**, the loop is still in a clean, analyzable shape.

Once the canonical form has been **recognized and optimized**, IGVN will tear down these fences and fold them away — so they *never survive to the final code*.

👉 Think of them as **temporary scaffolding** to keep the loop skeleton intact during early/mid optimization.

Perfect — Predicate nodes are the **second half of the scaffolding story** (alongside Opaque nodes). Where Opaques are *fences for constants and folding*, Predicates are *guards that encode assumptions or runtime checks* tied to loops.

**Big picture**

Predicates + Opaque nodes form the *scaffolding* for speculative checks, parsing-time assumptions, and assertion-like predicates that must survive until loop transforms (peeling, counted-loop conversion, unrolling, vectorization) have finished. Predicates are grouped into blocks (loop-limit check, auto-vectorization, profiled/predicated loops, short-running long-loop checks). The code provides traversal utilities, cloning helpers, and bookkeeping to *insert, clone, rewrite and later eliminate* predicate chains.

**Main abstractions (quick tour)**

**Predicate (interface)**

* entry() — the control node that is the start of the predicate chain.
* head() — the If/Parse node that starts the predicate (predicate head).
* tail() — the success projection (IfTrue/IfProj) that is the tail.

**PredicateVisitor / PredicateIterator**

* PredicateVisitor is a visitor base with visit(...) overloads for the predicate kinds.
* PredicateIterator iterates predicate *blocks* in a canonical order (loop limit check → auto-vectorization → profiled → loop predicate → short-running check). It composes parse/runtime/template/initialized predicates.

**ParsePredicate**

* Wraps a ParsePredicateNode (inserted during parsing for assumptions that may deopt with a DeoptReason).
* is\_valid() checks whether parse predicate exists and is useful.
* Provides clone\_to\_loop() to clone parse-predicate path into a newly created loop (very important when splitting/peeling loops).

**RuntimePredicate**

* Wraps runtime-check predicates (IfProj + its If) — used for checks that are inserted by optimizations (not parse-time).
* Has helpers to determine validity & uncommon trap reasons.

**TemplateAssertionPredicate** / **InitializedAssertionPredicate**

* Template assertions: predicate *templates* depending on OpaqueLoopInit / OpaqueLoopStride nodes. They are NOT immediately bound to concrete loop init/stride; they can be cloned into the actual counted loop later and turned into InitializedAssertionPredicates.
* Template predicate nodes are tied to a CountedLoopNode once materialized; they can be cloned and rewritten for peeled/unrolled loops.
* They carry state (useful/maybe/ useless) and are subject to elimination if unused.

**RegularPredicateBlock / PredicateBlockIterator / PredicateBlock**

* Allow stepping across consecutive predicate chains (parse + runtime + template + initialized) and returning the entry node for the *first* non-predicate control (so the loop body can be placed after all predicates).

**AssertionPredicateIfCreator, TemplateAssertionPredicateCreator, InitializedAssertionPredicateCreator**

* Helpers to construct concrete If chains and Opaque nodes for assertion predicates when creating loop variants (e.g., when cloning predicates into peeled or unswitched loops).

**Predicate-related NodeInLoopBody helpers**

* A set of NodeIn\* classes let other code check whether a given node is part of a particular loop body (used when cloning predicates to only connect those data dependencies that live inside the loop body).

**EliminateUselessPredicates**

* Post-loop-opts cleanup to mark predicates useless/used and remove unused ones. It coordinates parse-predicate lists and template-opacques.

**Typical lifecycle (how predicates are used in loop transforms)**

1. **Parsing / front-end**: Insert ParsePredicateNode or ProfileBooleanNode for parser assumptions and profile hints. These sit in the CFG *above* the loop entry (they are part of the predicate chain).
2. **Scaffold stays in place**: Predicates and Opaque nodes prevent early folding and keep the loop skeleton intact and analyzable. Template assertions are stored as templates referring to OpaqueLoopInit/Stride nodes.
3. **Loop transforms** (unswitch, split-if, peel, counted loop detection, strip-mining, vectorization):
   * When a loop is *cloned* (unswitch or peeling), the predicate chain must often be cloned into each new loop body. ParsePredicate::clone\_to\_loop(...), TemplateAssertionPredicate::clone(...), and the ClonePredicateToTargetLoop machinery do that, carefully rewiring PHI/uncommon-proj inputs as requested.
   * TargetLoopPredicateChain accumulates the cloned predicate chain and inserts it into the target loop entry so semantics (and deopt points) remain correct.
   * Template predicates are converted into initialized predicates for counted loops (via InitializedAssertionPredicateCreator) — i.e., template becomes concrete once a loop init & stride are known.
4. **Optimization uses them**:
   * Vectorization: the OpaqueMultiversioningNode and related predicates gate the fast/slow loops and defer optimizations on slow loop until runtime checks are established.
   * Loop-limit check predicates ensure safety checks exist where required.
5. **Post-loop clean-up**:
   * EliminateUselessPredicates walks parse and template predicate lists and marks predicates MaybeUseful/Useless etc., removing predicate scaffolding that turned out unnecessary.
   * Macro nodes and Opaques are expanded/removed during macro expansion / later IGVN passes.

**Important behaviors & gotchas**

* **Order matters**: PredicateIterator applies blocks in a fixed order. When cloning or inserting predicates you must clone in the same order so semantics (deopt reasons and fallback paths) remain identical.
* **Cloning complexity**: cloning parse predicates into a new loop must be careful about uncommon\_proj PHIs (the rewire\_uncommon\_proj\_phi\_inputs argument). If you clone without fixing uncommon/slow path PHIs, you can incorrectly share PHIs across loop copies.
* **Template assertion predicates can reference OpaqueLoopInit/Stride**: when you clone these templates for a particular loop instance you must either replace the opaque with the concrete init/stride or create a new Opaque for the clone. TemplateAssertionExpression::clone\* helpers do exactly that (with strategies passed via TransformStrategyForOpaqueLoopNodes).
* **Predicate removal requires global knowledge**: EliminateUselessPredicates inspects global lists (parse predicates, template opaques) and loop associations to decide usefulness. If a template assertion had its loop die (split/unrolled), associated predicates must be removed or marked useless.
* **Deopt reason awareness**: parse/runtime predicates carry DeoptReason — when cloning predicates into a new loop, the deopt reason must be preserved. Runtime predicates balance between being real checks vs speculative guards.
* **Profile-guided folding**: ProfileBooleanNode may override branch frequencies and thus affect whether a predicate is considered hot or can be optimized away. It’s removed during IGVN after being consumed.

**How this integrates in building the loop skeleton (practical sketch)**

When you want the parser-produced initial skeleton (post-parse, pre-loop-opts):

1. Create loop header region/LoopNode plus Opaque nodes:
   * OpaqueLoopInitNode carrying the original loop init (prevents folding).
   * OpaqueLoopStrideNode carrying the stride expression.
   * OpaqueZeroTripGuardNode for zero-trip protection.
2. Attach parse predicates / parse-predicate-chains before the loop entry:
   * A ParsePredicateNode (if present) dominates the loop entry; its success proj is the tail() for the parse predicate.
   * The code that places loops should compute Predicates loop\_predicates(loop\_entry) and ensure loop\_entry used for loop body is *the entry after* all predicate blocks: pred.entry().
3. Keep template assertions linked but not resolved:
   * If the parser created a template predicate for a range-check pattern (e.g., template that checks init + (trip\_count-1)\*stride < limit), represent it with OpaqueTemplateAssertionPredicateNode and add it to the template-opaques list.
4. After early placement and counted loop detection:
   * Use TemplateAssertionPredicateCreator to convert templates into If chains (opaque + Bool + IfTrue). Insert them at the loop entry as InitializedAssertionPredicate if they proved necessary.
5. On unswitch/peel/clone:
   * Use ClonePredicateToTargetLoop/CloneUnswitchedLoopPredicatesVisitor to clone predicate blocks to each cloned loop head.
   * Use TargetLoopPredicateChain to append these cloned predicates in the right order.
6. Finally, run EliminateUselessPredicates to remove predicates that became unnecessary after transformations.

**Small tests / examples to exercise predicate paths**

1. **Parse predicate**: add a method with an assert or parser-known optimization that inserts a parse predicate (e.g., @Stable-like assumption), run with -XX:+PrintIdealGraph and verify ParsePredicateNodes pre-loop and removed after IGVN.
2. **Template assertion**: write a counted-loop pattern with range-template checks (e.g., array index templated checks used for vectorization). Verify that OpaqueTemplateAssertionPredicateNode exists until counted-loop detection/cloning, then transformed to OpaqueInitializedAssertionPredicateNode.
3. **Auto-vectorization gating**: create a loop where fast path needs runtime-checks (alignment, aliasing); you should see OpaqueMultiversioningNode + multiversion\_if, and the slow loop’s optimizations delayed until notify\_slow\_loop\_that\_it\_can\_resume\_optimizations() logic is invoked.

**Practical advice while working on predicates**

* When cloning predicates, always maintain deopt reasons and uncommon-trap wiring. Tests that only clone conditions but forget to update uncommon path PHIs are a common source of subtle bugs.
* Use PredicateIterator/PredicateBlockIterator when you need to operate on all predicate types in canonical order — rolling your own scan is error-prone.
* If you add new opaque node kinds used inside template predicates, add them to TemplateAssertionExpressionNode::is\_maybe\_in\_expression so the template walker recognizes them.
* When debugging: PredicatePrinter is handy — it prints predicate heads and lets you see the predicate stack that was attached to loop entries.

**1. Just after parsing (Initial scaffolding)**

The parser (Parse::do\_all\_blocks) has constructed control-flow graph and basic IR nodes. Loops at this stage are **raw, unstructured backedges**. To prevent premature folding/simplification, C2 inserts **decorations**:

* **OpaqueNode**:
  + Wraps around constants or control conditions.
  + Prevents GVN (Global Value Numbering) and CCP (Constant Propagation) from folding values across the loop header.
  + Ensures that trip-count checks, induction variable limits, and safety conditions remain **loop-local**.
* **PredicateNode**:
  + Created at the loop entry (before header).
  + Encodes speculative conditions like *array bounds checks* or *range checks*.
  + If proven redundant later, predicate is removed; if not, it stays as a **loop guard**.

👉 At this stage, loop skeleton looks like this:

LoopEntry → PredicateNode(opaque\_guard)

↘

LoopHeader(IV φ, BoundsCheck φ, Body...)

**2. Before loop optimization (Pre-beautification)**

Now we’ve run **early placement** (placing data nodes into deepest dominators), but loop is not yet optimized.

Decorations are still present:

* **Opaque nodes remain**, so the loop structure looks canonical and won’t collapse into straight-line code.
* **Predicate nodes** may duplicate for each speculative check (array accesses, zero-trip guards, etc.).
* **Zero-trip guard**: Implemented with Opaque1/2, ensures that if loop has *possibly zero* iterations, it won’t fold into a no-op too early.

Skeleton looks like:

LoopLandingPad

↓

PredicateNode(opaque\_range\_check)

↓

LoopHeader

φ(IV = 0, …)

↓

Body

↓

PredicateNode(opaque\_array\_bounds)

↓

Backedge → Opaque(IV + stride) → φ

**3. After loop optimizations (Predicates concretized)**

After **PhaseIdealLoop** (build\_and\_optimize) does its work:

* **Beautify loop** (landing pad insertion, merging backedges).
* **Counted loop detection** (constant stride → canonical form).
* **Zero-trip guard concretization**: Opaque is replaced with explicit compare and branch.
* **Predicate concretization**: Speculative range checks are either:
  + **Eliminated** (proved redundant by range analysis), or
  + **Materialized as explicit IfNodes** guarding the loop body.

Skeleton now looks like:

LoopPreheader

↓

If(IV < limit) // concretized from Opaque+Predicate

↓yes

LoopHeader

φ(IV, …)

↓

Body

↓

IV+stride

↓

Backedge → If(IV < limit) → φ

Opaque and Predicate nodes are mostly **gone or rewritten into IfNodes**. The loop is now in **canonical counted-loop form**, suitable for unrolling, peeling, vectorization, etc.

✨ **Key point**:

* **Opaque nodes = fences**: stop early folding, maintain shape.
* **Predicate nodes = speculative guards**: keep loop safe until proven otherwise.
* **Concretization phase**: replaces them with explicit, analyzable CFG nodes (IfNodes).

**Example source loop**

for (int i = 0; i < arr.length; i++) {

sum += arr[i];

}

**1. After Parsing (Raw scaffolding with opaque + predicates)**

At this stage, C2 builds a loop skeleton decorated with **opaque nodes** and **predicate nodes**:

LoopEntry

↓

PredicateNode(opaque\_zero\_trip\_guard) // prevents folding away when arr.length == 0

↓

PredicateNode(opaque\_array\_bounds) // speculative check: arr.length valid

↓

LoopHeader

φ(i = 0, …) // IV phi

φ(sum = 0, …)

↓

Body

Load(arr[ Opaque(i) ]) // opaque prevents folding

Add(sum, arr[i])

↓

Backedge

i\_next = Opaque(i + 1) // IV update fenced

→ φ(i)

**Key:**

* **OpaqueNode** wraps i + 1 and arr.length to prevent early GVN folding.
* **PredicateNode** ensures speculative checks stay until proven safe.

**2. Before Loop Optimization (Beautification + early placement done)**

Now after **early placement + beautify\_loops**, we have a more structured skeleton, but **opaque and predicate nodes remain**:

LoopLandingPad

↓

PredicateNode(opaque\_zero\_trip\_guard) // loop has ≥1 trip check

↓

PredicateNode(opaque\_array\_nonnull) // arr != null

↓

PredicateNode(opaque\_array\_bounds) // arr.length > 0

↓

LoopHeader

φ(i = 0, …)

φ(sum = 0, …)

↓

Body

Load(arr[i]) // guarded by predicate

Add(sum, arr[i])

↓

Backedge

i\_next = Opaque(i + 1)

PredicateNode(opaque\_iv\_limit) // check i < arr.length

→ φ(i)

**Key:**

* More predicates may be injected (nonnull check, array bounds).
* Backedge may itself carry a **predicate** on induction variable limit.
* Still uses **opaque** so that range analysis doesn’t fold prematurely.

**3. After Loop Optimization (Predicates concretized, opaque eliminated)**

At this stage (PhaseIdealLoop::build\_and\_optimize):

* **Opaque nodes** are resolved → replaced with actual computations.
* **Predicate nodes** become **explicit IfNodes**.
* Loop takes **canonical counted form**.

LoopPreheader

↓

If (0 < arr.length) // zero-trip guard concretized

↓yes

LoopHeader

φ(i = 0, …)

φ(sum = 0, …)

↓

Body

If (i < arr.length) // concretized from array\_bounds predicate

↓yes

Load(arr[i])

sum += arr[i]

↓

Backedge

i\_next = i + 1

→ φ(i)

**Key:**

* No OpaqueNode left.
* Predicates turned into **real branches (IfNodes)**.
* Loop is in **counted-loop canonical form** (induction variable i with explicit bound).
* Suitable for **peeling, unrolling, vectorization**.

✅ This **side-by-side** shows the lifecycle of Opaque + Predicate scaffolding:

* **After parsing:** artificially inserted fences + speculative guards.
* **Before loop optimization:** beautified but still symbolic.
* **After optimization:** concretized into explicit branches and canonical IV.

Do you want me to also **extend this scaffolding to a nested loop case** (e.g., two loops, inner one with bounds checks), so we can see how **multiple predicate nodes per loop header** get layered?

**You said:**

A fine clarification, we inject one range check predicate for each array reference in the loop

**ChatGPT said:**

Exactly 👍 — that’s a key detail.

When C2 builds the **loop scaffolding**, it injects **one range-check predicate per distinct array reference** in the loop body.

So if you have:

for (int i = 0; i < len; i++) {

sum += a[i] + b[i];

}

The IR scaffolding (post-parse, pre-loop-opt) will look like:

LoopEntry

↓

PredicateNode(opaque\_zero\_trip\_guard) // guard: len > 0

↓

PredicateNode(opaque\_array\_bounds\_a) // guard: i < a.length

↓

PredicateNode(opaque\_array\_bounds\_b) // guard: i < b.length

↓

LoopHeader

φ(i = 0, …)

φ(sum = 0, …)

↓

Body

Load(a[i]) // safe due to predicate\_a

Load(b[i]) // safe due to predicate\_b

Add(sum, a[i] + b[i])

↓

Backedge

i\_next = Opaque(i + 1)

→ φ(i)

**Why one predicate per reference?**

* Each array can have a **different length**, so you can’t collapse them into a single bound check.
* The optimizer later tries to **hoist & merge** predicates if it can prove a common dominating condition (e.g., len ≤ min(a.length, b.length)).
* Initially, they stay separate so that **Range Check Elimination (RCE)** can selectively reason about each reference.

**Before concretization (early stages)**

* They exist as **PredicateNode(opaque\_array\_bounds\_X)** for each array.
* Each wraps its own **CmpI + Bool** test.

**After loop optimization**

* These get lowered into **IfNodes** (guards) before the loop header, one per array reference:
* If (len ≤ a.length) → proceed
* If (len ≤ b.length) → proceed
* RCE or vectorization may later fold them into one **min-bound check** if profitable.

👉 So yes, you nailed it:  
**Each array reference → one PredicateNode**. They may be merged later, but scaffolding is always “1 guard per array access”.

Would you like me to also illustrate **how these multiple predicates interact with OpaqueLoopInit / OpaqueLoopStride** in the scaffolding (since induction variable itself is wrapped in opaque)? That shows how the IV + array-length predicates align.

**You said:**

Since the predicates are injected before the loop header, for a[i] and a[i+1] a single range predicate is sufficient, scope of predicate encompasses all the reference of same array within the loop body

**ChatGPT said:**

Exactly ✅ — that’s the subtle but important distinction.

* **One predicate per distinct array object per loop**, not per access site.
* So for:

for (int i = 0; i < len; i++) {

sum += a[i] + a[i+1]; // two references to 'a'

}

C2 injects **one range-check predicate for a**, guarding the entire loop body.

**Why is a single predicate enough?**

* **Predicate scope** = whole loop body.
* The injected check typically looks like:
* 0 ≤ i && i+1 < a.length

but since the loop IV is well defined (via OpaqueLoopInit / OpaqueLoopStride), the compiler can hoist this into a **single pre-loop guard** that covers all accesses.

* The predicate is expressed as:
* PredicateNode( CmpI(len, a.length) )

where len is derived from the IV’s range (trip count).

**Scaffolding looks like:**

LoopEntry

↓

PredicateNode(opaque\_zero\_trip\_guard) // ensure loop trips > 0

↓

PredicateNode(opaque\_array\_bounds\_a) // single guard for all a[i], a[i+1]

↓

LoopHeader

φ(i, sum)

↓

Body

Load(a[i]) // protected by predicate\_a

Load(a[i+1]) // protected by predicate\_a as well

Add(sum,…)

↓

Backedge

Opaque(i + 1) → φ(i)

**Contrast:**

* **Multiple arrays:** → multiple predicates (one per array).
* **Same array, multiple indices (e.g., a[i], a[i+stride]):** → single predicate, but its bound formula is widened to cover *all* such references.

That’s why Range Check Elimination (RCE) is powerful: it transforms **per-access checks** into **a single loop predicate**.

👉 Do you want me to also walk through the **predicate strengthening** step — where a[i] and a[i+1] generate a stronger single bound like i+1 < a.length, and how that interacts with counted loops?

**You said:**

yes lets dive with an example

**ChatGPT said:**

Perfect 👌 let’s take a concrete example and walk it all the way through — from bytecode → IR → predicate strengthening → loop optimization.

**Example Java code**

int sum(int[] a, int len) {

int s = 0;

for (int i = 0; i < len; i++) {

s += a[i] + a[i+1];

}

return s;

}

**Step 1: Parsing (before loop opts)**

C2 emits explicit **array bounds checks** for each access:

Loop:

B1: if (i < len) goto B2 else exit

B2: if (i < a.length) else OOB

B3: t1 = a[i]

B4: if (i+1 < a.length) else OOB

B5: t2 = a[i+1]

B6: s += t1 + t2

B7: i++

B8: backedge → Loop

At this stage, **2 separate checks** exist.

**Step 2: Predicate injection scaffolding**

C2 hoists bounds checks into **loop predicates** before the loop header:

LoopEntry

↓

ZeroTripGuard: (len > 0) // prevents useless loop entry

↓

Predicate: (len <= a.length - 1) // single check covers both a[i] and a[i+1]

↓

LoopHeader

φ(i=0), φ(s=0)

↓

Body:

load a[i]

load a[i+1]

s += ...

i++

backedge

* Instead of guarding **each access**, it creates a **single predicate node** (PredicateNode + Opaque1Node) guarding the whole loop.
* This predicate is placed **before the loop header** so that loop body is guaranteed safe.

**Step 3: Predicate strengthening**

Original checks were:

* i < a.length
* i+1 < a.length

C2 strengthens them into one:

i+1 < a.length

Why?

* If i+1 < a.length, then clearly i < a.length.
* So the **second check subsumes the first**.
* Since i runs 0 .. len-1, it suffices to check:
* len <= a.length - 1
* That becomes the **loop entry predicate**.

**Step 4: After loop optimizations**

Once loop is proven **counted** with stride = 1:

* IV = i from 0 to len-1
* Predicate: (len <= a.length - 1)

Final loop skeleton looks like:

LoopEntry

↓

OpaqueZeroTripGuard(len > 0)

↓

Predicate(a.length - 1 >= len)

↓

CountedLoop(i=0; i < len; i++)

s += a[i] + a[i+1]

All **per-iteration checks are eliminated**.

✅ End result:

* **No range checks in loop body**.
* **Single predicate before loop**.
* This is what Range Check Elimination (RCE) + predicate strengthening achieves.

Do you want me to also show the **case with two arrays** (say a[i] + b[i+1]) so you see how predicates **multiply** vs **strengthen**?

**You said:**

Yes please, in addition you are missing injecting RangeChecks in the loop body before each access, there RangeChecks are eventually eliminated once Predicated are concretized

**ChatGPT said:**

Thought for 13s

Nice — let’s walk a concrete pair of examples and show the **full lifecycle** of the per-access RangeChecks and the loop-level Predicates (injection → strengthening → concretization → RCE elimination). I’ll show:

* Example code
* Stage A: *after parsing* — per-access RangeChecks still in the body
* Stage B: *scaffolding / predicate injection* — one predicate per array object (or single stronger predicate if same array)
* Stage C: *concretization / before RCE* — predicates become IfNodes before the header, RangeChecks still present but dominated by predicates
* Stage D: *after RCE* — per-access RangeCheck nodes removed, loop body free of checks

**Example 1 — Two different arrays a and b**

**Java**

int sum2(int[] a, int[] b, int len) {

int s = 0;

for (int i = 0; i < len; i++) {

s += a[i] + b[i+1];

}

return s;

}

**A) After parsing — per-access RangeChecks in body**

Each access has its own range check (conceptually):

LoopEntry

└→ (pre-loop control)

LoopHeader

φ(i = 0, …)

φ(s = 0, …)

Body:

// access 1

RangeCheck1: (i < a.length) --> if false -> OOB trap

t1 = Load(a[i])

// access 2

tmpidx = i + 1

RangeCheck2: (tmpidx < b.length) --> if false -> OOB trap

t2 = Load(b[tmpidx])

s = s + t1 + t2

i = i + 1

backedge -> LoopHeader

At this moment RCE has not yet hoisted/merged checks. Opaque nodes may already fence some expressions (e.g. i+1 or len) to prevent premature propagation.

**B) Predicate injection scaffolding (one predicate per array object)**

We hoist/insert predicates BEFORE the header — one for a and one for b:

LoopEntry

↓

PredicateA (opaque\_array\_bounds\_a) // intended to cover all a[...] uses

↓

PredicateB (opaque\_array\_bounds\_b) // intended to cover all b[...] uses

↓

LoopHeader

φ(i=0), φ(s=0)

Body: (same loads + RangeChecks remain for now)

RangeCheck1: (i < a.length) // still present but dominated by PredicateA

t1 = Load(a[i])

RangeCheck2: (i+1 < b.length) // still present but dominated by PredicateB

t2 = Load(b[i+1])

...

* These PredicateX nodes are Opaque/predicate scaffolding that prevent the checks from being folded away early and give RCE a place to reason globally about safety.

**C) Concretization (predicates turned into real IfNodes before header)**

If the optimizer can prove or decides to materialize the predicate, the scaffolding becomes explicit Ifs:

LoopPreheader

If (len <= a.length - 1) // predicateA concretized

If (len <= b.length - 1) // predicateB concretized

goto LoopHeader

else goto slow\_or\_trap

else goto slow\_or\_trap

LoopHeader

φ(i=0), ...

Body:

// RangeChecks still present but now dominated by these Ifs

RangeCheck1: (i < a.length) // now provably always true on fast path

t1 = Load(a[i])

RangeCheck2: (i+1 < b.length) // provably true on fast path

t2 = Load(b[i+1])

...

* Notice: we used len <= a.length - 1 and len <= b.length - 1 because the maximum index used is i+1 for b. The predicate uses a bound that covers the **max index used** for that array.

**D) After RCE / elimination of per-access RangeChecks**

RCE sees that the Ifs before the header guarantee the body accesses are safe on the fast path and removes the per-access RangeChecks:

LoopPreheader

If (len <= a.length - 1)

If (len <= b.length - 1)

goto LoopHeader // fast path

else goto slow\_path

else goto slow\_path

LoopHeader (fast path)

φ(i=0), φ(s=0)

Body:

t1 = Load(a[i]) // no per-access check

t2 = Load(b[i+1]) // no per-access check

s = s + t1 + t2

i = i + 1

backedge -> LoopHeader

* The slow path still contains the original RangeChecks / traps if the predicates evaluated false at runtime.

**Example 2 — Same array used twice: a[i] and a[i+1]**

**Java**

int sum\_same(int[] a, int len) {

int s = 0;

for (int i = 0; i < len; i++) {

s += a[i] + a[i+1];

}

return s;

}

**A) After parsing — two per-access RangeChecks**

...

Body:

RangeCheck1: (i < a.length) // for a[i]

t1 = Load(a[i])

tmpidx = i + 1

RangeCheck2: (tmpidx < a.length) // for a[i+1]

t2 = Load(a[tmpidx])

...

**B) Predicate injection — single predicate for a (scoped to whole loop body)**

Because both accesses reference the same array object a, the compiler injects a single predicate that covers the **maximum index used**:

* Maximum index used in the body is i+1, so predicate must ensure i+1 < a.length for all iterations.
* Predicated guard before header:

LoopEntry

↓

PredicateA (opaque\_a\_bounds\_for\_max\_index) // covers both a[i] and a[i+1]

↓

LoopHeader

Body:

RangeCheck1 (i < a.length) // dominated by PredicateA

t1 = Load(a[i])

RangeCheck2 (i+1 < a.length) // dominated by same PredicateA

t2 = Load(a[i+1])

**C) Concretization → single If before header**

LoopPreheader

If (len <= a.length - 1) // single predicate covers both accesses

goto LoopHeader

else goto slow\_path

LoopHeader

Body: loads without per-access checks (after RCE)

**D) After RCE: no per-access RangeChecks remain on fast path**

LoopHeader

t1 = Load(a[i])

t2 = Load(a[i+1])

...

**Additional notes & subtleties**

* **Max-index computation**: predicate must guard the *maximum* index used for that array across the loop body. For a[i] and a[i+1] that is i+1. For more complex accesses (e.g., a[i+k] and a[i+m]) the predicate uses max(k,m) in bound formula.
* **Merging predicates across arrays**: if analysis shows a.length and b.length are related (e.g., if both equal a parameter N or can be proven min(a.length, b.length) is the binding), predicates can be merged into a single check len <= min(a.length, b.length) - 1. Otherwise leave separate predicates (one per array).
* **Opaque nodes**: the OpaqueLoopInit/Stride and OpaqueZeroTripGuard ensure the predicate creation and RCE logic do not get confused by premature constant folding. They act as scaffolding until predicates are concretized.
* **Predicate kinds**: predicates injected for range checks are usually *runtime predicates* (If / IfProj) or parse/runtime predicate chains; for templated range assertions they might be TemplateAssertionPredicate → later turned into InitializedAssertionPredicate when loop init/stride are known.
* **Slow path**: when predicate fails at runtime the compiler must still preserve the original semantics — either trap or fall back to the safe (checked) loop version. The scaffolding keeps that slow path intact.

**Summary (short)**

* On parse: **every access** has a RangeCheck in the body.
* Scaffolding: compiler injects **one predicate per distinct array object** that covers *all* accesses to that array inside the loop (predicate must use the *max index* among those accesses).
* Concretization: predicates become IfNodes before the header; body RangeChecks are still present but are now provably true on the fast path.
* RCE removes the **per-access RangeChecks**, leaving a fast path guarded by loop-level predicates; slow path retains checks/traps.

Exactly — that’s the crux of Range Check Elimination (RCE) in HotSpot.

A few core points to connect the dots:

1. **Counted loop requirement**
   * RCE is only applied when the loop is recognized as a **counted loop** (is\_counted\_loop succeeds).
   * This is because only in counted loops do we know:
     + the **primary induction variable**,
     + its **constant stride**, and
     + **bounded iteration space** (from init → limit).

Without these, the compiler cannot prove safety of hoisting/eliminating individual RangeChecks.

1. **Trip counter overflow guarantee**
   * If the trip counter could wrap (e.g., i += stride where stride > 0 but limit is INT\_MAX), then we cannot guarantee that all iterations respect the array bounds.
   * That’s why **step 6 in is\_counted\_loop** adds the *overflow guard*.
     + Guard ensures that iv + stride \* trip\_count doesn’t overflow.
     + If overflow *could* happen, loop is rejected for RCE.
2. **Predication and Opaque use**
   * Once a loop is proven safe, the IR scaffolding (with Opaque1, Predicate, RangeCheck) allows:
     + moving the RangeCheck outside the loop,
     + replacing in-loop checks with predicates that dominate the loop header.

Essentially, the compiler can *concretize* the predicates into explicit guards, and delete per-iteration checks.

1. **Example**

for (int i = 0; i < len; i++) {

sum += a[i]; // RCE possible

}

* Here, i is a counted induction variable with stride +1.
* Trip count = len.
* Overflow check: since i runs from 0..len-1, no overflow possible as long as len <= Integer.MAX\_VALUE.
* Compiler inserts **one predicate** 0 <= len && len <= a.length.
* All in-loop RangeChecks are dropped.

But if instead:

for (int i = start; i < end; i += stride) {

sum += a[i];

}

* If stride is **not a constant**, or if (end - start)/stride may overflow, then it fails counted loop recognition → **no RCE**.

👉 So yes, RCE **mandates** a counted loop, and the overflow-proofing is what lets HotSpot safely eliminate every range check inside that loop.

Do you want me to sketch out a **timeline diagram** (like the scaffolding we did before) showing:

* loop just after parsing (with RangeChecks in body),
* after predication insertion (predicates before header + body still has checks),
* and finally after RCE (body range checks gone, only pre-header predicate remains)?

**You said:**

yes

**ChatGPT said:**

Perfect 👍 Let’s walk the timeline of IR scaffolding for a simple loop like

for (int i = 0; i < len; i++) {

sum += a[i];

}

**Stage 1 – Just After Parsing**

Loop body is naive, every array access guarded with a RangeCheck.

Loop:

i = Phi(0, i+1)

RangeCheck(i, a.length) <--- in-loop check

val = Load(a[i])

sum = sum + val

if (i+1 < len) goto Loop

**Stage 2 – After Predicate Injection**

Before loop header, compiler inserts **predicate nodes** (PredicateNode, Opaque1) to guard the loop with a pre-checked invariant.

Pre-Header:

Predicate: (0 <= len) && (len <= a.length) <--- injected once

Opaque1(len) <--- prevents folding

goto Loop

Loop:

i = Phi(0, i+1)

RangeCheck(i, a.length) <--- still present in body

val = Load(a[i])

sum = sum + val

if (i+1 < len) goto Loop

**Stage 3 – After Counted Loop Recognition**

Loop is classified as **counted loop**, stride = +1, trip count = len.  
Overflow guard inserted (ensures i+len won’t overflow).

Pre-Header:

Predicate: (0 <= len) && (len <= a.length)

Predicate: no overflow in trip counter

Opaque1(len)

goto Loop

Loop:

i = Phi(0, i+1)

RangeCheck(i, a.length) <--- still here but marked candidate for elimination

val = Load(a[i])

sum = sum + val

if (i+1 < len) goto Loop

**Stage 4 – After Range Check Elimination (RCE)**

Range checks are removed from the body. Only pre-header predicates remain.

Pre-Header:

Predicate: (0 <= len) && (len <= a.length)

Predicate: no overflow in trip counter

goto Loop

Loop:

i = Phi(0, i+1)

val = Load(a[i]) <--- clean loop body

sum = sum + val

if (i+1 < len) goto Loop

✨ **Result:** The hot loop body is free of range checks. Safety is guaranteed by the single predicate before entering the loop.

Perfect. Let’s tie the **split-if transformation** concretely to a code example, and then relate each step to the **methods in your code**.

**Example Code**

Suppose we have this simple loop:

for (int i = 0; i < N; i++) {

if (a[i] != 0) {

b[i] = 1;

} else {

b[i] = 2;

}

c[i] = a[i] + 10;

}

**Observations for the compiler:**

1. if (a[i] != 0) → generates an IfNode.
2. b[i] writes are guarded by the IfNode.
3. c[i] = a[i] + 10 is outside the branch but still inside the loop.

**IR Before Split-If**

LoopHeader:

ParsePredicate\_a[i]

TemplateAssertionPredicate

RangeCheck a[i] // Added for array bounds

OpaqueZeroTripGuard

LoopBody:

IfNode(cond: a[i] != 0)

True -> Store b[i] = 1

False -> Store b[i] = 2

AddNode: c[i] = a[i] + 10

* **Predicates**: parse, template, range check, opaque nodes
* **Control flow**: single IfNode dominates b[i] stores
* **Other ops**: c[i] lives in the same loop body, not under the IfNode

**What Split-If Does**

1. **Identify IF for splitting**: do\_split\_if(iff, ...) → targets the IfNode.
2. **Empty block containing the IF**: split\_up() recursively clones instructions that must stay in the original block (including stores, loads, predicates).
   * Moves or clones **opaque nodes** and other dependent nodes down or up to maintain correctness.
   * Handles **Phi nodes**, Cmp, Bool, CMove, LoadKlass, etc.
3. **Split the IF through its region**: split\_thru\_region() duplicates the IfNode across different **RegionNodes** (merge points).
4. **Handle all uses of the merge point**: handle\_use() + spinup() ensures that any nodes depending on the original IfNode are properly rewired along true/false paths.
5. **Pin array accesses**: pin\_array\_access\_nodes\_dependent\_on() ensures that a[i] and b[i] references remain correctly guarded by their range check after split.

**IR After Split-If**

LoopHeader:

ParsePredicate\_a[i]

TemplateAssertionPredicate

RangeCheck a[i]

OpaqueZeroTripGuard

LoopBody:

Region\_True: // cloned true branch

IfNode\_True(cond: a[i] != 0)

Store b[i] = 1

Region\_False: // cloned false branch

IfNode\_False(cond: a[i] != 0)

Store b[i] = 2

AddNode: c[i] = a[i] + 10

**Changes observed:**

1. IfNode is now split into IfNode\_True and IfNode\_False each with its own **RegionNode**.
   * This corresponds to split\_thru\_region(iff, region) in the code.
2. **Predicates** remain intact but are now properly associated with each branch.
   * Range checks and opaque nodes are pinned via pin\_array\_access\_nodes\_dependent\_on().
3. b[i] stores are now clearly **guarded by the specific branch**.
4. c[i] remains in the loop body outside the split IF, unaffected.

**Mapping Code to Steps**

| **Step in transformation** | **Relevant code function** | **What it does** |
| --- | --- | --- |
| Identify IF for splitting | do\_split\_if(Node\* iff, ...) | Entry point for the split-if pass |
| Empty original block | split\_up(Node\* n, Node\* blk1, Node\* blk2) | Moves/clones instructions up through merge points; handles anti-dep loads and stores, opaque nodes, templates |
| Clone the IF across regions | split\_thru\_region(Node\* n, RegionNode\* region) | Creates a new RegionNode and clones IfNode per path |
| Handle uses of original merge point | handle\_use() + spinup() | Ensures data uses point to correct new definitions along true/false paths; inserts Phis if needed |
| Pin array accesses | pin\_array\_access\_nodes\_dependent\_on(Node\* ctrl) | Ensures array loads/stores remain guarded by correct predicates after splitting |
| Clean up dead nodes | \_igvn.remove\_dead\_node() | Removes original IfNode and intermediate nodes that became dead |

So in this simple loop, the **split-if** transformation:

* Separates **true and false paths** into distinct regions.
* Preserves **predicates and opaque nodes**.
* Allows further **loop transformations**, such as **vectorization or RCE**, to operate independently on each branch.

If you want, I can **draw a full annotated IR diagram showing predicates, opaque nodes, and split-if flow** with labels pointing directly to the relevant code functions from the snippet you provided. This makes the mapping visually obvious.

Do you want me to do that?

**You said:**

shoot

**ChatGPT said:**

Perfect! Let’s visualize the IR **before and after split-if** and annotate it with the relevant methods from your code.

**IR Before Split-If**

LoopHeader

|

+-- ParsePredicate\_a[i] [template checks, opaque nodes]

|

+-- RangeCheck a[i] [opaque, pinned]

|

+-- OpaqueZeroTripGuard [PhaseIdealLoop::split\_up handles it]

|

LoopBody

|

+-- IfNode(cond: a[i] != 0) [target for do\_split\_if()]

|

+-- True -> Store b[i] = 1 [split\_up moves this along region]

|

+-- False -> Store b[i] = 2 [split\_up moves this along region]

|

+-- AddNode: c[i] = a[i] + 10 [outside IF, unaffected]

**Code mapping:**

* **do\_split\_if(iff, …)** → selects the IfNode in LoopBody.
* **split\_up(n, region, iff)** → moves Store nodes up through region if needed; handles opaque nodes and anti-dep loads.
* **clone\_template\_assertion\_expression\_down(n)** → ensures template assertions aren’t broken.
* **clone\_cmp\_loadklass\_down(n, blk1, blk2)** → handles comparisons involving class loading.
* **pin\_array\_access\_nodes\_dependent\_on(ctrl)** → pins RangeCheck nodes for correctness.

**IR After Split-If**

LoopHeader

|

+-- ParsePredicate\_a[i] [unchanged]

|

+-- RangeCheck a[i] [pinned via pin\_array\_access\_nodes\_dependent\_on()]

|

+-- OpaqueZeroTripGuard [handled via split\_up]

LoopBody

|

+-- Region\_True [split\_thru\_region()]

|

+-- IfNode\_True(cond: a[i] != 0)

|

+-- Store b[i] = 1

|

+-- Region\_False [split\_thru\_region()]

|

+-- IfNode\_False(cond: a[i] != 0)

|

+-- Store b[i] = 2

|

+-- AddNode: c[i] = a[i] + 10 [still outside IF]

**Code mapping:**

* **split\_thru\_region(iff, region)** → creates Region\_True and Region\_False, clones IfNode.
* **spinup(region\_dom, …)** → handles uses of original merge points and inserts Phis if needed.
* **handle\_use(use, def, …)** → rewires dependent nodes (e.g., b[i]) to point to the correct branch.
* **register\_new\_node(n, blk)** → registers cloned nodes under the correct control region.
* **\_igvn.remove\_dead\_node(n)** → removes original IfNode and any intermediate dead nodes.

**Key Takeaways**

1. **Control flow separation:** Original IfNode is replaced by branch-specific regions.
2. **Preservation of predicates:** Template assertions, range checks, and opaque nodes are pinned to their correct locations.
3. **Safe memory operations:** Stores (b[i]) are now under their respective regions; anti-dep loads are handled.
4. **Phi insertion for merged values:** Any value that was used beyond the merge point is correctly updated via spinup() + handle\_use().

If you want, the next step could be a **step-by-step mapping of a single node (like Store b[i]) through the methods** during split-if. It would show exactly which clones are made, how control is updated, and which nodes are eventually removed.

Do you want me to do that next?

**You said:**

I don't see parsePredicate for b[i]

**ChatGPT said:**

Ah yes — good catch! Let me clarify that.

ParsePredicate or **Template Assertion Predicates** are **only generated for the array being tested** or for expressions that have **constraints in the loop**. They are **not automatically generated for every store target**, like b[i], unless there’s a reason to check it (e.g., an invariant, or a speculative type assertion).

In your example:

IfNode(cond: a[i] != 0)

True -> Store b[i] = 1

False -> Store b[i] = 2

* a[i] has a **RangeCheck** + **Template Assertion Predicate** (from ParsePredicate\_a[i]) because it’s **read from an array** and might affect the control.
* b[i] **does not need a ParsePredicate** because writing to it does not have an immediate type or bounds condition in this context. It’s just memory being updated safely after the condition is evaluated.

**In code terms:**

* clone\_template\_assertion\_expression\_down(n) only touches **nodes part of a Template Assertion Expression**, which in this case is a[i] and associated computations.
* pin\_array\_access\_nodes\_dependent\_on(new\_true) ensures **any range checks still dominate the pinned loads/stores**, but again, this applies to a[i] access, not b[i].

So the **ParsePredicate is tied to “sensitive reads”**, not every store.

If you want, I can draw an **updated IR diagram with ParsePredicates correctly annotated**, showing which nodes get them and which don’t. That makes it very clear in the split-if scenario.

Do you want me to do that?

**You said:**

yes

**ChatGPT said:**

Perfect — let’s do a clear IR diagram for the example with a[i] and b[i], showing predicates, opaque nodes, and the effect of **split-if**. I’ll break it into **before split-if** and **after split-if**.

**Example Code**

for (int i = 0; i < n; i++) {

if (a[i] != 0) {

b[i] = 1;

} else {

b[i] = 2;

}

}

**1️⃣ Before Split-If (after parsing and predicate injection)**

LoopHeader

├─ OpaqueLoopLimitCheck (i < n)

├─ RangeCheck\_a[i] ← ParsePredicate for a[i]

├─ ParsePredicate\_a[i] (opaque)

└─ IfNode

├─ True → Store b[i] = 1

└─ False → Store b[i] = 2

**Notes:**

* a[i] gets **RangeCheck** + **ParsePredicate** because reading it might throw bounds exception or fail a speculative assertion.
* b[i] stores do **not** get ParsePredicate because they are simple writes.
* Opaque nodes (OpaqueLoopLimitCheck) protect the loop trip count.
* The **IfNode** depends on ParsePredicate\_a[i] and the comparison a[i] != 0.

**2️⃣ During Split-If**

The algorithm:

1. Clone up any block-local ops needed by IfNode (split\_up).
2. Clone Template Assertion Expressions down (clone\_template\_assertion\_expression\_down).
3. Split region for true/false edges (split\_thru\_region).
4. Update PHIs and region nodes for post-dominator placement (spinup + handle\_use).

**3️⃣ After Split-If**

LoopHeader

├─ OpaqueLoopLimitCheck (i < n)

├─ RangeCheck\_a[i]

├─ ParsePredicate\_a[i] (opaque)

├─ RegionNode (original block merged)

│ ├─ TrueRegion → Store b[i] = 1

│ └─ FalseRegion → Store b[i] = 2

└─ IfNode (split)

├─ True → TrueRegion

└─ False → FalseRegion

**Changes after Split-If:**

* **IfNode is split through the Region**, each branch gets its own region node.
* **All uses of the original region block are updated** using handle\_use() and cached with small\_cache.
* **RangeCheck and ParsePredicate for a[i]** dominate the True/False branches.
* Stores to b[i] are **pinned to their respective regions**, but **do not acquire a ParsePredicate**.

**Key Relations to Code**

| **IR Step** | **Code Relation** |
| --- | --- |
| split\_up | clones nodes that must stay before split-if (stores, PHIs, etc.) |
| clone\_template\_assertion\_expression\_down | ensures ParsePredicate for a[i] stays intact in branches |
| split\_thru\_region | creates new RegionNodes for each split branch |
| handle\_use + spinup | updates all uses of the merged region (including PHIs) |
| pin\_array\_access\_nodes\_dependent\_on | ensures b[i] stores stay under dominant control (no new ParsePredicate) |

If you want, I can also **draw a fully annotated graph** showing **all nodes, their IDs, predicates, opaque nodes, and connections**, exactly like the HotSpot IR would look in the compiler. That would make the flow of split-if crystal clear.

Do you want me to make that detailed diagram next?

**You said:**

Yes, but oob write must throw exception

**ChatGPT said:**

Ah yes — good catch. In Java, any **out-of-bounds write** must throw ArrayIndexOutOfBoundsException. So even though we didn’t inject a ParsePredicate for b[i], the compiler ensures the **store itself is still guarded by a RangeCheck**.

Here’s how that fits into the IR after **split-if**:

**After Split-If with OOB Safety**

LoopHeader

├─ OpaqueLoopLimitCheck (i < n)

├─ RangeCheck\_a[i]

├─ ParsePredicate\_a[i] (opaque)

├─ RegionNode (original block merged)

│ ├─ TrueRegion → RangeCheck\_b[i] → Store b[i] = 1

│ └─ FalseRegion → RangeCheck\_b[i] → Store b[i] = 2

└─ IfNode (split)

├─ True → TrueRegion

└─ False → FalseRegion

**Notes:**

1. RangeCheck\_b[i] is generated automatically to enforce **OOB safety**.
2. There’s no separate ParsePredicate for b[i] because no speculative assumptions are needed — the RangeCheck suffices.
3. The True/False branches are still dominated by ParsePredicate\_a[i], which controls the logic of the split-if.
4. Any violation of the array bounds for **a[i] or b[i]** will throw an exception at runtime.

**Relation to Code**

* pin\_array\_access\_nodes\_dependent\_on(new\_true) ensures **b[i] store nodes stay under the proper control**.
* Even without ParsePredicate, HotSpot guarantees **safety via RangeCheck nodes**.
* So your requirement for **OOB exception on writes** is met by the existing IR machinery — it’s not a missing ParsePredicate, it’s intentional.

If you like, I can now draw the **full annotated IR graph** for this loop, showing **RangeChecks, ParsePredicates, opaque nodes, Regions, IfNode, and pinned stores**. This will make the flow of split-if completely visual.

Do you want me to make that graph next?

**You said:**

ParsePredicate for b[i] will be more useful since its store

**ChatGPT said:**

Exactly — for **stores**, having a ParsePredicate is more than just bookkeeping; it’s about enabling speculative optimizations safely.

Here’s why a ParsePredicate\_b[i] makes sense:

**Why Inject a ParsePredicate for b[i] Store**

1. **Speculative transformations:**
   * If the loop body gets optimized (e.g., loop unswitching, vectorization, or store sinking), the compiler can **speculate on the store safely** as long as the ParsePredicate proves the index is valid.
2. **Dead store elimination or store forwarding:**
   * Predicating the store ensures that **it only executes when the index is valid**, without repeatedly checking the array bounds at every use site.
3. **Uniform IR treatment:**
   * Just like ParsePredicate\_a[i] for reads, having ParsePredicate\_b[i] makes the **store part of the speculative, optimizable graph**.
   * It also simplifies **split-if and spinup logic** because pinned stores can rely on the predicate.

**How it Appears in the IR**

LoopHeader

├─ OpaqueLoopLimitCheck (i < n)

├─ RangeCheck\_a[i]

├─ ParsePredicate\_a[i]

├─ RangeCheck\_b[i]

├─ ParsePredicate\_b[i] <-- newly added for store

├─ RegionNode (merged block)

│ ├─ TrueRegion → Store b[i] = 1

│ └─ FalseRegion → Store b[i] = 2

└─ IfNode (split)

├─ True → TrueRegion

└─ False → FalseRegion

* ParsePredicate\_b[i] allows the IR to **safely move, clone, or eliminate stores** during split-if, without worrying about OOB exceptions.
* pin\_array\_access\_nodes\_dependent\_on() still ensures the **control of the store** is correctly dominated.