|  |
| --- |
| Experiment No.10 |
| Case study on Optimization Techniques in Compilers. |
| Date of Performance:07/01/2025 |
| Date of Submission:17/01/2025 |

**Aim:** Case study on Optimization Techniques in Compilers.

**Objective:** To develop an understanding about the different optimization techniques used in compilers (e.g., constant folding, loop unrolling, dead code elimination) and how they improve the performance of generated code.

**Theory:**

**Case Study: Optimization Techniques in Compilers**

**Objective:**  
To develop an understanding of different optimization techniques used in compilers (e.g., constant folding, loop unrolling, dead code elimination) and how they improve the performance of generated code.

**Introduction**

Compiler optimization is a critical phase in the compilation process, aiming to improve the efficiency of the generated machine code without altering its semantics. Optimization techniques help reduce execution time, minimize memory usage, and enhance overall program performance. This case study explores key optimization techniques such as constant folding, loop unrolling, dead code elimination, inlining, strength reduction, and common subexpression elimination.

**Optimization Techniques**

**1. Constant Folding**

**Definition:** Constant folding is a compiler optimization technique that precomputes constant expressions at compile time instead of runtime.

**Example:**

int x = 5 \* 10; // Compiler replaces this with: int x = 50;

**Benefits:**

* Reduces computation overhead at runtime.
* Eliminates unnecessary arithmetic operations.

**2. Loop Unrolling**

**Definition:** Loop unrolling is a technique that increases a program's execution speed by reducing the overhead of loop control and increasing instruction-level parallelism.

**Example:** Before optimization:

for (int i = 0; i < 4; i++) {

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

}

After optimization:

a[0] = b[0] + c[0];

a[1] = b[1] + c[1];

a[2] = b[2] + c[2];

a[3] = b[3] + c[3];

**Benefits:**

* Reduces loop control overhead.
* Enhances parallel execution on modern processors.

**3. Dead Code Elimination**

**Definition:** Dead code elimination removes code statements that do not affect the program’s output.

**Example:** Before optimization:

int x = 10;

x = 20; // The previous assignment to x is never used.

After optimization:

int x = 20;

**Benefits:**

* Reduces code size.
* Improves execution speed by eliminating redundant operations.

**4. Function Inlining**

**Definition:** Function inlining replaces a function call with the actual function code to reduce function call overhead.

**Example:** Before optimization:

inline int square(int x) { return x \* x; }

int y = square(5);

After optimization:

int y = 5 \* 5;

**Benefits:**

* Reduces function call overhead.
* Improves execution speed by reducing stack operations.

**5. Strength Reduction**

**Definition:** Strength reduction replaces expensive operations with equivalent but less expensive ones.

**Example:** Before optimization:

int x = y \* 8;

After optimization:

int x = y << 3; // Uses bitwise shift instead of multiplication

**Benefits:**

* Reduces computation cost.
* Enhances execution speed.

**6. Common Subexpression Elimination (CSE)**

**Definition:** CSE identifies and eliminates repeated expressions by storing their results in variables.

**Example:** Before optimization:

int a = (x + y) \* (x + y);

After optimization:

int temp = x + y;

int a = temp \* temp;

**Benefits:**

* Reduces redundant calculations.
* Improves execution speed by avoiding unnecessary recomputation.

**Impact on Performance**

To evaluate the impact of these optimizations, consider a sample benchmark:

| **Optimization Technique** | **Execution Time Reduction** | **Memory Usage Reduction** |
| --- | --- | --- |
| Constant Folding | 10-15% | Negligible |
| Loop Unrolling | 20-30% | Slight increase |
| Dead Code Elimination | 5-10% | Moderate reduction |
| Function Inlining | 15-25% | Possible increase |
| Strength Reduction | 5-10% | No effect |
| Common Subexpression Elimination | 10-20% | Slight reduction |

**Conclusion:**

Optimization techniques in compilers significantly enhance program performance by reducing redundant computations, improving parallel execution, and eliminating unnecessary code. Modern compilers employ a combination of these techniques to generate highly efficient machine code, ensuring better resource utilization and faster execution times.