Loop jamming and eliminating induction variable ?

Loop Jamming

Loop jamming, also known as **loop fusion**, is a compiler optimization technique that combines two adjacent loops that iterate over the same index range into a single loop. This can improve performance by reducing loop overhead, enhancing data locality, and lowering instruction dispatch costs.

```
/ Original code with two separate loops
for (int i = 0; i < n; ++i) {
    A[i] = B[i] + C[i];
}
for (int i = 0; i < n; ++i) {
    D[i] = A[i] * 2;
}
// Fused loop
for (int i = 0; i < n; ++i) {
    A[i] = B[i] + C[i];
    D[i] = A[i] * 2;
}
```

Eliminating Induction Variables

An *induction variable* is a variable in a loop that gets incremented or decremented by a fixed amount each iteration. Eliminating redundant induction variables can reduce unnecessary arithmetic operations inside the loop, thereby speeding up execution.

> **Definition:**
> An induction variable `j` whose value can be expressed as a linear function of another induction variable `i` may be eliminated.

Consider the following code:

```
for (int i = 0; i < n; ++i) {
    int j = 3*i + 5;
    // Use j in the loop body
    X[j] = ...;
}
```

Here, `j` is a *secondary induction variable* since it depends linearly on `i`. We can eliminate `j` by introducing a new variable `t` (initialized before the loop) and updating it within the loop:

```
int t = 5;          // t = 3*0 + 5
for (int i = 0; i < n; ++i) {
    // Use t in place of j
    X[t] = ...;
    t += 3;         // Update for next iteration: t = 3*(i+1) + 5
}
```

- Initialization: $t_0 = 3*0+5 = 5$ t_0 = 3 * 0 + 5 = 5
- Update per iteration: $t_{k+1} = t_k + 3$ t_{k+1} = t_k + 3

Now the multiplication `3*i` is replaced by an addition `t += 3`, which is typically cheaper on most architectures.

Key Benefits of Induction Variable Elimination

- **Reduced Arithmetic Overhead:** Multiplications or divisions per iteration are replaced by simpler additions.
- **Improved Register Allocation:** Fewer variables may allow the compiler to keep values in registers.
- **Enhanced Loop Performance:** Especially in tight loops, eliminating heavy operations can lead to significant speedups.

Loop Optimisation?

Loop Optimization

Loop optimization encompasses various compiler transformations to make loops more efficient by reducing overhead, improving data locality, and exposing parallelism. Below are key techniques, each illustrated with a brief example.

- **Loop Fusion (Jamming):** *(See earlier Loop Jamming section for detailed example.)*
- **Loop Unrolling:** *(See earlier Loop Unrolling section for detailed example.)*
- **Loop Interchange:** Swaps the order of nested loops to traverse multi-dimensional arrays in a cache-friendly manner.

```
// Original (inefficient for row-major array)
for (int j = 0; j < M; ++j) {
    for (int i = 0; i < N; ++i) {
        sum += A[i][j];
    }
}
// After interchange (cache-friendly)
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
        sum += A[i][j];
    }
}
```

- **Loop Tiling (Blocking):** Breaks a large iteration space into smaller blocks that fit in cache, reducing cache misses.

```
for (int ii = 0; ii < N; ii += T) {
    for (int jj = 0; jj < M; jj += T) {
        for (int i = ii; i < min(ii+T, N); ++i) {
            for (int j = jj; j < min(jj+T, M); ++j) {
                C[i][j] = A[i][j] + B[i][j];
            }
        }
    }
}
tile size=T
```

- **Induction Variable Elimination:**
  SEE BELOW SECTION

Loop Unrolling?

Loop Unrolling

Loop unrolling is a simple optimization that replicates the loop body multiple times to reduce the loop overhead (branching and index updates) and increase instruction-level parallelism.

> **Example:**

```
 // Original loop
> for (int i = 0; i < n; ++i) {
>     sum += A[i];
> }
> // Unrolled by factor of 4
> for (int i = 0; i + 3 < n; i += 4) {
>     sum += A[i] + A[i+1] + A[i+2] + A[i+3];
> }
> for (; i < n; ++i) {
>     sum += A[i];
> }

 Here, the loop overhead is reduced by roughly a factor of 4, and the compiler
may schedule the additions in parallel.
```

Code Motion ?

Code motion, also called loop-invariant code motion, moves computations that yield the same result on every iteration out of the loop. This reduces redundant work inside the loop and can significantly improve performance.

```
// Before code motion
for (int i = 0; i < n; ++i) {
    int t = a + b;        // loop-invariant computation
    X[i] = t * C[i];
}

// After code motion
int t = a + b;            // moved outside the loop
for (int i = 0; i < n; ++i) {
```

```
    X[i] = t * C[i];
}
```

Here, the addition `a + b` is computed once, rather than `n` times, reducing the loop's work by one addition per iteration.

Elimination Of Common Subexpression?

Common subexpression elimination (CSE) identifies and removes expressions that are computed multiple times, replacing them with a single variable to avoid redundant computation.

```
// Original code with repeated expression
for (int i = 0; i < n; ++i) {
    int t1 = A[i] * B[i] + C[i];
    D[i] = (A[i] * B[i]) - E[i];
    F[i] = (A[i] * B[i]) + G[i];
}

// After common subexpression elimination
for (int i = 0; i < n; ++i) {
    int tmp = A[i] * B[i];   // compute once
    int t1 = tmp + C[i];
    D[i] = tmp - E[i];
    F[i] = tmp + G[i];
}
```

The multiplication `A[i] * B[i]` is computed only once per iteration, reducing the number of multiplications from three to one.

Dead Code Elimination?

Dead code elimination removes code statements that compute values never used in any observable way, reducing program size and improving runtime by avoiding unnecessary work.

```
//before
for (int i = 0; i < n; ++i) {
    int x = A[i] * 2;
    // x is never used after this computation
    B[i] = C[i] + D[i];
```

```
}
//after
for (int i = 0; i < n; ++i) {
    B[i] = C[i] + D[i];
}
```

Here, the computation of `x` is removed entirely since its value does not affect any program output or side effect.

what is DAG its example and advantages?

A Directed Acyclic Graph (DAG) is used in compilers to represent expressions and statements in a way that identifies common subexpressions and avoids recomputation. Each node corresponds to a unique computation or variable, and directed edges represent dependencies.

> **Construction Steps:**
>
> 1. **Parse Expressions:** Traverse each statement or expression in the basic block.
> 2. **Node Lookup/Creation:** For each operand or computed result, check if a node for that exact computation exists:
>   - If yes, reuse the existing node (captures common subexpression).
>   - If no, create a new node and record its operator and operands.
> 3. **Add Edges:** Link the operand nodes to the result node, preserving evaluation order.
> 4. **Handle Assignments:** Map variable names to the corresponding DAG nodes, updating when variables are reassigned.

```
// Original code in a basic block
t1 = a + b;
t2 = a + b;
t3 = t1 * c;
x  = t2 * c;
```

> **DAG Nodes:**
>
> - Node N0: `a`
> - Node N1: `b`
> - Node N2: `a + b` (reused for t1 and t2)
> - Node N3: `c`
> - Node N4: `N2 * c` (reused for t3 and x)

```
t1 = a + b;
t3 = t1 * c;
x  = t3;
```

Here, the addition `a + b` and multiplication by `c` are computed only once.

Advantages of DAG Construction

- **Common Subexpression Elimination:** Automatically merges identical computations into a single node.
- **Code Motion Opportunities:** Identifies invariant computations that can be hoisted.
- **Dead Code Detection:** Nodes with no path to output variables can be pruned.
- **Better Register Allocation:** The DAG structure exposes use-def chains for liveness analysis.

Basic blocks its construction steps and example ?

Basic Block Formation

A *basic block* is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit. Compiler analyses often divide code into basic blocks as the first step in control-flow optimization.

> **Steps to Divide Code into Basic Blocks:**
>
> 1. **Identify Leaders:**
>   - The first instruction in the program is a leader.
>   - Any target of a jump (conditional or unconditional) is a leader.
>   - Any instruction immediately following a jump is a leader.
> 2. **Form Blocks:**
>   - For each leader, its basic block consists of the leader and all following instructions up to, but not including, the next leader or jump instruction.
> 3. **Establish Control-Flow Edges:**
>   - Add edges between blocks based on jump and fall-through transfers.

```
1:  a = 1;
2:  if (a < 10) goto L1;
3:  b = 2;
4:  goto L2;
```

```
5: L1: c = 3;
6: L2: d = 4;
```

- **Leaders:** Instruction 1, instruction 5 (`L1:`), instruction 6 (`L2:`).
- **Basic Blocks:**
  - **B1:** Lines 1–2
  - **B2:** Line 3–4
  - **B3:** Line 5
  - **B4:** Line 6

Control-Flow Graph:

- B1 → B2 (fall-through) and B1 → B3 (if branch)
- B2 → B4 (unconditional goto)
- B3 → B4 (fall-through)

PeepHole optimisation?

Peephole optimization examines a small sliding window ("peephole") of consecutive instructions and replaces inefficient sequences with shorter or faster ones.

```
Before peephole optimization
MOV R1, #0     ; load zero into R1
ADD R1, R1, R2 ; R1 = R1 + R2
```

After peephole optimization MOV R1, R2 ; directly move R2 into R1

```
After peephole optimization
MOV R1, R2     ; directly move R2 into R1
```

Here, the window of two instructions is replaced by a single, more efficient instruction.
Key patterns for peephole optimization include:

1. **Redundant Load/Store Elimination:** Remove loads/stores where the value is already in a register.
2. **Strength Reduction:** Replace expensive ops (e.g., multiplication by constant) with shifts/adds when possible.
3. **Branch Simplification:** Combine or invert conditional branches to reduce jumps.
4. **Instruction Combining:** Merge instruction pairs into a single complex instruction supported by the target architecture.

How commutivity and assosciativity can be used to generate more efficiant code from DAG?

By leveraging algebraic properties like commutativity (order of operands) and associativity (grouping of operations), a compiler can restructure DAGs to minimize intermediate temporaries, improve constant folding, and enable further optimizations.

```
Example: Consider the computation:
y = a * b * c;
```

- **Commutativity:** Multiplication is commutative, so nodes for `a*b` and `b*a` are identical and can be shared.
- **Associativity:** Grouping can change without affecting result: `(a * b) * c` vs. `a * (b * c)`.

> **Original DAG:**
>
> - Node N0: `a`
> - Node N1: `b`
> - Node N2: `c`
> - Node N3: `N0 * N1` (computes `a*b`)
> - Node N4: `N3 * N2` (computes `(a*b)*c`)
>
> We can reassociate to compute `b*c` first:
>
> - Node N5: `N1 * N2` (computes `b*c`)
> - Node N6: `N0 * N5` (computes `a*(b*c)`)
>
> Both DAGs produce the same result, but if `b*c` appears elsewhere, Node N5 is shared. Also, if `c` is 1 (constant), constant folding on N5 simplifies early.
>
> **Generated Optimized Code:**

```
tmp1 = b * c;
y    = a * tmp1;
```

By reordering and regrouping:

1. We eliminate redundant temporaries when common subexpressions overlap.
2. We expose opportunities for constant folding and strength reduction earlier.