**Loop Optimization**

When a program runs, loops often consume the majority of the execution time, particularly in applications involving large datasets, graphics, simulations, or artificial intelligence. Because loops repeat operations many times, even a small inefficiency inside a loop can become very costly. Thus, optimizing loops can lead to dramatic improvements in a program’s speed and efficiency. In compiler design, **loop optimization** refers to a group of techniques that either reduce the amount of computation performed inside loops or minimize the number of loop iterations, all while ensuring that the final output remains unchanged.

The main goal of loop optimization is two-fold:  
First, to **reduce the time complexity** of the loop body by eliminating redundant calculations, simplifying operations, and reusing computations wherever possible.  
Second, to **reduce the overhead** of loop control itself, such as incrementing counters and checking loop termination conditions.

Now let us explore the major types of loop optimizations in detail.

**Loop Invariant Code Motion**

One of the most basic forms of loop optimization is **loop invariant code motion**.  
An operation inside a loop is called **loop invariant** if it yields the same result in every iteration because it does not depend on the loop variable or any value modified inside the loop.

Such computations do not need to be performed repeatedly; they can be calculated once before the loop starts and reused inside the loop body.

For example, consider the following code:

for (int i = 0; i < 100; i++) { y = p \* q; z[i] = y + i; }

Here, y = p \* q remains constant across all 100 iterations because p and q do not change inside the loop. Hence, we can optimize the code as:

int y = p \* q; for (int i = 0; i < 100; i++) { z[i] = y + i; }

Thus, only one multiplication happens instead of 100 redundant multiplications.

If a loop invariant computation is not moved out of the loop, it wastes CPU cycles for every iteration, which can lead to huge inefficiencies.

**Important Considerations:**  
It is essential to check that the computation truly remains constant. If the computation indirectly depends on a value that changes inside the loop, moving it out can change the program behavior, which is unacceptable.

**Solved Numerical Example 1:**

Suppose inside a loop of 1000 iterations, a loop-invariant multiplication operation takes 5 CPU cycles, and the rest of the loop body takes 8 CPU cycles per iteration.

**Without optimization:**  
Total time = (5 + 8) × 1000 = 13 × 1000 = 13,000 cycles.

**After optimization:**  
Multiplication done once = 5 cycles.  
Loop body (only 8 cycles per iteration) = 8 × 1000 = 8000 cycles.  
Total time = 5 + 8000 = 8005 cycles.

**Cycles saved:**  
Cycles saved = 13,000 - 8005 = 4995 cycles.

Thus, almost 38% of execution time is saved by simply moving one computation out of the loop.

**Induction Variable Simplification**

Another powerful optimization involves **induction variables**. An induction variable is a variable that changes in a predictable manner with each iteration of the loop, typically by addition or subtraction.

Sometimes, complicated expressions involving the induction variable are recomputed in every iteration. These can often be simplified into simple updates from the previous value.

Consider the following example:

for (int i = 0; i < n; i++) { arr[i] = 5 \* i + 7; }

In each iteration, the multiplication 5 \* i is recomputed. We can simplify this by initializing a temporary variable to 7 before the loop and adding 5 to it on every iteration.

Optimized code becomes:

int temp = 7; for (int i = 0; i < n; i++) { arr[i] = temp; temp = temp + 5; }

Thus, instead of a multiplication per iteration, only an addition (which is faster) is needed.

This is called **induction variable substitution** or **strength reduction** of induction variables.

**Solved Numerical Example 2:**

Suppose multiplication requires 6 cycles while addition requires only 1 cycle.

* Original loop (with multiplication): each iteration = 6 cycles.
* Optimized loop (with addition): each iteration = 1 cycle.

For n = 500 iterations:

**Without optimization:**  
Total cycles = 6 × 500 = 3000 cycles.

**With optimization:**  
Total cycles = 1 × 500 = 500 cycles.

**Cycle saving:**  
Cycle saving = 3000 - 500 = 2500 cycles.

Thus, induction variable simplification gives more than 80% saving in loop execution time.

**Loop Unrolling**

Each loop iteration involves some overhead: incrementing the counter, checking the termination condition, and branching back to the start.  
When loops are small, this overhead can become comparable to the actual work being done.

**Loop unrolling** reduces this overhead by performing multiple iterations' worth of work inside a single iteration.

Suppose originally you have:

for (int i = 0; i < 8; i++) { a[i] = b[i] + c[i]; }

This loop processes one element per iteration.  
We can unroll it by a factor of two:

for (int i = 0; i < 8; i += 2) { a[i] = b[i] + c[i]; a[i+1] = b[i+1] + c[i+1]; }

Now, two elements are processed per iteration, halving the number of loop condition checks and branches.

Loop unrolling can greatly reduce loop overhead, particularly for small simple loops. However, it also increases code size (called **code bloat**), so unrolling must be balanced carefully.

**Solved Numerical Example 3:**

Assume loop control operations (increment, compare, jump) take 2 cycles and the loop body takes 3 cycles.

Original loop:

* Loop overhead per iteration = 2 cycles.
* Body per iteration = 3 cycles.
* Total per iteration = 2 + 3 = 5 cycles.
* Total for 8 iterations = 5 × 8 = 40 cycles.

Unrolled loop (by 2):

* Now 4 iterations.
* Each iteration = 2 (overhead) + 6 (body for 2 elements) = 8 cycles.
* Total = 8 × 4 = 32 cycles.

**Saving:**  
Saving = 40 - 32 = 8 cycles (20% improvement).

Thus, loop unrolling efficiently reduces the number of iterations and the relative cost of loop control.

**Strength Reduction in Loops**

Strength reduction is a classic compiler optimization technique where expensive operations (like multiplication or division) are replaced by cheaper operations (like addition or bit-shifting).

In loops, this is particularly important because the operation may be repeated hundreds or thousands of times.

Suppose the loop is:

for (int i = 0; i < n; i++) { arr[i] = i \* 32; }

Multiplying by 32 can be replaced by shifting i left by 5 bits (i << 5), because 2⁵ = 32.

Thus, the optimized loop becomes:

for (int i = 0; i < n; i++) { arr[i] = i << 5; }

Bitwise shifts are faster than multiplication operations.

**Solved Numerical Example 4:**

Suppose:

* Multiplication takes 5 cycles.
* Bitwise shift takes 1 cycle.

For n = 1000 iterations:

* Without optimization: 5 × 1000 = 5000 cycles.
* With optimization: 1 × 1000 = 1000 cycles.

**Cycles saved:**  
5000 - 1000 = 4000 cycles (80% faster).

Strength reduction thus offers a major performance boost, particularly in tight loops.

Loop optimizations aim to reduce unnecessary computations, simplify expensive operations, and minimize the number of iterations or loop overhead. Loop invariant code motion moves constant calculations out of the loop. Induction variable simplification replaces complex updates with simple arithmetic. Loop unrolling reduces the number of jumps and condition checks. Strength reduction substitutes costly multiplications or divisions with cheap additions or shifts. Together, these optimizations can greatly improve the speed and efficiency of programs, sometimes achieving improvements of 30%, 50%, or even 80%, especially in critical or performance-sensitive applications.

Care must be taken to ensure that after any optimization, the logic of the program remains exactly the same, and numerical accuracy is preserved if applicable.

**The DAG Representation of Basic Blocks**

When compilers try to optimize programs, one of their goals is to identify redundant computations and eliminate them.  
To achieve this efficiently, especially inside a **basic block**, compilers use a data structure known as a **Directed Acyclic Graph (DAG)**.

A **basic block** is a sequence of instructions in which control enters at the beginning and exits at the end without any branching except possibly at the end. Within a basic block, the instructions are executed one after another without interruption, which makes it a perfect unit for local optimizations.

The **DAG representation** of a basic block provides a visual and logical method to capture computations and their dependencies. It allows the compiler to:

* Detect **common subexpressions** (i.e., the same computation appearing multiple times).
* Identify **dead code** (computations whose results are never used).
* Optimize the code by **reusing computed values** and **eliminating redundant instructions**.

Let us now understand how a DAG for a basic block is constructed and used for optimization.

**Basic Concepts of DAG Representation**

In a DAG for a basic block:

* **Leaves** represent either constants or variables that have values before the block starts.
* **Interior nodes** represent operations such as addition, subtraction, multiplication, division, etc.
* **Edges** represent dependencies: if a node n has children n₁ and n₂, it means that n depends on the results of n₁ and n₂.
* A **label** at a node keeps track of the variable(s) that the result is assigned to.

Because a DAG is **acyclic**, there are no cycles, meaning a computation cannot depend on itself, which is natural in the context of sequential instructions.

**Constructing a DAG**

To construct a DAG from a sequence of instructions inside a basic block, we proceed step-by-step as follows:

1. **For every computation**, check whether a node with the same operands and operator already exists.
2. If it exists, reuse that node.
3. If it does not exist, create a new node.
4. **Label the node** with the name of the variable assigned in the instruction.

Let us now study this using a detailed example.

**Detailed Example of DAG Construction**

Suppose we have the following sequence of instructions in a basic block:

1. t1 = a + b

2. t2 = t1 - c

3. t3 = a + b

4. t4 = t3 - d

5. t5 = t2 \* t4

Let’s build the DAG step-by-step:

* First, t1 = a + b.  
  We create a node with operation + and children a and b. Label it t1.
* Next, t2 = t1 - c.  
  Since t1 is already represented in the graph, create a node - with children t1 and c. Label it t2.
* Now, t3 = a + b appears again.  
  Before creating a new node, we check: a + b already exists (from t1). Thus, no need to create a new node.  
  Instead, we just add the label t3 to the existing a+b node.
* Next, t4 = t3 - d.  
  Since t3 points to the existing a+b node, create a new - node with children a+b and d. Label it t4.
* Finally, t5 = t2 \* t4.  
  Create a node \* with children t2 and t4. Label it t5.

Thus, the DAG efficiently represents that t1 and t3 are the same computation (a+b), and no extra computation is needed.

**Optimizing the Basic Block Using the DAG**

Once the DAG is constructed:

* We recognize that the calculation a+b is common between t1 and t3, and compute it only once.
* There is no need to recompute a+b separately for t3.
* If any variable (say t2 or t4) is found unused later, the corresponding sub-graph can be eliminated (dead code elimination).

Thus, using the DAG, redundant computations and dead code can be safely removed without affecting the correctness of the program.

**Common Subexpression Elimination**

One of the most important uses of the DAG is in **common subexpression elimination**.

In the above example, a+b was computed twice if we naively followed the original code. But in the DAG, since a+b appears only once, both t1 and t3 can be assigned from a single computation, thus eliminating one subexpression.

If two different instructions compute the same expression and neither of the involved variables (a or b) are modified in between, then the second computation is unnecessary.

Thus, **DAG allows identification and elimination of such redundant subexpressions automatically**.

**Dead Code Elimination**

Another powerful application of DAGs is **dead code elimination**.

Suppose after the basic block, the value of a temporary variable is never used. Then, any computation leading to that variable can be safely eliminated.

For example, if t2 was not used later, the subgraph involving t2 (i.e., t1-c) could be removed.

Thus, DAGs help compilers to perform safe and powerful dead code elimination within basic blocks.

**Solved Numerical Example: Build a DAG and Optimize**

Given the following instructions:

1. t1 = x + y

2. t2 = t1 - z

3. t3 = x + y

4. t4 = t2 \* t3

**Step-by-step DAG construction:**

Step 1: Create t1 node with + operator and children x and y.

Step 2: Create t2 node with - operator, children are t1 and z.

Step 3: For t3 = x + y, we observe that x+y is already computed in t1, so we reuse the existing + node and label it also as t3.

Step 4: Create t4 node with \* operator, children are t2 and t3.

**Optimized Code:**

After building the DAG, the optimized code sequence is:

* Compute t1 = x + y.
* Compute t2 = t1 - z.
* Compute t4 = t2 \* t1. (Since t3 = t1)

Notice that t3 is eliminated as a separate computation, saving one addition operation.

Thus, The DAG representation of a basic block is a crucial technique for local optimization in compilers.  
It allows:

* Detection and elimination of **common subexpressions**.
* Removal of **dead code**.
* Efficient representation of computations.

By building a DAG, the compiler avoids redundant work and ensures that programs execute faster without changing their behavior.  
The DAG thus acts like a **blueprint** for better program code inside each basic block.

**Value Numbering and Algebraic Laws**

When optimizing a program inside a basic block, one of the most important tasks for a compiler is to recognize when two different computations are actually **computing the same value**.  
Instead of always using a Directed Acyclic Graph (DAG), another very efficient technique for small and quick optimizations is known as **Value Numbering**.

**Value numbering** assigns a unique **value number** to each distinct computed value inside a basic block.  
If two computations result in the same value number, then they are recognized as computing the same thing, even if they appear as separate instructions.

This simple idea is extremely powerful because it allows a compiler to:

* **Identify common subexpressions quickly**.
* **Reuse previously computed results** instead of recomputing them.
* **Eliminate redundant calculations** efficiently.

Let us now study how value numbering works in detail.

**Basic Concept of Value Numbering**

When a basic block is being optimized:

* Each variable, constant, or result of an operation is associated with a unique value number.
* When a new instruction is encountered, we check if an identical computation already exists by comparing value numbers.
* If it exists, we reuse the earlier computation.
* Otherwise, a new value number is assigned.

For example, suppose we encounter the following statements:

t1 = a + b

t2 = b + a

In normal execution, t1 and t2 would be computed separately.  
However, by **value numbering**, we recognize that **addition is commutative** (a+b = b+a).  
Thus, t2 can reuse the result of t1, avoiding a redundant addition.

Thus, **algebraic laws** like **commutativity**, **associativity**, and **distributivity** are used to recognize when two expressions are actually equivalent even if they appear different in writing.

**Detailed Step-by-Step Example of Value Numbering**

Let us consider the following sequence of statements inside a basic block:

1. t1 = a + b

2. t2 = b + a

3. t3 = a \* b

4. t4 = a + (b + c)

5. t5 = (a + b) + c

Now, let us apply value numbering manually:

* At instruction 1, t1 = a + b, we assign it a new value number, say **v1**.
* At instruction 2, t2 = b + a, we realize that addition is commutative.  
  So, b+a is the same as a+b. Hence, t2 also gets value number **v1**.
* At instruction 3, t3 = a \* b, multiplication is also commutative.  
  Since no previous multiplication of a\*b is present, we assign it a new value number **v2**.
* At instruction 4, t4 = a + (b + c), parentheses matter.  
  Without applying associativity yet, we first need to compute b+c, assign a value number (say **v3**), and then add a+v3, giving a new result **v4**.
* At instruction 5, t5 = (a+b)+c.  
  Here, because of **associativity of addition**, (a+b)+c = a+(b+c).  
  Thus, t5 is equivalent to t4, and hence t5 can reuse the result of t4.

Thus, even though the code seems to have five computations, **only three real unique computations** occur:  
one addition (a+b), one multiplication (a\*b), and one associative addition (a+(b+c)).

**Algebraic Laws Used in Value Numbering**

To recognize such equivalences, compilers apply important algebraic laws:

* **Commutativity**:  
  For addition and multiplication, the order of operands does not matter.  
  That is, a+b = b+a, and a\*b = b\*a.
* **Associativity**:  
  For addition and multiplication, grouping does not matter.  
  That is, (a+b)+c = a+(b+c), and (a\*b)\*c = a\*(b\*c).
* **Distributivity**:  
  Multiplication distributes over addition: a\*(b+c) = a\*b + a\*c.

Using these algebraic laws during value numbering helps the compiler find more cases where different-looking computations are actually the same.

**Solved Numerical Example: Value Numbering and Optimization**

Given the following instructions:

1. t1 = p + q

2. t2 = q + p

3. t3 = r \* (p + q)

4. t4 = r \* t1

5. t5 = p \* q + p \* q

Apply value numbering and show how optimizations are found.

**Step-by-step solution:**

Step 1: t1 = p + q  
Assign value number **v1** to p+q.

Step 2: t2 = q + p  
Since addition is commutative, q+p = p+q.  
Thus, t2 also gets **v1**.

Step 3: t3 = r \* (p+q)  
Since p+q has value number **v1**, we interpret t3 = r \* v1.  
Assign a new value number **v2** to this result.

Step 4: t4 = r \* t1  
t1 is already v1, thus t4 = r \* v1, which is same as t3.  
Thus, t4 can reuse **v2** without recomputation.

Step 5: t5 = p\*q + p\*q  
First, compute p\*q, assign a value number **v3**.  
Since both operands are p\*q, t5 simplifies to 2\*(p\*q) or 2\*v3, a multiplication by constant.

Thus, the optimized operations are:

* Compute v1 = p + q.
* Compute v2 = r \* v1.
* Compute v3 = p \* q.
* Compute t5 = 2 \* v3.

Notice that only four real computations are needed instead of five.

**Value numbering** is a powerful method for **local optimization** inside basic blocks.  
It allows the compiler to **detect identical values**, **eliminate redundant computations**, and **reuse previous results**.  
Using **algebraic laws** like **commutativity**, **associativity**, and **distributivity**, it becomes possible to find even hidden equivalences between expressions that do not look identical at first.

Value numbering is computationally cheap and extremely useful for small blocks of code where full DAG construction might be unnecessarily expensive.

By applying value numbering properly, compilers make programs faster and reduce the number of instructions without sacrificing correctness.

**Global Data-Flow Analysis**

In earlier topics like **Loop Optimization**, **DAG construction**, and **Value Numbering**, we mostly dealt with **local optimizations** inside a **basic block** — meaning optimizations were confined to a small sequence of instructions without any branches or jumps.

However, real-world programs are made up of many basic blocks connected by branches, loops, and conditionals.  
Thus, to perform optimizations **across basic blocks**, compilers use a more powerful technique called **Global Data-Flow Analysis**.

**Global Data-Flow Analysis** is a systematic way to gather information about how variables behave across an entire program (or a large region like a procedure or function) instead of just looking at isolated basic blocks.  
This analysis helps compilers perform important optimizations such as:

* **Dead code elimination** across basic blocks.
* **Code motion** across basic blocks (moving calculations from inside loops to outside).
* **Register allocation** based on variable lifetimes.
* **Constant propagation** and **strength reduction** across functions.

Thus, global data-flow analysis is at the heart of many advanced optimizations.

**Basic Concepts of Data-Flow Analysis**

Global Data-Flow Analysis is based on the idea that at any point in a program, the compiler needs to know information such as:

* Which definitions of variables reach this point? (**Reaching Definitions**)
* Which variables are alive and needed later? (**Live Variables**)
* Which expressions have already been computed and are available? (**Available Expressions**)

At every basic block, we define two sets:

* **IN[B]**: Information that is true just **before** entering basic block B.
* **OUT[B]**: Information that is true **after** completing basic block B.

Using **transfer functions** and **data-flow equations**, we calculate these sets for each basic block, usually by solving them iteratively.

**Important Problems in Data-Flow Analysis**

There are several classic problems that data-flow analysis can solve. Let’s explain the most important ones:

**1. Reaching Definitions**

A **definition** of a variable (like a = 5) **reaches** a point in the program if there exists a path from the definition to that point without any redefinition of the variable along the path.

**Purpose**: To know which previous assignments might influence the current use of a variable.

**Example**:  
Consider the code:

1. a = 3

2. b = a + 5

3. a = 7

4. c = a + b

At line 2, the definition a = 3 reaches.  
At line 4, the definition a = 7 reaches, not a = 3, because a was redefined at line 3.

Thus, data-flow analysis helps track active definitions across program paths.

**2. Live Variable Analysis**

A variable is said to be **live** at a point if its value is going to be used in the future, before it is redefined.

**Purpose**:  
If a variable is not live, its storage can be freed or its computation can be eliminated.

**Example**:  
Suppose:

1. a = 5

2. b = a + 2

3. a = 8

4. c = a + b

After line 2, a is no longer needed because it is redefined at line 3. Thus, the first a = 5 becomes dead after line 2.

Thus, live variable analysis helps identify dead code.

**3. Available Expressions**

An expression like x + y is said to be **available** at a point if:

* It has been computed previously.
* Neither x nor y have changed since then.

**Purpose**:  
To avoid recomputing expensive expressions if they are already available.

**Example**:  
Suppose:

1. t1 = x + y

2. a = t1 \* 2

3. b = x + y

At line 3, x + y is available because neither x nor y changed after line 1. Thus, line 3 can reuse t1 instead of recalculating x + y.

**Data-Flow Equations**

At the heart of global data-flow analysis are **data-flow equations**.

For every basic block **B**, the general structure is:

* OUT[B] = f(IN[B]) (using the statements inside the block)
* IN[B] = union of OUT[P] for all predecessors P of B

where f() is a **transfer function** that defines how the block changes information.

These equations are usually solved **iteratively** until no further changes occur (this is called **reaching a fixed point**).

**Solved Numerical Example: Reaching Definitions**

Let us solve a real example manually:

**Given basic blocks:**

B1:

1. a = 5

2. b = 7

B2:

3. c = a + b

B3:

4. a = 8

5. d = a + c

Control flow:

* B1 → B2 → B3

**Step-by-step solution:**

**Initialization**:  
At the beginning, we assume IN[B1] = {} (no prior definitions).

**Transfer functions**:

* In B1, after a=5 and b=7, OUT[B1] = {a=5, b=7}.
* In B2, c = a+b, but this is a use, not a new definition, so definitions flow through:  
  OUT[B2] = {a=5, b=7, c=a+b}.
* In B3, a = 8 redefines a, so it kills a=5.  
  Thus, OUT[B3] = {a=8, b=7, c=a+b, d=a+c}.

Thus, **Reaching Definitions** at each point are:

* **IN[B1]** = {}
* **OUT[B1]** = {a=5, b=7}
* **IN[B2]** = {a=5, b=7}
* **OUT[B2]** = {a=5, b=7, c=a+b}
* **IN[B3]** = {a=5, b=7, c=a+b}
* **OUT[B3]** = {a=8, b=7, c=a+b, d=a+c}

Thus, by applying the data-flow equations iteratively, we find how definitions reach different parts of the program.

Global Data-Flow Analysis extends the ideas of local optimization to the entire program flow.  
It provides structured, mathematical ways to track:

* Where variables are defined and used.
* Which expressions are available.
* Which variables are live or dead.

Using **IN** and **OUT** sets, **transfer functions**, and **fixed-point iteration**, compilers collect global information that enables highly effective optimizations such as:

* Dead code elimination
* Code motion
* Register allocation
* Constant propagation

Without data-flow analysis, a compiler would not be able to generate efficient machine code for large, branching programs.