1. Introduction to Code Optimization

1. Introduction :

Code optimization is the process of improving the intermediate code to produce an equivalent target code that is more efficient in terms of time and space. The meaning or result of the program remains unchanged.

Goal of Code Optimization

- ✓ Reduce execution time.
- ✓ Reduce memory usage.
- ✓ Improve throughput and performance.
- ✓ Reduce power consumption.
- ✓ Make better use of hardware resources.
- ✓ Keep program size small.

2. Why is Code Optimization Needed ?

- ➔ High level languages produce inefficient code.
- ➔ Naïve translation leads to redundant computations.
- ➔ Better performance is required for real-time systems.
- ➔ To reduce cost and improve reliability.
- ➔ To utilize registers and memory efficiently.
- ➔ To adapt code for different target machines.

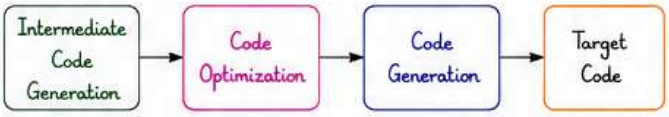
3. Example (Need of Optimization) :

Consider the following code segment :

Unoptimized Code	Optimized Code
t1 = a + b t2 = a + b t3 = t1 * c t4 = t2 * c d = t3 + t4	t1 = a + b t2 = t1 * c d = t2 + t2

In optimized code, redundant computations are eliminated.

4. Phases of Optimization in Compiler :



Optimization is usually performed on intermediate code.

5. Types of Optimization :

- ✱ Local (Machine Independent) Optimization :
  Does not depend on target machine. Done on IR.
- ✱ Machine Dependent Optimization :
  Depends on target machine architecture.

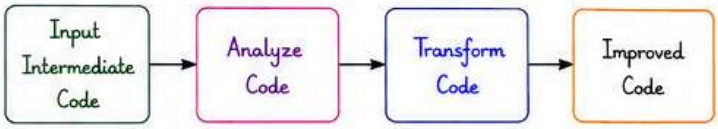
Important Point

- ✱ Optimization improves efficiency but increases compile time.
- ✱ Over-optimization may increase code size.
- ✱ Correctness must never be sacrificed.

6. Optimization Levels :

- Peephole Optimization (Very local).
- Basic Block Level Optimization (Local).
- Loop Level Optimization.
- Global Optimization (across basic blocks / whole program).

7. General Model of Code Optimization :



Where :

Analyze : Find inefficiencies, redundancies, dependencies.

Transform : Apply optimization techniques.

Improved Code : Equivalent but more efficient code.

8. Importance of Code Optimization :

- ✱ Improves performance of programs.
- ✱ Reduces execution time.
- ✱ Reduces memory requirements.
- ✱ Increases throughput.
- ✱ Important for embedded and real-time applications.
- ✱ Provides competitive advantage in software.

Benefit
Better Program Performance with same functionality.

9. Where Optimization Can Be Applied ?

- ✱ During or after intermediate code generation.
- ✱ On basic blocks.
- ✱ Across basic blocks (global).
- ✱ Inside loops.
- ✱ During code generation for target machine.

10. Example to Understand Optimization :

Unoptimized Code :

```
x = a + b
y = a + b
z = x * y
w = z + 0
u = w * 1
```

(Redundant operations removed)

Optimized Code :

```
t1 = a + b
t2 = t1 * t1
u = t2
```

11. Summary :

- ✱ Code optimization is an essential phase of compiler.
- ✱ It improves the quality and efficiency of target code.
- ✱ It eliminates redundancies and makes better use of resources.
- ✱ It can be applied at various levels.
- ✱ Optimization must preserve the original meaning of the program.

2. Sources of Optimization

1. Introduction :

A compiler can perform optimization at different places. These places or parts of the program from where optimization opportunities can be identified are called sources of optimization. These sources help the compiler to apply proper optimization techniques to improve code.

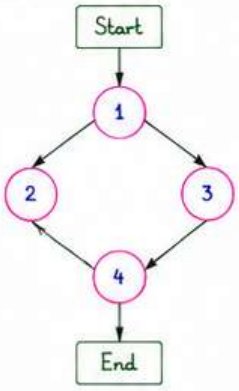
Goal :
- ✓ Find opportunities
- ✓ Remove inefficiencies
- ✓ Improve performance
- ✓ Produce efficient target code

2. Main Sources of Optimization :

Source	Description
1. Basic Blocks	Sequence of instructions with single entry and single exit. Many local optimization opportunities exist here.
2. Loops in Flow Graphs	Loops cause repeated execution. Optimizing loops greatly improves performance.
3. Global Data Flow Information	Information about data values across basic blocks helps in global optimization.
4. Redundant / Dead Code	Code which does not affect output. Eliminating it improves efficiency.
5. Expressions and Variables	Common subexpressions, constant expressions, copy propagation etc. provide optimization chances.
6. Control Flow Information	Information about branches, jumps and conditions helps in code improvement.
7. Register and Memory Usage	Better use of registers and memory reduces load/store operations.

3. Flow Graph as a Source :

Flow graph (CFG) represents control flow of a program. It helps compiler to identify basic blocks, loops and paths for optimization.



Basic Blocks :
- B1 = {1}
- B2 = {2}
- B3 = {3}
- B4 = {4}

Source Information :
- Entry point
- Loops (if any)
- Branches
- Join points
- Exit point

Importance :
- Helps in identifying blocks and loops.
- Helps in performing data flow analysis.
- Provides structure for optimization.

4. Types of Optimization Opportunities :

1. Eliminate redundant computations.
2. Remove dead code.
3. Simplify expressions.
4. Reduce strength of operations.
5. Improve register allocation.
6. Reduce memory references.
7. Improve instruction scheduling.
8. Unroll loops.
9. Code motion (move computations).

Note : Not all opportunities are applicable at all places. Compiler must choose best one.

5. Local vs Global Sources :

Aspect	Local Sources	Global Sources
Information	Within a basic block	Across basic blocks or whole program
Example	Constant folding, dead code elim.	Common subexpr. elim., code motion
Complexity	Less complex	More complex
Benefit	Quick improvement	Better improvement
Used For	Local optimization	Global optimization

6. Example - Different Sources :

Consider the following code :

```
a = b + c
d = a * 2
e = d + 0
if (e > 0)
    f = e + a
g = a + c
if (g > 10)
    h = g * 1
```

Optimization sources in above code :
- Redundant expression : a = b + c used again in g = a + c
- e = d + 0 → adding 0 (can remove)
- h = g * 1 → multiplying 1 (can remove)
- Dead code : if result unused
- Loop (if any) not shown here

7. Summary :

- Sources of optimization are the places where compiler finds improvement opportunities.
- Major sources are basic blocks, loops, data flow information, dead code, expressions and control flow.
- Local sources give quick benefits, global sources give better overall performance.
- Proper analysis of these sources leads to efficient optimized code.

3. Optimization of Basic Blocks

1. Introduction :

A basic block is a sequence of instructions with single entry and single exit. Optimizing basic blocks is the simplest and most local form of optimization. It reduces the number of instructions inside a block without changing the meaning of the program.

Goal :

- ✓ Reduce number of instructions.
- ✓ Remove redundant computations.
- ✓ Improve speed.
- ✓ Reduce code size.

2. What is a Basic Block ?

A basic block is a straight line code with :

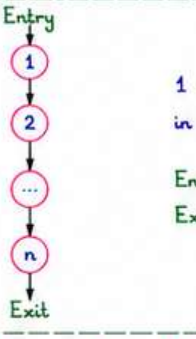
- Single entry point (first instruction).
- Single exit point (last instruction).
- No branches except at the end.

Example :

```
t1 = a + b
t2 = t1 * c
t3 = d - e
if t3 > 0 goto L1
```

→ This is a basic block.
Entry → first instruction a + b
Exit → last instruction (if condition)

3. Flow Graph of a Basic Block :



1 to n are instructions in a basic block.

Entry → node 1
Exit → node n

4. Opportunities in Basic Block Optimization :

1. Eliminate redundant expressions.
2. Eliminate common subexpressions.
3. Eliminate dead code.
4. Constant folding and propagation.
5. Strength reduction.
6. Copy propagation.
7. Algebraic simplification.
8. Code motion.

Note :

These optimizations are performed without knowing information about other blocks.

5. Techniques for Basic Block Optimization :

Technique	Purpose
Constant Folding	Evaluate constant expressions at compile time.
Constant Propagation	Replace variables with constant values.
Common Subexpression Elimination	Avoid re-computation of same expression.
Dead Code Elimination	Remove code that does not affect output.
Algebraic Simplification	Use algebraic identities to simplify code.
Strength Reduction	Replace expensive operations by cheaper ones.
Copy Propagation	Replace variables by their copies.
Code Motion	Move code to reduce execution frequency.

6. Example - Optimization in a Basic Block :

Unoptimized Code :

```
1  t1 = a + b
2  t2 = a + b
3  t3 = t1 * 2
4  t4 = t3 * 1
5  t5 = 0 + t3
6  t6 = t5
```

(Redundant code removed)

Optimized Code :

```
1  t1 = a + b
2  t3 = t1 * 2
3  t6 = t3
```

Explanation :

Step 1 : t1 is used again in line 2 → so replace t2 by t1 (removed line 2).
Step 2 : Multiplying by 1 does nothing → remove line 4.
Step 3 : Adding 0 does nothing → remove line 5.
Step 4 : t6 = t5 → replace by t6 = t3.

7. Important Local Optimizations :

Optimization	Before	After	Explanation
Constant Folding	t = 3 + 4	t = 7	Compute constant expression.
Constant Propagation	a = 5 b = a + 2	a = 5 b = 7	Replace variable by constant.
Common Subexp. Elimination	t1 = a + b t2 = a + b	t1 = a + b t2 = t1	Same expression computed once.
Dead Code Elimination	t = a + b x = t y = 10	t = a + b y = 10	Variable x is never used.
Algebraic Simplification	t = x * 1 u = y + 0 w = z * 2	t = x u = y w = z + z	Using algebraic identities.
Strength Reduction	t = x * 8	t = x << 3	Replace multiply by shift.

8. Benefits of Basic Block Optimization :

- Reduces the number of instructions.
- Reduces execution time.
- Improves cache utilization.
- Reduces code size.
- Improves overall performance.

Benefit :

Small changes at local level give big improvement in performance.

9. Summary :

- A basic block has single entry and single exit.
- Basic block optimization is the first and simplest level of optimization.
- It includes techniques like constant folding, propagation, CSE, dead code elimination, algebraic simplification, strength reduction, copy propagation and code motion.
- These optimizations improve the intermediate code efficiently without using global information.

Exam Tip :

- Always identify basic blocks first.
- Apply as many local optimizations as possible.
- Show before and after code in exams.
- Write advantages of basic block optimization.

4. Loops in Flow Graphs

1. Introduction :

In a flow graph, a loop (or cycle) is a set of nodes with a path from some node in the set back to the same node. Loops cause repeated execution of a group of statements. Optimizing loops gives huge improvement in performance.

Goal :
- ✓ Reduce loop overhead.
- ✓ Reduce repeated computations.
- ✓ Increase speed.
- ✓ Reduce memory access.

2. What is a Loop in Flow Graph ?

A loop exists in a flow graph if there is a path from a node back to a dominator node.

It has :
- Entry node (loop header).
- Body (set of nodes inside loop).
- Exit node(s).

Loop = Cycle with single entry.

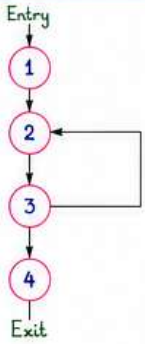
3. Types of Loops :

① Natural Loop : Has single entry.

② Irreducible Loop : Has multiple entry points.

③ Simple Loop : Single condition loop (for, while).

4. Example - Flow Graph with a Loop :

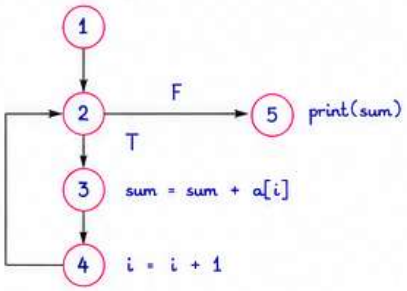


Explanation :
- Loop header : Node 2
- Loop body : Nodes 2, 3
- Back edge : 3 → 2
- Exit node : 4

5. Natural Loop Example (Code) :

```
i = 1;
while (i <= n) {
    sum = sum + a[i];
    i = i + 1;
}
print(sum);
```

Flow Graph :



6. Characteristics of Loops in Flow Graphs :

- A loop has a single entry (natural loop).
- It may have multiple exits.
- It contains at least one back edge.
- Loop body may contain nested loops.
- Loop optimization significantly improves performance.

Important Points :
- Loops increase execution time.
- Nested loops are more expensive.
- Loop optimization is most important in compiler.
- Always identify loops in flow graph first.

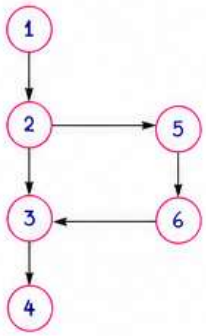
7. Detecting Loops in Flow Graphs :

A back edge exists if there is an edge from node n to a dominator node d where d dominates n.

Back edge indicates a loop.

Note : Dominator of a node n is a node d that appears on every path from Entry to n.

Example :



Dominators	Node
1	{1}
2	{1,2}
3	{1,2,3}
4	{1,2,3,4}
5	{1,2,5}
6	{1,2,3,6}

Back edges : 6 → 3 (since 3 dominates 6)

So loop exists with header 3.

8. Importance of Loop Optimization :

- Loops are executed many times.
- Small improvement in loop gives big benefit.
- Reduces time complexity.
- Reduces memory access.
- Improves cache performance.

Benefit : Faster execution and better resource utilization.

9. Opportunities in Loop Optimization :

Opportunity	Description
Loop Invariant Code Motion	Move computations out of loop that do not change inside loop.
Strength Reduction	Replace expensive operations with cheaper ones.
Induction Variable Optimization	Eliminate unnecessary induction variables.
Loop Unrolling	Duplicate loop body to reduce overhead.
Dead Code Elimination	Remove dead statements inside loop.
Code Motion	Move computations inside or outside loop.

10. Example - Loop Invariant Code Motion :

Before Optimization :

```
for (i = 1; i <= n; i++) {
    t = a * b;    // invariant
    c[i] = t + d[i];
}
```

⇨

After Optimization :

```
t = a * b;    // moved outside
for (i = 1; i <= n; i++) {
    c[i] = t + d[i];
}
```

Explanation :

a*b is invariant inside loop, so moved outside, thus reduces repeated computation.

11. Summary :

- Loops in flow graphs cause repeated execution.
- Natural loops have single entry and one or more exits.
- Loops can be detected using dominators and back edges.
- Loop optimization is essential for efficient code.
- Various techniques are used to optimize loops.

5. Dead Code Elimination

1. Introduction :

Dead code is a part of program that does not affect the output of the program. Dead code elimination is the process of removing such statements from the program. It reduces code size and improves execution time.

Goal :
- ✓ Reduce code size.
- ✓ Reduce execution time.
- ✓ Improve performance.
- ✓ Make program efficient.

2. What is Dead Code ?

A statement is dead if :
- * Its result is never used.
- * It does not affect any future computation.
- * Its removal does not change program output.

Dead Code
No impact on program output.

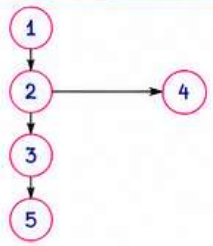
3. Example :

Consider the following code :

```
1. a = b + c
2. d = e + f
3. g = a + d
4. x = y * z    // dead code
5. print(g)
```

⇒ Statement 4 does not affect output. So it is dead code.

4. Flow Graph Representation :



Explanation :
- * Node 4 (x = y * z) has no path to node 5 (print(g)).
- * So node 4 is dead.
- * It does not contribute to output.

5. Algorithm for Dead Code Elimination :

① Identify all variables whose values are used in output.
② Mark all statements that affect these variables.
③ Traverse backward in the flow graph.
④ Mark all statements that affect marked variables.
⑤ Remove all unmarked statements (dead code).

Important Point :
- * Work backward from outputs.
- * Remove statements that do not affect final output.
- * It is a local optimization technique.

6. Example - Dead Code Elimination :

Before Optimization :

```
1. a = b + c
2. d = e + f
3. g = a + d
4. x = y * z    // dead code
5. w = p + q    // dead code
6. print(g)
```

⇒

After Optimization :

```
1. a = b + c
2. d = e + f
3. g = a + d
4. print(g)
```

Explanation :
Statements 4 and 5 are removed because their results are never used in final output.

7. Types of Dead Code :

Type	Description
Unused Variables	Variables whose values are computed but never used.
Unreachable Code	Code that cannot be executed due to control flow.
Redundant Computations	Computations whose results are overwritten before use.
Ineffective Code	Code that has no effect on program output.

8. Detecting Dead Code (Data Flow Approach) :

Live Variable Analysis (Backward)

Compute LIVE OUT for each node → Compute LIVE IN for each node → If variable is not LIVE OUT, its computation is dead → Eliminate dead code

Where :
LIVE OUT[n] = Set of variables live after node n.
LIVE IN[n] = Set of variables live before node n.
If a variable is not live after a statement, then that statement is dead.

9. Example - Using Live Variable Analysis :

Node	Statement	LIVE OUT	LIVE IN	Action
5	print(g)	{ }	{ g }	g is live
4	x = y * z	{ g }	{ g }	x not live → Dead
3	g = a + d	{ x, g }	{ a, d }	g is live
2	d = e + f	{ a, d }	{ a, e, f }	d is live
1	a = b + c	{ a, e, f }	{ b, c, e, f }	a is live

So, statement at node 4 is dead and can be eliminated.

10. Benefits of Dead Code Elimination :

- * Reduces code size.
- * Reduces execution time.
- * Improves memory utilization.
- * Simplifies other optimization passes.
- * Improves readablity of code.

Benefit :
Efficient code with less size and faster execution.

11. Limitations :

- * May not remove all redundancies.
- * Depends on accurate data flow information.
- * Some dead code may appear after other optimizations.

Note :
Dead code elimination is safe because it does not change the output of program.

12. Summary :

- * Dead code does not affect program output.
- * Dead code elimination removes such unnecessary statements.
- * It is based on data flow analysis (live variable analysis).
- * It reduces code size and improves performance.
- * It is an important local optimization technique.

6. Loop Optimization

1. Introduction :

Loop optimization is the process of improving the code inside a loop so that the loop executes faster and uses fewer resources. Since loops are executed many times, small improvements inside the loop body can give significant overall performance gains.

Goal :
- ✓ Reduce loop execution time.
- ✓ Reduce loop overhead.
- ✓ Improve program performance.

2. Why Loop Optimization is Important ?

- Loops are executed repeatedly.
- Even small savings in each iteration give big benefit.
- Common in numerical and array intensive programs.
- Improves CPU cache usage and memory access.

3. Common Loop Optimization Techniques :

Technique	Description
Loop Invariant Code Motion	Move computations that do not change inside loop to outside the loop.
Strength Reduction	Replace expensive operations (like *, /) with cheaper operations (like +, <<).
Induction Variable Optimization	Eliminate unnecessary induction variables and simplify their updates.
Loop Unrolling	Duplicate loop body to reduce overhead of loop control.
Dead Code Elimination	Remove statements inside loop that do not affect output.
Code Motion (Hoisting & Sinking)	Move computations upward (hoist) or downward (sink) to reduce execution time.
Peeling	Execute first few iterations separately to simplify remaining loop.
Fusion (Jamming)	Combine two adjacent loops operating on same data.
Fission (Splitting)	Split a single loop into multiple loops.

4. Example - Loop Invariant Code Motion :

Before Optimization :

```
for (i = 0; i < n; i++) {
  t = a * b;    // invariant
  c[i] = t + d[i];
}
```

After Optimization :

```
t = a * b;    // moved outside
for (i = 0; i < n; i++) {
  c[i] = t + d[i];
}
```

Explanation : a*b is same in every iteration, so compute it once.

5. Example - Strength Reduction :

Before Optimization :

```
for (i = 1; i <= n; i++) {
  y = y * 2;    // multiply
}
```

After Optimization :

```
y = 2;
for (i = 1; i <= n; i++) {
  y = y + y;    // add
}
```

Explanation : Multiplication is costly; replaced by addition.

Important Point :
- Always try to reduce work done inside the loop.
- Move invariant code outside the loop.
- Use cheaper operations in place of expensive ones.
- Remove unnecessary computations.

6. Example - Loop Unrolling :

Before (Original Loop) :

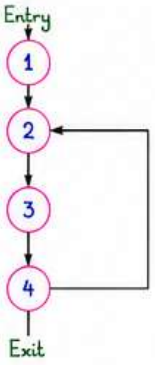
```
for (i = 0; i < n; i++) {
  sum = sum + a[i];
}
```

After (Unrolled by 2) :

```
for (i = 0; i < n; i = i + 2) {
  sum = sum + a[i];
  if (i+1 < n)
    sum = sum + a[i+1];
}
```

Explanation : Loop body is executed two times in each iteration, so loop control overhead is reduced.

7. Flow Graph Example of Loop :



Description :
- Node 1 : Entry
- Node 2 : Loop header
- Node 3 : Loop body
- Node 4 : Exit
- Back edge : 3 → 2

This is a natural loop.

8. Other Advanced Loop Optimizations :

Technique	Description
Loop Peeling	Execute first iteration separately to simplify loop body.
Loop Fusion	Combine two loops that use same data and have same bounds.
Loop Fission	Split one loop into two loops to improve cache performance.
Blocking (Tiling)	Break loop into blocks to improve cache locality.

9. Benefits of Loop Optimization :

- Reduces execution time.
- Improves cache performance.
- Reduces memory traffic.
- Reduces code size (sometimes).
- Better utilization of CPU resources.

Benefit : Big improvement in performance with small changes.

10. Limitations :

- Not all loops can be optimized.
- Some transformations may increase code size.
- Compilers must ensure correctness after optimization.

Note : Always maintain correctness of program.

11. Summary :

- Loops are executed repeatedly, so their optimization gives huge performance boost.
- Techniques like code motion, strength reduction, unrolling, fusion, fission etc. are used.
- Always move invariant code outside the loop.
- Use cheaper operations and reduce loop overhead.
- Compiler uses these techniques automatically to generate efficient target code.

7. Introduction to Global Data Flow Analysis

1. Introduction :

A program consists of many basic blocks connected by control flow. Local data flow analysis works inside a basic block. Global data flow analysis works on the whole program (flow graph). It collects information from all blocks and combines it to find facts that hold at different points in the program.

Goal :
- ✓ Collect information across basic blocks.
- ✓ Learn facts about variables at each program point.
- ✓ Use these facts for optimization.

2. Need for Global Data Flow Analysis :

- Program statements in one block may affect other blocks.
- Helps in detecting properties like variable value, liveness, reaching definitions, available expressions, etc.
- Provides information needed for global optimizations like code motion, dead code elimination, etc.

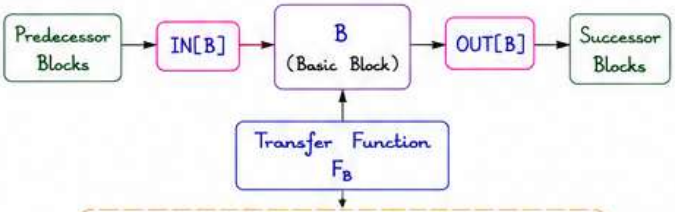
3. What is Global Data Flow Analysis ?

It is the problem of computing information that may be valid at various points over all possible paths in the flow graph of a program.

4. Basic Concepts :

1. Flow Graph : Directed graph showing flow of control.
2. Data Flow Fact : Information about program (e.g., variable is live, definition reaches a point, etc.).
3. Data Flow Equation : Mathematical relation used to compute IN and OUT sets of each block.
4. Meet Operator : Operator that combines information from different paths (e.g., $\cup$, $\cap$).
5. Transfer Function : Describes how a block transforms IN information to OUT information.

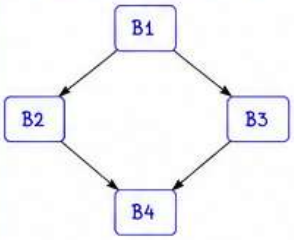
5. Global Data Flow Framework :



$$OUT[B] = F_B(IN[B])$$

$$IN[B] = \bigcap \text{ or } \bigcup \text{ of } OUT[P] \text{ for all predecessor } P$$

Example Flow Graph :



Edges :

B1 → B2 , B1 → B3

B2 → B4 , B3 → B4

Predecessors :

Pred(B1) = { }

Pred(B2) = { B1 }

Pred(B3) = { B1 }

Pred(B4) = { B2 , B3 }

6. Data Flow Equations (General Form) :

$$IN[B] = \bigcap OUT[P] \quad \text{for all predecessors } P \text{ of } B \text{ (may be } \cup \text{ also)}$$

$$OUT[B] = F_B(IN[B]) \quad (F_B \text{ is transfer function of block } B)$$

$\cap$ (intersection) is used in problems like available expressions.

$\cup$ (union) is used in problems like live variable analysis.

7. Types of Data Flow Analysis :

Type	Direction	Meet Operator	Examples
Forward	Entry → Exit	$\cup$ (Union)	Reaching Definitions, Available Expressions
Backward	Exit → Entry	$\cup$ (Union)	Live Variables, Very Busy Expressions
May Analysis	May be true	$\cup$ (Union)	Reaching Definitions, Live Variables
Must Analysis	Always true	$\cap$ (Intersection)	Available Expressions, Very Busy Expressions

8. Steps in Global Data Flow Analysis :

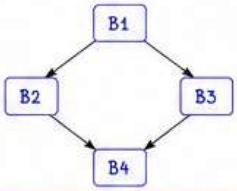
1. Construct the flow graph of the program.
2. Identify GEN and KILL sets for each block (if required).
3. Write the data flow equations for IN and OUT sets.
4. Initialize IN and OUT sets.
5. Apply the equations iteratively until no change occurs.
6. The final IN and OUT sets give the solution.

9. Example - Reaching Definitions (Forward May Analysis) :

Program :

```
B1 : a = 1
B2 : b = a
B3 : a = 2
B4 : c = a + b
```

Flow Graph :



Block	GEN[B]	KILL[B]	IN[B] (Initial)	OUT[B] (Initial)
B1	{ a=1 }	{ a=2 }	{ }	{ }
B2	{ b=a }	{ b=a }	{ }	{ }
B3	{ a=2 }	{ a=1 }	{ }	{ }
B4	{ c=a+b }	{ c=a+b }	{ }	{ }

Iterate using equations :

$$IN[B] = \bigcup OUT[P] \quad (P = \text{predecessors of } B)$$

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$$

After iterations, stable solution gives reaching definitions at each program point.

10. Summary :

- Global data flow analysis collects information across the whole program.
- It is based on flow graph, data flow facts, and transfer functions.
- Meet operators ($\cup$ or $\cap$) combine information from different paths.
- It is the foundation for many global code optimizations.

8. Code Improving Transformations

1. Introduction :

Code improving transformations are applied on the intermediate code to obtain an equivalent code that is easier and cheaper to execute. These transformations improve the speed, reduce memory usage and make the code more efficient.

Goal :
- ✓ Improve execution time
- ✓ Reduce code size
- ✓ Reduce memory access
- ✓ Improve register usage
- ✓ Simplify the program

2. Why Code Improving Transformations ?

- Removes unnecessary computations.
- Reduces the number of instructions.
- Helps in better register allocation.
- Improves cache performance.
- Leads to faster and smaller code.

3. Basic Idea :

Transform the code without changing the meaning (semantics) of the program.

If $S \Rightarrow T$

then S and T are equivalent ($S \equiv T$)

4. Types of Code Improving Transformations :

Type	Description
Algebraic Transformations	Use algebraic identities to simplify expressions.
Strength Reduction	Replace expensive operations with cheaper ones.
Common Subexpression Elimination	Compute common expressions once.
Constant Folding	Evaluate constant expressions at compile time.
Constant Propagation	Replace variables with constant values.
Copy Propagation	Replace copies by original variable.
Dead Code Elimination	Remove code that does not affect output.
Code Motion	Move code to reduce execution frequency.
Loop Transformations	Change loop structure for better performance.
Induction Variable Elimination	Replace induction variables with simple values.

5. Algebraic Transformations (Examples) :

① $x + 0 \Rightarrow x$

② $x * 1 \Rightarrow x$

③ $x * 0 \Rightarrow 0$

④ $x - x \Rightarrow 0$

⑤ $x * 2 \Rightarrow x + x$

⑥ $(x + y) + z \Rightarrow x + (y + z)$

Example :

Before :
```
t1 = a + 0
t2 = b * 1
t3 = c - c
```

After :
```
t1 = a
t2 = b
t3 = 0
```

Note :
Algebraic transformations do not change the meaning of the code, only the form.

6. Strength Reduction :

Replace costly operations (like *, /, power) with cheaper ones (like +, -, <<, >>, increment, decrement).

Example :

Before :
```
for (i = 0; i < n; i++){
  t1 = i * 8;
  arr[t1] = x;
}
```

After (Strength Reduced) :
```
t1 = 0;
for (i = 0; i < n; i++){
  arr[t1] = x;
  t1 = t1 + 8; // add instead of multiply
}
```

Explanation : Multiplication is replaced by addition.

7. Common Subexpression Elimination (CSE) :

If the same expression occurs more than once, compute it only once and reuse the result.

Example :

Before :
```
t1 = a + b
t2 = c * d
t3 = a + b
t4 = t1 * t2
t5 = (a + b) - e
```

After (CSE) :
```
t1 = a + b
t2 = c * d
t4 = t1 * t2
t5 = t1 - e    // use t1 again
```

Explanation : (a + b) computed once in t1 and used again.

8. Constant Folding :

Evaluate constant expressions at compile time.

Example :

Before :
```
t1 = 2 + 3 * 4
t2 = t1 + 5
t3 = 10 / 2
```

After (Constant Folded) :
```
t1 = 14
t2 = 19
t3 = 5
```

Explanation : 2 + 3 * 4 = 14, 14 + 5 = 19, 10 / 2 = 5

9. Constant Propagation :

If a variable is assigned a constant value, replace all its uses by that constant.

Example :

Before :
```
a = 10
b = a + 5
c = b * a
```

After (Constant Propagated) :
```
a = 10
b = 10 + 5
c = 15 * 10
c = 150    // further simplified
```

Explanation : Replace variable 'a' by constant 10.

10. Summary :

- Code improving transformations produce an equivalent but better code.
- They reduce execution time and memory usage.
- They do not change the meaning of the program.
- They are essential part of code optimization phase.
- Common transformations include algebraic, strength reduction, CSE, constant folding, constant propagation, etc.

9. Data Flow Analysis of Structure Flow Graph

1. Introduction :

A Structure Flow Graph (SFG) represents the control flow of a program. Data Flow Analysis collects information about the flow of data (values, variables, expressions) along the paths of an SFG. It is used to obtain information that is useful for program optimization and compiler decision making.

Goal :
- ✓ Collect data flow information.
- ✓ Combine information from all paths.
- ✓ Use information for optimization.

2. Why Data Flow Analysis ?

- Helps in detecting variable usage, live variables, available expressions, reaching definitions, etc.
- Helps in dead code elimination.
- Helps in register allocation and code optimization.
- Improves execution time and reduces memory usage.

3. Basic Concepts :

① Structure Flow Graph (SFG) : Directed graph showing control flow of structured program.

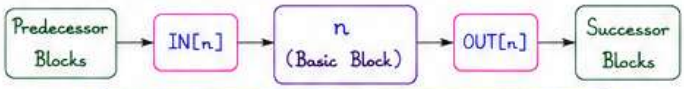
② Node : Represents basic block (sequence of statements).

③ Edge : Represents flow of control between blocks.

④ Data Flow Information : Information about data facts (like variable is live, expression is available, definition reaches a point, etc.).

⑤ Data Flow Equations : Mathematical equations used to compute data flow information at each node.

4. Data Flow Analysis Framework :



Transfer Function F_n : Describes how block n transforms IN[n] to OUT[n].

$$IN[n] = \bigcap (OUT[p] \text{ for all } p \in Pred(n))$$
$$OUT[n] = F_n (IN[n])$$

Where,

IN[n] : Information available at entry of block n.
OUT[n] : Information available at exit of block n.
Pred(n) : Set of predecessor blocks of n.
F_n : Transfer function of block n.

5. Types of Data Flow Problems :

Type	Direction
Forward Analysis	Information flows in direction of control flow (Entry ⟶ Exit).
Backward Analysis	Information flows opposite to control flow (Exit ⟶ Entry).
May Analysis	Information may be true on some paths.
Must Analysis	Information must be true on all paths.

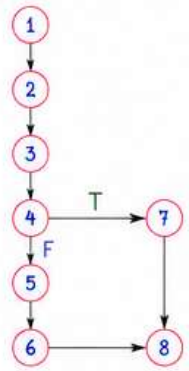
- Forward problems use Union (∪) to combine information.
- Backward problems use Union (∪) to combine information.
- Must problems use Intersection (∩) to combine information.

6. Example - Structure Flow Graph (SFG) :

Program :

```
1. a = b + c
2. d = e + f
3. g = a + d
4. if (g > 0) goto L1
5: h = g - 1
6. goto L2
L1: h = g + 1
L2: print(h)
```

Flow Graph :



Description :

```
1 : a = b + c
2 : d = e + f
3 : g = a + d
4 : if (g>0) goto L1
5 : h = g - 1
6 : goto L2
7 : h = g + 1
8 : print(h)
```

7. Data Flow Example - Reaching Definitions (Forward Must) :

Definition : A definition of a variable v at node n reaches a point p if there exists a path from n to p without any other definition of v on that path.

Let's find reaching definitions for variable 'g'.

Node	GEN[n]	KILL[n]	IN[n]	OUT[n]
1	{ }	{g}	{ }	{ }
2	{ }	{g}	{ }	{ }
3	{3}	{g}	{ }	{3}
4	{ }	{ }	{3}	{3}
5	{5}	{ }	{ }	{5}
6	{ }	{ }	{5}	{5}
7	{7}	{ }	{3}	{7}
8	{ }	{ }	{5,7}	{5,7}

Where, GEN[n] : Set of definitions of g generated in node n.
KILL[n] : Set of other definitions of g killed by node n.

Equations (Forward Must) :

- $IN[n] = \bigcup OUT[p]$ for all $p \in Pred(n)$
- $OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$
- (initially IN[Entry] = { }, OUT[Entry] = { })

8. Data Flow Example - Live Variables (Backward May) :

Definition : A variable is live at a point if its value may be used along some path from that point to program exit.

Node	USE[n]	DEF[n]	OUT[n]	IN[n]
8	{h}	{ }	{ }	{h}
7	{g}	{h}	{h}	{g}
6	{ }	{ }	{g}	{g}
5	{g}	{h}	{g}	{g}
4	{g}	{ }	{g}	{g}
3	{a,d}	{g}	{g}	{a,d}
2	{e,f}	{d}	{a,d}	{a,e,f}
1	{b,c}	{a}	{a,e,f}	{b,c,e,f}

Equations (Backward May) :

- $OUT[n] = \bigcup IN[s]$ for all $s \in Succ(n)$
- $IN[n] = USE[n] \cup (OUT[n] - DEF[n])$
- (initially OUT[Exit] = { }, IN[Exit] = { })

9. Steps to Perform Data Flow Analysis :

Important Point :
Iterative computation is necessary because change in one node may affect other nodes.

10. Summary :

- Data flow analysis works on structure flow graph of program.
- It collects and propagates data information across all paths.
- Forward problems use Union (∪) and backward problems also use Union (∪).
- Must problems use Intersection (∩) to get safe information.
- It is the foundation for many compiler optimizations.

10. Symbolic Debugging of Optimized Code

1. Introduction :

When a program is optimized, the original code is transformed into a more efficient form. Debugging such optimized code using actual values is difficult because variable values change or get eliminated. Symbolic debugging uses symbolic expressions instead of actual values to debug optimized code.

Goal :
- ✓ Detect errors in optimized code.
- ✓ Use symbolic values instead of actual values.
- ✓ Relate optimized code back to original code.

2. Why Symbolic Debugging ?

- Actual debugging becomes difficult after optimization.
- Variables may be renamed, eliminated or combined.
- It helps in tracing the flow of symbolic values.
- Helps in locating faults in optimized code.

3. Basic Idea :

Instead of using actual values, we represent each variable with a symbolic expression. As the program executes, these expressions are updated according to the operations performed.

4. Basic Concepts :

① Symbolic Value : An expression in terms of program inputs.

② Symbolic State : A set of symbolic values for all variables at a program point.

③ Symbolic Execution : Execute program using symbolic values instead of actual values.

④ Error Detection : If an expression becomes undefined, we detect a possible error.

5. Example Program :

Original Code :

```
1. a = b + c
2. d = a * e
3. if (d > 0) goto L1
4. f = d - 1
5. goto L2
L1: f = d + 1
L2: print(f)
```

⇨

Optimized Code :

```
1. t1 = b + c
2. t2 = t1 * e
3. if (t2 > 0) goto L1
4. t3 = t2 - 1
5. goto L2
L1: t3 = t2 + 1
L2: print(t3)
```

6. Symbolic Execution of Optimized Code :

Assume initial symbolic values :

b = B , c = C , e = E

Step	Statement	Symbolic Value / Condition
1	t1 = b + c	t1 = B + C
2	t2 = t1 * e	t2 = (B + C) * E
3	if (t2 > 0) goto L1	Condition : (B + C) * E > 0
	True (Goto L1)	t3 = t2 + 1 = (B + C) * E + 1
	False (Next line)	t3 = t2 - 1 = (B + C) * E - 1
6	print(t3)	Output = t3

Note :
- No actual numbers are used.
- Expressions tell us how values are related.
- Errors like division by zero, invalid operations can be detected symbolically.

7. Detecting Errors Symbolically :

Type of Error	Symbolic Detection
Division by Zero	If denominator expression becomes 0.
Null / Undefined	If expression becomes NULL or undefined.
Array Out of Bounds	If index expression is < 0 or >= size.
Overflow	If expression exceeds max range.
Invalid Condition	If condition expression is always true or always false.

8. Example - Division by Zero :

Code :

```
1. a = b - c
2. d = e / a
```

Symbolic Execution :

Step 1: a = B - C

Step 2: d = E / (B - C)

Error if : (B - C) = 0

So, if B = C then division by zero error will occur.

9. Mapping Optimized Code to Original Code :

- Optimization may rename variables and remove some statements.
- We maintain a mapping table between optimized temporary variables and original variables.
- This helps to trace errors back to original source code.

Mapping Table Example :

Optimized Variable	Mapped From (Original)
t1	b + c
t2	(b + c) * e
t3 (true path)	d + 1
t3 (false path)	d - 1

10. Advantages :

- ✓ Can debug optimized code without actual inputs.
- ✓ Works even when variables are eliminated or renamed.
- ✓ Helps in early detection of errors.
- ✓ Useful for automated debugging tools.

Benefit :
Makes debugging possible even after aggressive code optimization.

11. Limitations :

- ✖ Expressions may become too large and complicated.
- ✖ Path explosion problem - many possible execution paths.
- ✖ Cannot detect all run-time errors.
- ✖ May require sophisticated tools.

Note :
Works best for static errors and symbolic conditions.

12. Summary :

- Symbolic debugging uses symbolic expressions instead of actual values to debug optimized code.
- It tracks how symbolic values flow through the program.
- Helps in detecting errors like division by zero, null usage, out of bounds, overflow, etc.
- Requires mapping optimized code back to original code.
- Essential for verifying correctness of optimized programs.