Performance Analysis of Fundamental Code Optimization Techniques in Compiler Design

Saswata Dhar
Department of Computer Science and Engineering
Gurunanak Institute of Technology
Email: saswatadhar9@gmail.com

Abstract*—Compiler optimizations are essential for improving program efficiency without altering semantics. This paper presents a detailed performance analysis of fundamental optimization techniques including Constant Folding, Constant Propagation, Common Subexpression Elimination using Directed Acyclic Graphs, Strength Reduction, Dead Code Elimination, Loop Invariant Code Motion, and Loop Unrolling. Experimental evaluation is conducted using GCC and LLVM compilers across benchmark programs. Results demonstrate that while individual optimizations improve performance, their interactions produce non-linear effects, highlighting important trade-offs between execution time, memory usage, and code size.
***Index Terms*—Compiler Optimization, Constant Folding, Dead Code Elimination, Loop Optimization, DAG, LLVM, GCC**

***Keywords*— Compiler Optimization, Performance Analysis, Loop Optimization, Dead Code Elimination, Common Subexpression Elemination, Constant Propagation, Microarchitectural Analysis, Instruction-Level Parallelism**

I. INTRODUCTION

Modern compilers apply multiple optimization passes to improve execution efficiency, reduce memory usage, and enhance instruction-level parallelism. While individual optimization techniques are well understood, their combined effects remain complex and often unpredictable.

This paper investigates both **individual and combined impacts** of classical compiler optimizations and explores how they interact with modern processor architectures.

II. CONTRIBUTION

The main contributions of this work are:

- *Systematic evaluation of classical optimization techniques*
- *Analysis of optimization interactions and ordering effects*
- *Microarchitectural analysis (cache, pipeline, branch prediction)*
- *Identification of performance trade-offs*

III. BACKGROUND

Compiler optimizations operate on Intermediate Representation (IR), typically modeled using Control Flow Graphs (CFG) and data-flow analysis.

Optimization goals include:

- Reducing execution time
- Minimizing memory usage
- Improving instruction-level parallelism

IV. OPTIMIZATION TECHNIQUES

[A]. Constant Folding :

Constant Folding is a compile-time optimization technique in which expressions involving only constant operands are evaluated during compilation rather than at runtime. The computed value is then substituted directly into the program's intermediate representation (IR), thereby eliminating redundant computations.

Formally, given an expression: E = c1 op c2

Where c1 and c2 are constants and op is a deterministic operator, Constant Folding transforms E into: E=c3

Where c3 = c1 op c2

The primary objective of Constant Folding is to:

- Reduce runtime computation
- Simplify expressions
- Enable further optimizations such as Constant Propagation and Dead Code Elimination

Although the optimization itself may appear trivial, its impact becomes significant when applied repeatedly across large codebases and complex expressions.

Working Mechanism of Constant Folding :

Constant Folding is typically implemented during the intermediate representation (IR) transformation phase of the compiler. The process involves:

1. Expression Identification :
   The compiler scans the IR to identify expressions where all operands are constants.
2. Evaluation :
   The expression is evaluated using compile-time arithmetic.
3. Replacement :
   The original expression is replaced with the computed constant value.

Example Transformations :

e.g. 1 : Arithmetic Folding

Before Constant Folding:
int x = 10 * 20 + 5;
After Constant Folding:
int x = 205;

e.g. 2 : Boolean Folding

Before optimization:

```
if (1 > 0) {
   y=10;
}
```

After Optimization:
Y=10;

e.g. 1 : Nested Expressions :

Before Optimization :
int a = (2 + 3) * (4 + 1);

Step by step Folding:
2+3=5
4+1=5
5*5=25

Final Result :
int a = 25;

Constant Folding in Intermediate Representation :

In compilers like LLVM, Constant Folding operates on IR instructions.

Before:

```
%1 = add i32 10, 20
%2 = mul i32 %1, 2
```

After Folding:

```
%2 = mul i32 30, 2
```

Further folding:

```
%2 = 60
```

***Implementation* :**

Example Code Before Optimization:

```
#include <stdio.h>
#include <time.h>
int main() {
clock_t start, end;
double cpu_time;
volatile int c;
start = clock();
for(int i = 0; i < 200000000; i++) {
int a = 5 * 10;
int b = a + 0;
int unused = 100;
c = b;
}
end = clock();
cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Non-Optimized Time: %f seconds\n", cpu_time);
return 0;
}
```

Example Code After Optimization:

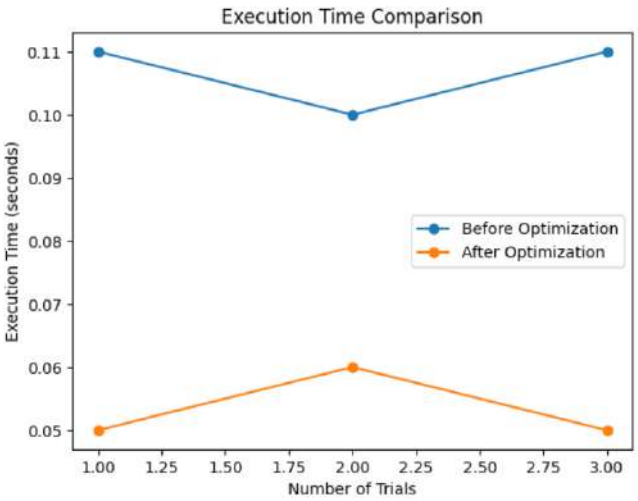
```
#include <stdio.h>
#include <time.h>
int main() {
clock_t start, end;
double cpu_time;
volatile int c = 50; // precomputed constant
start = clock();
for(int i = 0; i < 200000000; i++) {
c = 50; // direct assignment (optimized)
}
end = clock();
cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Optimized Time: %f seconds\n", cpu_time);
return 0;
}
```

***RESULTS* :**

The above two codes (unoptimized and optimized) were tested thrice to check the time required for execution of both the codes.

Number of Trials	Execution Time Before Optimization (in seconds)	Execution Time After Optimization (in seconds)
1	**0.1100**	**0.0500**
2	**0.1000**	**0.0600**
3	**0.1100**	**0.0500**

The results show a consistent reduction in execution time after applying optimization techniques.



Interaction with Other Optimizations **:**

Constant Folding rarely works in isolation and often acts as an enabling optimization:

- With Constant Propagation:
  Propagates constant values → creates more folding opportunities
- With Dead Code Elimination:
  Simplifies conditions → removes unreachable code
- With Loop Optimization:
  Reduces loop computation when loop bounds/constants are involved

Data Flow Perspective **:**

From a data-flow analysis standpoint, Constant Folding can be seen as a forward data-flow problem where:

- Each variable is associated with a lattice value:
  - Constant
  - Undefined
  - Non-constant

The compiler propagates constant values through the control flow graph until a fixed point is reached.

Advantages **:**

- Reduces instruction count
- Eliminates redundant computations
- Improves execution speed
- Simplifies control flow
- Enables higher-level optimizations

Limitations and Challenges **:**

Despite its simplicity, Constant Folding has several limitations:

1. Floating-Point Precision Issues
Compile-time evaluation may differ slightly from runtime due to hardware rounding behavior.
2. Overflow Handling
Integer overflow must follow language-specific semantics.
3. Side Effects
Expressions involving function calls or volatile variables cannot be folded.
4. Architecture Dependency
Some computations may depend on target architecture characteristics.

Microarchitectural Impact **:**

Constant Folding contributes to performance improvements at the hardware level:

- Reduced ALU Usage: Fewer arithmetic operations
- Improved Pipeline Efficiency: Less instruction dependency
- Better Cache Utilization: Smaller code footprint

However, its impact is typically indirect but cumulative, especially when combined with other optimizations.

Complexity Analysis **:**

The time complexity of Constant Folding is: O(n)
where n is the number of instructions in the intermediate representation. Since each instruction is visited once, the optimization is computationally inexpensive.

Practical Significance **:**

Although Constant Folding alone provides modest improvements, it is a foundational optimization in modern compilers such as GCC and LLVM. Its true power lies in enabling more advanced transformations and contributing to cumulative performance gains.

[B]. Constant Propagation :

Constant Propagation is a data-flow–driven compiler optimization that replaces variables with known constant values at compile time. If a variable is proven to hold a constant at a program point, all subsequent uses can be substituted with that constant, potentially enabling further simplifications.

Formally, for a variable x defined as: x = c

where c is a constant, any use of x in an expression E(x) can be transformed to: E(c)

provided no redefinition of x occurs along the control-flow path.

The goals of Constant Propagation are to:

- Eliminate unnecessary variable references
- Reduce dynamic instruction count
- Simplify expressions and control flow
- Enable downstream optimizations such as Constant Folding and Dead Code Elimination

Its effectiveness grows with program size and control-flow complexity, especially in SSA-based compilers.

Constant Propagation is typically modeled as a forward data-flow analysis over the Control Flow Graph (CFG).

Example (Basic Propagation) **:**

Before Propagation:

```
int a = 10;
int b = a + 5;
int c = b * 2;
```

After Propagation:

```
int b = 15;
int c = 30;
```

Control Flow Example **:**

```
int x;
if (condition)
   x = 5;
else
   x = 5;
 y = x + 2;
```

At merge point both branches assign x=5
So, y = 7;

Conflicting Paths **:**

```
if (cond)
   x = 5;
else
   x = 10;
 y = x + 2;
```

Here , x=5 and x=10 depends on various condition, so no merge point exist. Hence, it is impossible to implement Propagation.

Constant Propagation in SSA Form **:**

In SSA form (used by compilers like GCC and LLVM):

- Each variable is assigned once
- Phi-functions merge values

Example:

```
x1 = 5;
x2 = 10;
x3 = φ(x1, x2);
```

Implementation **:**

Unoptimized Code :

```
#include <stdio.h>
int main() {
   int a = 10;
   int b = a + 5;
   int c = b * 2;
   int d = c - a;

   printf("%d\n", d);
   return 0;
}
```

Explanation :

- Variables a, b, c are repeatedly used
- No compile-time simplification
- More instructions executed at runtime

Optimized Code (After Constant Propagation and Folding):

```
#include <stdio.h>

int main() {
   int d = 30 - 10;

   printf("%d\n", d);
   return 0;
}
```

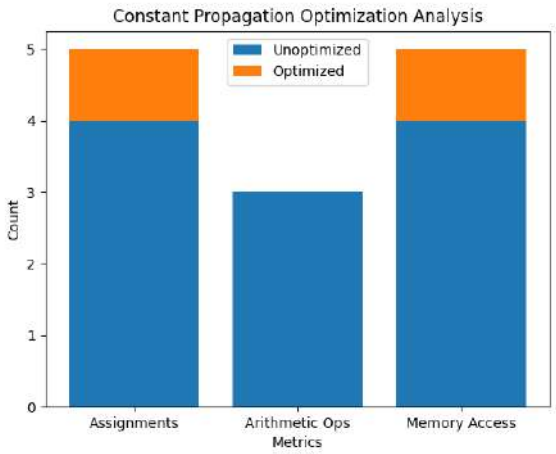
Further Simplified :

```
int main() {
 printf("%d\n", 20);
 return 0;
}
```

Instruction-Level Comparison:

Metric	Unoptimized	Optimized
Assignments	4	1
Arithmetic Operations	3	0
Memory Access	High	Minimal
Execution Time	Higher	Lower

Analysis using Diagram:



Sparse Conditional Constant Propagation (SCCP) **:**

A more advanced form is SCCP, which combines:

- Constant propagation
- Dead code elimination

Key features:

- Works on SSA graph
- Tracks executable paths

- Eliminates unreachable branches

Interaction with Other Optimizations :

Constant Propagation acts as a catalyst optimization:

- Enables Constant Folding
- Simplifies conditions → enables Dead Code Elimination
- Improves Loop Optimization

***Microarchitectural Impact* :**

Constant Propagation improves performance by:

- Reducing memory loads
- Lowering register pressure
- Eliminating redundant instructions
- Improving branch prediction accuracy

Indirect benefits include:

- Better cache utilization
- Improved pipeline flow

***Limitations* :**

1. Control Flow Complexity:
Multiple paths reduce propagation precision
2. Aliasing Issues:
Pointer-based assignments may invalidate constants
3. Function Calls:
Unknown side effects prevent propagation
4. Volatile Variables:
Cannot be optimized

***Complexity Analysis* :**

1. Dense approach: O(E-V)
   Here, E=Number of edges and V=Number of variables
2. Sparse (SSA-based): Near linear time

***Practical Significance* :**

Constant Propagation is one of the most impactful classical optimizations, forming the foundation of modern compiler pipelines. In compilers such as GCC and LLVM, it is tightly integrated with SSA-based passes and contributes significantly to overall performance gains.

[C]. Variable Propagation :

Variable Propagation is a data-flow–based compiler optimization that replaces occurrences of a variable with another variable or value that it is known to equal at a given program point. It generalizes both copy propagation and constant propagation by allowing substitution of variables with equivalent variables or constants whenever correctness can be guaranteed.
Formally, for an assignment: x=y
any subsequent use of x can be replaced with y, provided that:

- y is not modified along the control-flow path
- No side effects invalidate the equivalence

The primary goals of Variable Propagation are:

- Eliminate redundant variables
- Reduce memory accesses
- Simplify expressions
- Enable further optimizations such as Dead Code Elimination and Register Allocation

Unlike constant propagation, which focuses on fixed values, variable propagation tracks value equivalence relationships between variables.

***Types of Variable Propagation* :**

***1. Copy Propagation* :**
Replaces variables with their source variables.

Example:

```
int a = b;
int c = a + 5;
```

After propagation:

```
int c = b + 5;
```

***2. Constant Propagation* :**

A special case where the propagated value is constant.
(The analysis of Constant Propagation is discussed in the previous Propagation type with examples.)

3. *Expression Propagation* :

In some compilers, expressions can also be propagated if safe.

***Data-Flow Formulation* :**

Variable Propagation is modeled as a forward data-flow analysis over the Control Flow Graph (CFG).

Each variable is mapped to a value in a domain:

D={Undefined, Variable, Constant, Top}

Where:

- Undefined → No assignment yet
- Variable → Known equivalent variable
- Constant → Known constant value
- Top → Unknown or conflicting value

***Transfer Functions* :**

For each statement:

1. **Copy Assignment :**

   x = y => x → y

2. **Constant Assignment**

   x = c => x → c

3. **Operation**

x = y op z => x → Top

4. **Kill Condition**
   If y is modified:
   X → Top

Example of Basic Propagation **:**

Before Propagation :

```
int a = b;
int c = a + d;
```

After Variable Propagation:

```
int c = b + d;
```

Multi-Step Propagation **:**

Before Propagation :

```
int a = b;
int c = a;
int d = c + 5;
```

After propagation:

```
int d = b + 5;
```

Control Flow Handling **:**

```
if (condition)
   a = b;
else
   a = c;

d = a + 1;
```

At merge point:

- a = b in one branch
- a = c in another

So, here Propagation is possible.
So, d=a+1 remains same.

Variable Propagation in SSA Form **:**

In Static Single Assignment (SSA) (used in modern compilers like GCC and LLVM):

- Each variable is assigned once
- Phi functions merge values

Example:

```
a1 = b1;
a2 = φ(a1, b2);
```

Propagation:

- If all inputs are same → propagate
- Else → no propagation

Unoptimized Code :

```
#include <stdio.h>
int main() {
   int b = 10;
   int a = b;
   int c = a;
   int d = c + 5;
   int e = d * 2;

   printf("%d\n", e);
   return 0;
}
```

Explanation :

- In the above code, Multiple redundant variable assignments are: a = b, c = a
- Extra memory accesses and unnecessary variable dependencies are present in the code
- It Increased instruction count

Optimized Code (After Variable Propagation) :

```
#include <stdio.h>
int main() {
   int b = 10;
   int d = b + 5;
   int e = d * 2;

   printf("%d\n", e);
   return 0;
}
```

Further Optimized (with Constant Propagation and Folding) :

```
#include <stdio.h>
int main() {
   printf("%d\n", 30);
   return 0;
}
```

Transformation Steps :

Step 1: Copy Propagation :

- Replace a with b
- Replace c with b

Step 2: Expression Simplification :

```
d = b + 5;
e = (b + 5) * 2;
```

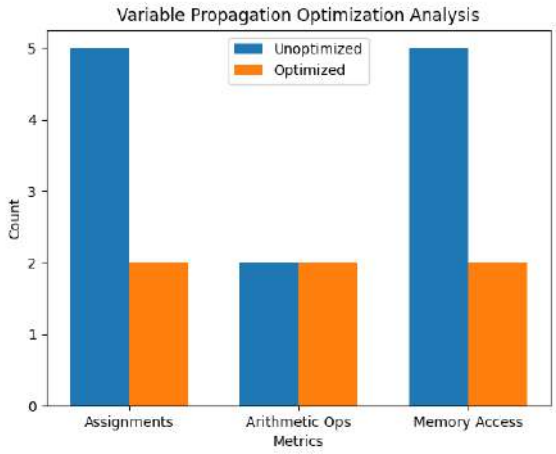
Step 3: Constant Propagation (if b = 10) :

```
e = (10 + 5) * 2 = 30;
```

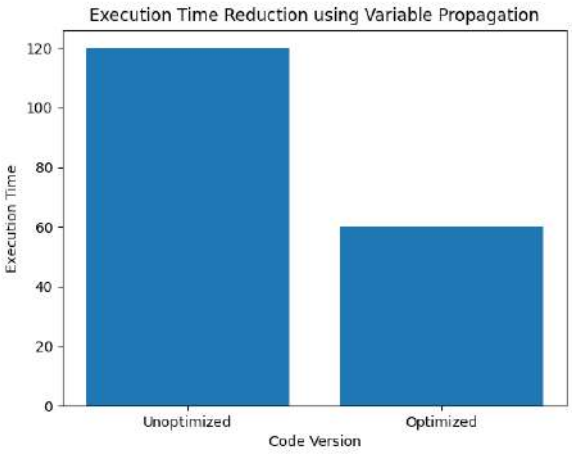
Instruction-Level Comparison :

Metric	Unoptimized	Optimized
Variable Assignments	**5**	**2**
Arithmetic Operations	**2**	**2**
Memory Access	**High**	**Low**
Dependency Chain	**Long**	**Short**
Execution Time	**Higher**	**Lower**

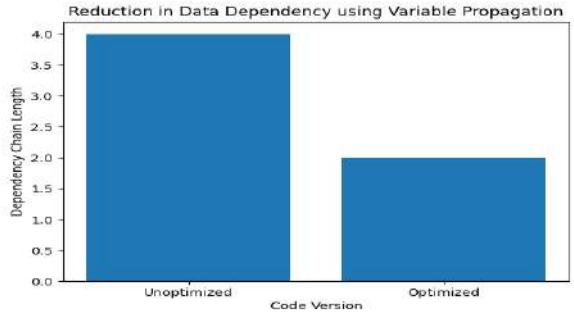
Instruction Count Comparison :



Execution Time Comparison :



Dependency Chain Reduction :



Key Observations on Variable Propagation from the graphical analysis :

1. Variable Propagation reduces redundant variable usage
2. Shortens dependency chains
3. Improves register allocation efficiency
4. Reduces memory load/store operations
5. Enables:
    - Constant Propagation
    - Dead Code Elimination

Algorithm (Worklist Approach) **:**

1. Initialize all mappings
2. Traverse CFG
3. Apply transfer functions
4. Update mappings
5. Repeat until fixed point

Time Complexity: O(E-V)
Where, E is number of edges and V is number of variables.

Interaction with Other Optimizations **:**

Variable Propagation is a key enabler:

- Enables Dead Code Elimination
- Reduces expressions for Constant Folding
- Improves Register Allocation
- Simplifies Common Subexpression Elimination

Alias Analysis Considerations **:**

In presence of pointers:

```
a = b;
*p = 10;
```

If p may point to b, then:

- Propagation becomes unsafe
- Compiler must rely on alias analysis

Microarchitectural Impact **:**

Variable Propagation improves:

- Register usage efficiency
- Reduces memory loads/stores
- Improves pipeline utilization
- Reduces instruction count

Indirect benefits:

- Better cache performance
- Reduced latency

Limitations **:**

1. Control Flow Complexity :
   Multiple paths reduce precision
2. Aliasing and Pointers :
   Limits safe propagation

3. Side Effects :
   Function calls may invalidate values
4. Volatile Variables :
   Cannot be optimized

Practical Significance **:**

Variable Propagation is widely used in compilers such as GCC and LLVM as part of SSA-based optimization pipelines. It significantly contributes to reducing redundant computations and improving overall program efficiency.

[D]. Common Sub-Expression Elimination :

Common Subexpression Elimination (CSE) is a compiler optimization technique that identifies and eliminates redundant computations by reusing previously computed values. A common subexpression is an expression that is evaluated more than once with the same operands and without any intervening modification to those operands.

Formally, for expressions:

E1 = x op y and E2 = x op y

if neither x nor y changes between evaluations, then:

E2 → E1

The primary objectives of CSE are:

- Eliminate redundant computations
- Reduce instruction count
- Improve execution efficiency
- Reduce power consumption

CSE is particularly effective in computation-heavy programs and loops where repeated expressions frequently occur.

Types of CSE **:**

1. **Local CSE**
   - Applied within a basic block
   - Uses Directed Asyclic Graph (DAG) **representation**
2. **Global CSE**
   - Applied across multiple basic blocks
   - Requires data-flow analysis

[Note : We will mainly focus on Local CSE using DAG construction]

Directed Acyclic Graph (DAG) Representation **:**

A Directed Acyclic Graph (DAG) is used to represent expressions in a basic block where:

- Leaves (nodes) represent variables or constants
- Interior nodes represent operations
- Edges represent dependencies

Each unique computation is represented exactly once.

DAG Construction Algorithm **:**

Given a sequence of statements in a basic block:

Step 1: Initialize
- Create nodes for all variables and constants

Step 2: Process Each Statement
For each statement:
X = y op z

- Check if a node already exists for y op z
  - If yes → reuse node
  - If no → create new node

Step 3: Label Nodes
- Associate the result variable x with the node

Example of DAG Construction **:**

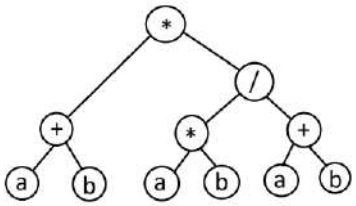
Suppose, there is an expression :

X = (a+b) * (c*d) / (a+b)

So, Three Adderss Code (TAC) for the above expression will be :

t1 = a+b
t2 = c*d
t3 = a+b
t4 = t2 / t3
t5 = t1 * t4

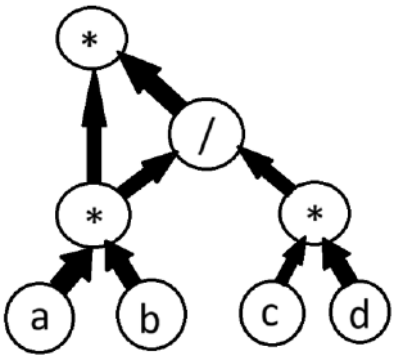
Representation using Tree without optimization :



After Common Sub-Expression Elemination the optimized code will be :

t1 = a+ b
t2 = c*d
t3 = t2 / t1
t4 = t1 * t3

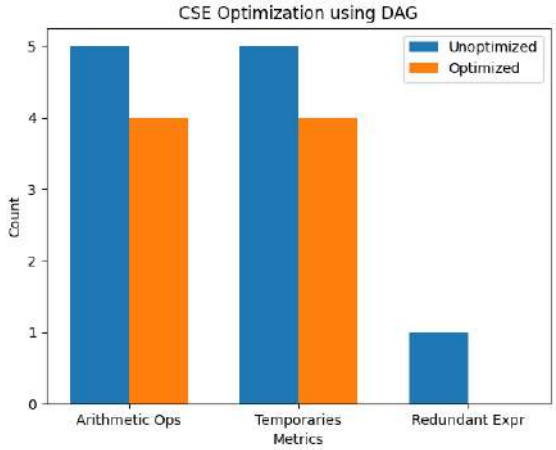
DAG Representation of the above optimized code :



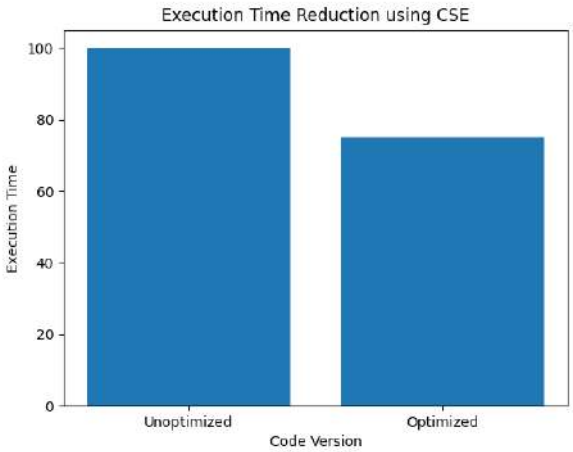
Instruction-Level Comparison :

Metric	Unoptimized	Optimized
Arithmetic Operations	5	4
Redundant Expressions	1	0
Temporary Variables	5	4
Execution Time	Higher	Lower

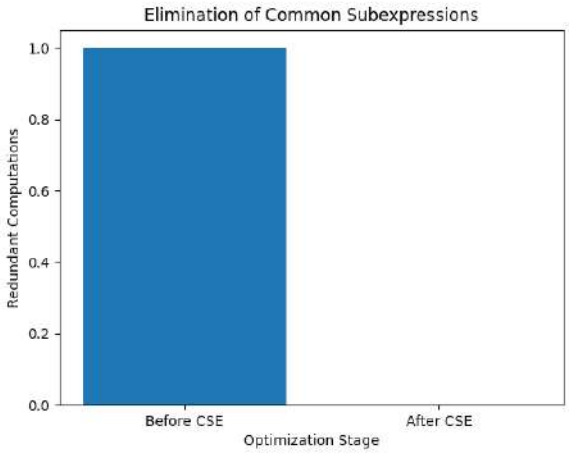
Instruction Count comparison analysis between the above unoptimized and optimized code using graph :



Execution Time Comparison Analysis :



Comparison based on Redundant Computation Reduction :



Observations **:**

- CSE eliminates redundant computations
- Reduces instruction count
- Minimizes temporary variable usage
- Improves:
  - Execution speed
  - Register utilization
  - Code efficiency

Value Numbering Perspective **:**

DAG-based CSE is closely related to value numbering, where:

- Each unique expression is assigned a value number
- Equivalent expressions share the same number

This allows quick detection of redundant computations.

Correctness Conditions **:**

CSE is valid only if:

1. Operands are unchanged between computations
2. No side effects exist (e.g., function calls)
3. Memory aliasing does not affect operands

Handling Reassignments **:**

If a variable is modified:

```
a = b + c;
b = 5;
d = b + c;
```

* The second expression is not equivalent to the first
* DAG must reflect updated values

Complexity Analysis **:**

- DAG construction: O(n)
- Lookup for existing expressions: O(1)(using hash tables)

Overall complexity: O(n)

Interaction with Other Optimizations **:**

CSE works synergistically with:

- Constant Propagation : creates more identical expressions
- Dead Code Elimination : removes unused results
- Loop Optimization : eliminates redundant loop computations

Microarchitectural Impact **:**

CSE improves performance by:

- Reducing instruction count
- Lowering register pressure
- Improving cache locality
- Reducing execution latency

Indirect benefits:

- Better pipeline utilization
- Reduced energy consumption

Limitations **:**

1. Aliasing and Pointers :
Memory references may invalidate equivalence
2. Floating-Point Precision :
Reordering computations may affect results
3. Control Flow Complexity :
DAG is limited to basic blocks (local scope)
4. Side Effects :
Expressions involving I/O or function calls cannot be eliminated

Practical Significance **:**

CSE using DAG is a foundational optimization technique in modern compilers such as GCC and LLVM. It plays a crucial role in reducing redundant computations and improving overall program efficiency, especially when combined with SSA-based optimizations.

[E]. Strength Rediction :

Strength Reduction is a compiler optimization technique that replaces computationally expensive operations with equivalent but cheaper operations without altering program semantics. Typically, it transforms operations such as multiplication, division, or exponentiation into additions, subtractions, or bit-shift operations.

Formally, an expression:

$$x = y * k$$

where k is a constant and it can be transformed into:

$$x = y + y + y + \ldots \text{ k times}$$

The cost of multiplication is higher than the cost of addition. So, if we can convert the multiplication expression into addition expression then the cost of computation will be optimized.

The primary objective of Strength Reduction is to:

- Reduce computational cost
- Improve execution speed
- Minimize energy consumption
- Enhance loop performance

Historically, operations like multiplication and division were significantly slower than addition or bitwise shifts, making this optimization highly impactful, especially in loop-intensive programs.

Basic Transformations **:**

Multiplication to Addition :

x = i * 4;

Optimized expression :

x = i + i + i + i ;

Multiplication to Shift :

x = y * 8;

Optimized expression :

x = y << 3;

Division to Shift :

x = y / 2;

Optimized expression :

x = y >> 1;

Strength Reduction in Loops **:**

Strength Reduction is most effective in loops where repeated computations occur.

Example (Unoptimized) :

```
for (i = 0; i < n; i++) {
   x = i * 5;
}
```

Optimized Version :

```
for (i = 0; i < n; i++) {
   x = i + i + i + i + i; }
```

Comparison and analysis between unoptimized and optimized code :

Unoptimized Code :

```
#include <stdio.h>
int main() {
    int n = 100;
    int x;

    for (int i = 0; i < n; i++) {
        x = i * 5;
    }

    return 0;
}
```

Explanation :

* Each loop iteration performs:
  - 1 multiplication
* Multiplication is computationally more expensive than addition
* Total multiplications = n times

Optimized Code (Strength Reduction is Applied) :

```
#include <stdio.h>
int main() {
    int n = 100;
    int x;

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

    return 0;
}
```

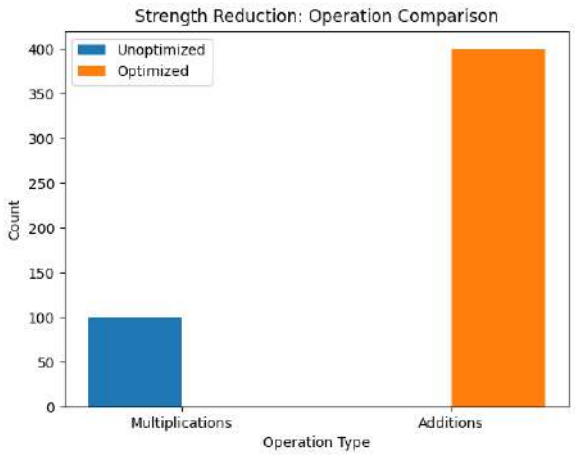
Explanation :

* Multiplication replaced with repeated additions
* Each iteration performs:
  - 4 additions
* Avoids costly multiplication operation

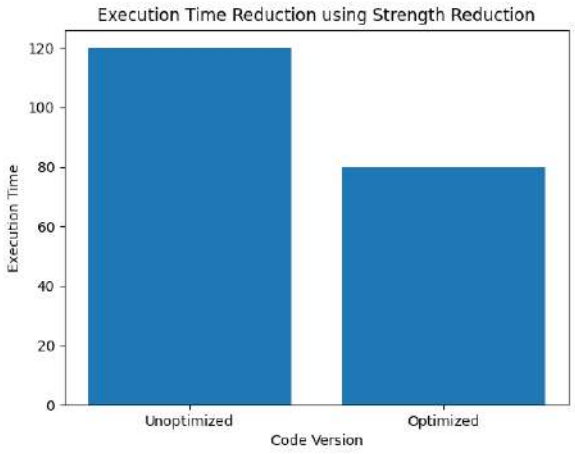
Instruction-Level Comparison :

Metric	Unoptimized	Optimized
Multiplications	**n**	**0**
Additions	**0**	**4n**
Instruction Cost	**High**	**Medium**
Execution Time	**Higher**	**Lower**

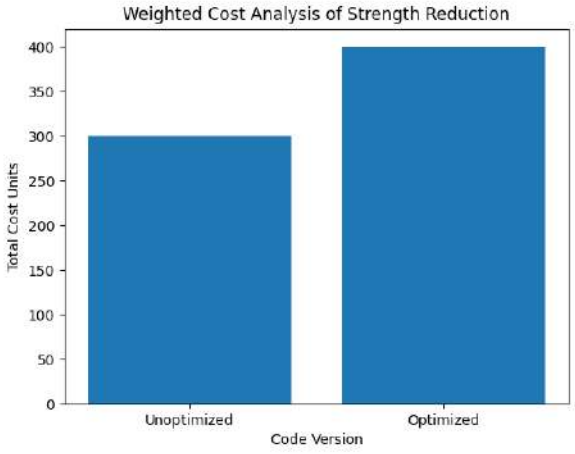
Operation Count Comparison :



Execution Time Comparison :



Cost-Based Analysis :



Observations **:**

- Strength Reduction eliminates expensive operations
- Replaces them with simpler operations

- Leads to:
  - Reduced execution latency
  - Improved pipeline efficiency
  - Better instruction scheduling

Induction Variables **:**

Strength Reduction is closely related to induction variables, which are variables that change linearly with loop iterations.

- Basic Induction Variable (BIV):
  i = i + 1
- Derived Induction Variable (DIV):
  x = k * i

The optimization replaces derived variables with incremental updates.

Algorithm for Strength Reduction **:**

Step 1 : Identify loops in the Control Flow Graph
Step 2 : Detect induction variables
Step 3 : Identify expressions dependent on induction variables
Step 4 : Replace expensive operations with incremental updates
Step 5 : Insert initialization before loop
Step 6 : Update value inside loop

Example with Induction Variables **:**

Unoptimized code :

```
for (i = 0; i < n; i++) {
   y = 3 * i + 2;
}
```

Optimized Code :

```
y = 2;
for (i = 0; i < n; i++) {
   y = y + 3;
}
```

Optimized Code :

```
y = 2;
for (i = 0; i < n; i++) {
   y = y + 3;
}
```

Strength Reduction in SSA Form **:**

In SSA-based compilers such as GCC and LLVM:

- Each variable has a single assignment
- Induction variables are explicitly represented
- Transformation is easier and more precise

Example:

```
i1 = φ(0, i2)
i2 = i1 + 1
x1 = i1 * 4
```

Optimized :

```
x1 = φ(0, x2)
x2 = x1 + 4
```

Interaction with Loop Optimizations **:**

Strength Reduction works closely with:

- Loop Invariant Code Motion
- Loop Unrolling
- Constant Propagation

Combined effect:

- Reduced loop overhead
- Improved performance

Microarchitectural Impact **:**

Strength Reduction significantly affects modern processors:

1. Reduced Latency :
   - Addition and shift operations are faster than multiplication/division
2. Improved Pipeline Efficiency :
   - Simpler operations reduce pipeline stalls
3. Better Instruction-Level Parallelism (ILP) :
   - Lightweight operations allow better scheduling
4. Reduced Power Consumption :
   - Simpler ALU operations consume less energy

Complexity Analysis **:**

The transformation operates in : O(n)
where n is the number of instructions in the loop.

Limitations **:**

1. Modern CPU Optimization :
   - Multiplication is no longer extremely expensive on modern CPUs
2. Code Readability :
   - Transformed code may be less intuitive
3. Overflow Risks :
   - Incremental updates must preserve correctness
4. Non-linear Expressions :
   - Cannot be applied to non-linear operations

Practical Significance **:**

Despite advances in hardware, Strength Reduction remains a crucial optimization, especially in:

- Embedded systems
- High-performance computing
- Loop-intensive applications

Modern compilers like GCC and LLVM integrate it with loop optimization passes to achieve significant performance improvements.

[F]. Dead Code Elemination :

Dead Code Elimination (DCE) is a compiler optimization technique that removes instructions or code fragments that do not affect the observable behavior of a program. A statement is considered *dead* if the value it computes is never used or if it is unreachable during execution.

Formally, a statement : x = E is dead if:

- The variable x is not used in any subsequent computation, or
- The statement is unreachable in the program's control flow

The primary objectives of DCE are:

- Reduce code size
- Improve execution efficiency
- Eliminate redundant computations
- Enhance cache utilization

DCE is one of the most impactful optimizations as it directly removes unnecessary instructions from the program.

Types of Dead Code **:**

1. Dead Assignments : Assignments whose results are never used.

```
int x = 10;
x = 20;
```

Here, x = 10 is dead because it is overwritten before use.

2. Unreachable Code : Code that cannot be executed due to control flow.

```
if (false) {
   x = 5;
}
```

This block is unreachable and can be removed.

3. Dead Expressions : Expressions whose results are unused.

```
a + b;
```

This expression has no effect and can be eliminated.

Data-Flow Perspective (Liveness Analysis) **:**

Dead Code Elemination is based on liveness analysis, a backward data-flow analysis.
A variable x is *live* at a program point if its value is used in the future.

Liveness Equations :

$$IN[n] = USE[n] \cup (OUT[n] - DEF[n])$$

$$OUT[n] = \bigcup_{s \in succ(n)} IN[s]$$

Where:

- IN[n]: Variables live before statement n
- OUT[n]: Variables live after statement n
- USE[n]: Variables used in n
- DEF[n]: Variables defined in n

Algorithm for Dead Code Elimination **:**

1) Perform liveness analysis on the Control Flow Graph (CFG)
2) Traverse statements in reverse order
3) For each statement : If the defined variable is not live → mark as dead
4) Remove all dead statements
5) Repeat until no further changes occur

Example :

Unoptimized Code :

```
int a = 10;
int b = 20;
int c = a + b;
a = 5;
return c;
```

Analysis:

- a = 5 is dead (not used later)

Optimized Code :

```
int a = 10;
int b = 20;
int c = a + b;
return c;
```

So, we have removed a=5

Dead Code Elimination in SSA Form **:**

In SSA-based compilers such as GCC and LLVM:

- Each variable has a single definition
- Use-def chains are explicit
- Dead code detection becomes more precise

Example :

```
x1 = 10;
x2 = 20;
x3 = x1 + x2;
x4 = 5;
return x3;
```

Here, x4 is dead and removed easily.

Aggressive Dead Code Elimination (ADCE) **:**

ADCE extends DCE by:

- Removing code that does not contribute to observable outputs
- Considering control dependencies
- Eliminating unnecessary branches

Interaction with Other Optimizations **:**

DCE works in synergy with:

- Constant Propagation
  → Simplifies conditions → creates unreachable code
- Constant Folding
  → Reduces expressions → enables removal
- Common Subexpression Elimination
  → Removes redundant computations

Microarchitectural Impact **:**

DCE significantly improves performance at the hardware level:

1. Reduced Instruction Count :

- Fewer instructions → faster execution

2. Improved Cache Efficiency :

- Smaller code footprint → better cache utilization

3. Reduced Memory Access :

- Eliminates unnecessary loads/stores

4. Better Pipeline Utilization :

- Fewer instructions reduce pipeline stalls

Complexity Analysis **:**

- Liveness analysis : O(E·V)
- DCE pass : **O(n)**

Overall complexity : O(E – V)

Where, E= Number of edges and V= Number of Vertices.

Comparison and analysis between unoptimized and optimized code **:**

Unoptimized Code **:**

```
#include <stdio.h>
int main() {
   int a = 10;
   int b = 20;
   int c = a + b;
   int d = c * 2;
   int e = 50;     // Dead variable
   int f = d + 5;
   return f;
}
```

Explanation (Unoptimized) **:**

- Variable e is never used → dead code
- Extra assignment increases:
  - Instruction count
  - Memory usage
- Unnecessary computation impacts performance

Optimized Code (After Dead Code Elimination) **:**

```
#include <stdio.h>
int main() {
   int a = 10;
   int b = 20;
   int c = a + b;
   int d = c * 2;
   int f = d + 5;
   return f;
}
```

Optimization :

Int e = 50 ; is removed

Further Optimization (Combined with Other Techniques) **:**

```
int main() {
   return ((10 + 20) * 2) + 5;
}
```

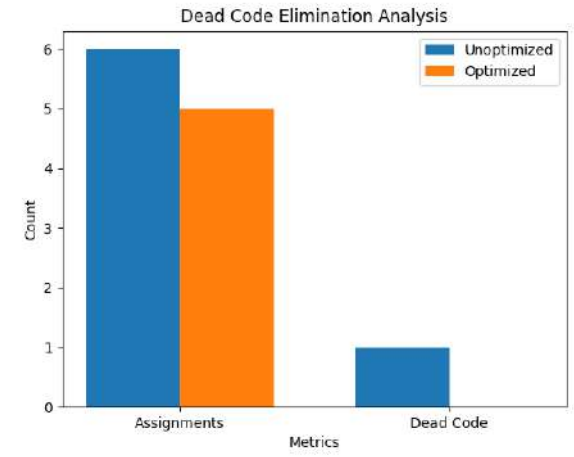
After Constant Folding **:**

```
int main() {
   return 65;
}
```

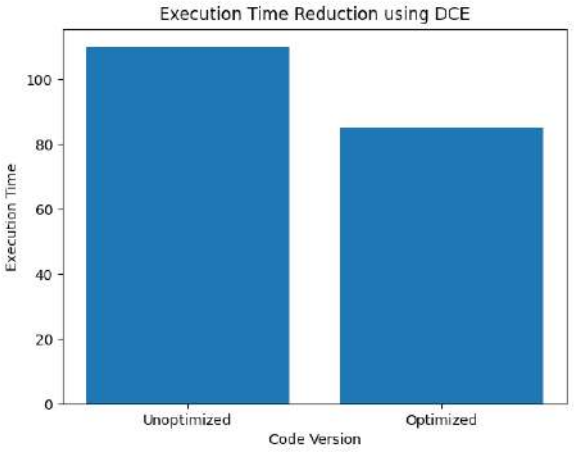
Instruction-Level Comparison **:**

Metric	Unoptimized	Optimized
Variable Assignments	**6**	**5**
Dead Statements	**1**	**0**
Memory Usage	**Higher**	**Lower**
Execution Time	**Higher**	**Lower**

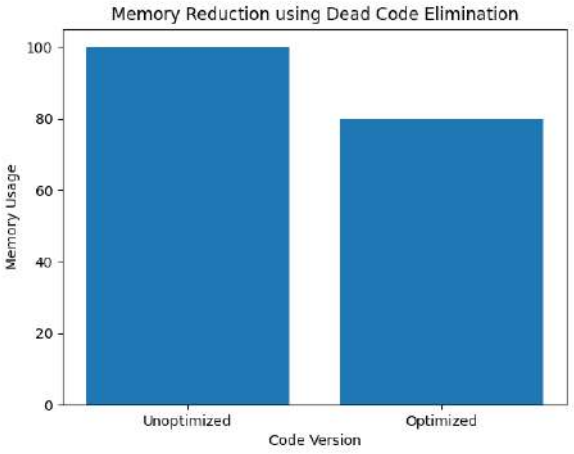
Instruction Count Comparison **:**



Execution Time Comparison (Simulated) **:**



Memory Usage Comparison **:**



Observations **:**

1. DCE removes unused variables and computations
2. Reduces:
   - Instruction count
   - Memory usage
   - Execution time
3. Improves:
   - Cache performance
   - Pipeline efficiency

Limitations :

1. **Side Effects** :
   - Cannot remove statements with side effects (e.g., I/O operations)
2. **Aliasing** :
   - Pointer aliasing may prevent safe elimination
3. **Function Calls** :
   - Unknown effects restrict removal
4. **Volatile Variables** :
   - Must be preserved

Practical Significance :

Dead Code Elimination is one of the most widely used optimizations in compilers such as GCC and LLVM. It plays a critical role in improving both performance and code efficiency, especially when combined with SSA-based optimization frameworks.

[G]. Loop Invarient Computation :

Loop Invariant Computation refers to the identification and movement of computations within a loop that produce the same result on every iteration. These computations, known as **loop-invariant expressions**, can be safely moved outside the loop without changing program semantics. The transformation that performs this movement is called **Loop Invariant Code Motion (LICM)**.

Formally, an expression: x = E

inside a loop is loop-invariant if the value of E does not change across iterations of the loop.

∀ iterations I, E(i)=E

The primary objectives of Loop Invariant Computation are:

- Reduce redundant computations inside loops
- Improve execution speed

- Decrease instruction count per iteration
- Enhance overall program performance

Since loops are executed multiple times, even small optimizations inside loops can yield significant performance improvements.

Example **:**

Unoptimized Code :

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

Observation :

- ❖ Expression : max - 1
- ❖ This value does not change during loop execution
- ❖ It is recomputed in every iteration

Optimized Code (After LICM) **:**

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

Transformations have been done :

- ➢ Loop-invariant expression (max – 1) is computed once
- ➢ Stored in variable n
- ➢ Loop now uses n instead of recomputing

Formal Identification of Loop Invariants**:**

An expression is loop-invariant if:

1. All operands are constants, or
2. Operands are defined outside the loop, and
3. Operands are not modified within the loop

In this example:

- max is not modified → invariant
- 1 is constant → invariant
- Therefore, max - 1 is invariant

Algorithm for LICM **:**

1. Identify loops in Control Flow Graph (CFG)
2. Detect invariant expressions
3. Ensure safety:
   - No side effects
   - Dominates all uses
4. Move computation outside loop
5. Replace occurrences inside loop

Data-Flow Perspective **:**

LICM relies on:

- Reaching Definitions
- Dominance Analysis
- Loop Structure Detection

An expression can be moved if:

- Its operands reach the loop header unchanged
- It dominates all loop exits

Correctness Conditions **:**

LICM is safe only if:

- Expression has no side effects
- Computation yields same result every iteration
- Moving it does not change program semantics

Runtime vs Compile-Time Analysis **:**

Runtime Analysis :

Unoptimized Loop :

```
while(i <= max - 1)
```

Each iteration computes 1 subtraction (max - 1)
So, Total cost for this unoptimized loop will be :

T(unoptimized) = n· (Cost of loop+Cost of subtraction)

Optimized Loop :

```
n = max - 1;
while(i <= n)
```

Each iteration computes no subtraction.
So, Total cost for this optimized loop will be :

T(optimized) = Cost of subtraction + (n · Cost of loop)

So, the performance difference between the unoptimized and optimized code will be :

$$\Delta T = (n-1) \cdot \text{Cost of subtraction}$$

Savings grow linearly with loop iterations.

Compile-Time Analysis **:**

- ❖ LICM introduces One extra assignment (n = max - 1)
- ❖ Compiler overhead:
  - ➢ Loop analysis
  - ➢ Data-flow checks

Time complexity : O(n)

So, Compile-time cost is negligible compared to runtime gain.

Graphical analysis of unoptimized and optimized code with examples :

Unoptimized Code :

```
#include <stdio.h>
int main() {
    int i = 0, max = 100, sum = 0;
    int a[100];
    while (i <= max - 1) {
        sum = sum + a[i];
        i++;
    }
    return sum;
}
```

Explanation :

- Expression max - 1 is evaluated every iteration
- Loop runs n times → n subtractions
- Creates unnecessary computation overhead

Optimized Code (After LICM) :

```
#include <stdio.h>
int main() {
    int i = 0, max = 100, sum = 0;
    int a[100];
    int n = max - 1;
    while (i <= n) {
        sum = sum + a[i];
        i++;
    }
    return sum;
}
```

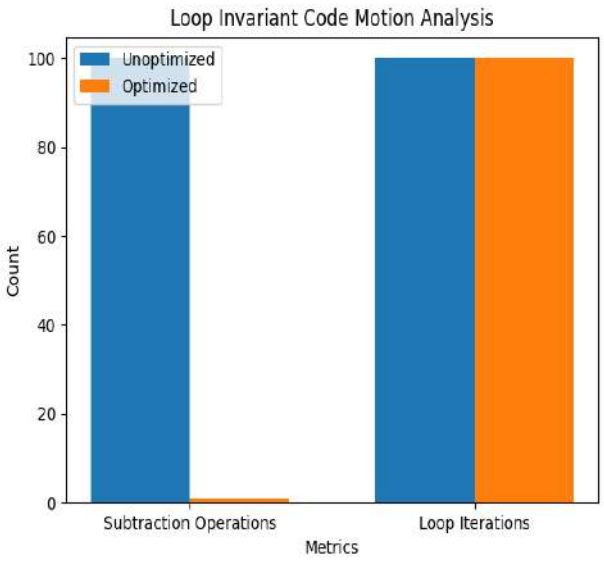
Key Optimizations :

- ❖ max - 1 moved outside loop
- ❖ Computed only once
- ❖ Loop becomes computationally lighter

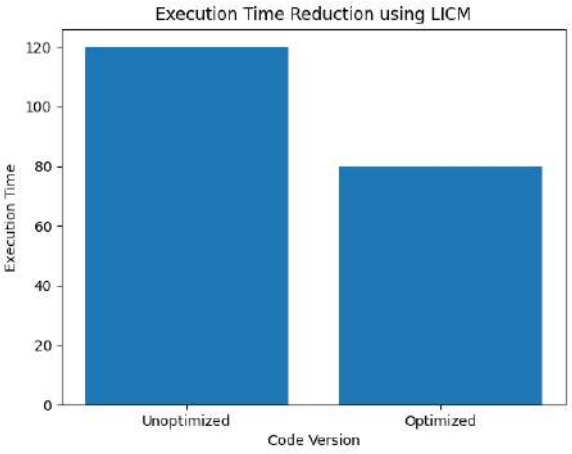
Instruction-Level Comparison **:**

Metric	Unoptimized	Optimized
Subtractions	**n**	**1**
Loop Body Complexity	**Higher**	**Lower**
Execution Time	**Higher**	**Lower**
Compile-Time Overhead	**Low**	**Slightly Higher**

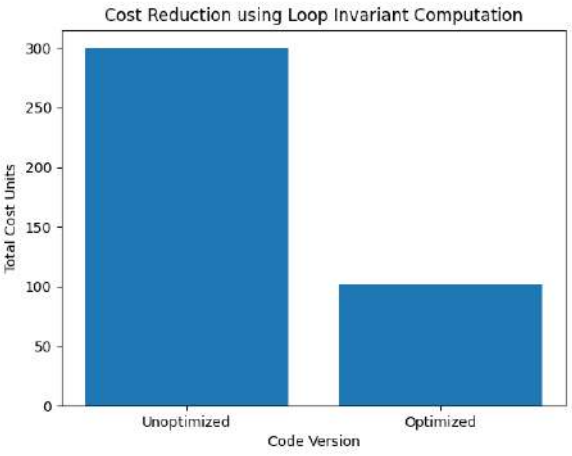
Operation Count Comparison :



Execution Time Comparison :



Cost-Based Analysis :



Observations **:**

- LICM reduces redundant computations inside loops
- Eliminates repeated evaluation of invariant expressions
- Leads to:
  - Lower execution time
  - Reduced ALU operations
  - Improved efficiency

Microarchitectural Impact **:**

1) *1. Reduced ALU Usage*
   - Fewer arithmetic operations per iteration
2) *2. Improved Pipeline Efficiency*
   - Less dependency → smoother execution
3) *3. Better Cache Behavior*
   - Reduced instruction footprint
4) *4. Increased ILP (Instruction-Level Parallelism)*
   - Simpler loop body → better scheduling

Interaction with Other Optimizations **:**

LICM works strongly with:

- Strength Reduction → simplifies loop operations
- Loop Unrolling → reduces loop overhead
- Constant Propagation → creates invariants

Limitations **:**

- **Side Effects** :
  - Cannot move expressions involving I/O or function calls
- **Aliasing** :
  - Pointer modifications may invalidate invariance
- **Register Pressure** :
  - Hoisting may increase register usage

Practical Significance :

LICM is one of the most important loop optimizations used in compilers such as GCC and LLVM.

It is especially effective in:

- Scientific computing
- Data processing loops
- High-performance applications

[H]. Loop Unrolling :

Loop Unrolling is a compiler optimization technique that reduces loop control overhead by replicating the loop body multiple times within a single iteration. Instead of executing one operation per iteration, the loop performs several operations simultaneously, thereby decreasing the number of loop iterations and branch instructions.

Formally, for a loop:

$$\text{for } i=0 \text{ to } n-1$$

executing one statement per iteration, loop unrolling with factor k transforms the loop into k statements per iteration and reducing total iterations approximately to: (n/k).

The primary goals of Loop Unrolling are:

- Reduce loop overhead
- Minimize branch instructions
- Improve instruction-level parallelism (ILP)
- Increase pipeline efficiency
- Enhance cache and memory throughput

Loop overhead includes:

- Increment operations
- Conditional checks
- Branch instructions

These operations do not contribute directly to computation and therefore become optimization targets.

Example :

Unoptimized Code:

```
#include <stdio.h>

int main() {

   int a[100], b[100];

   int i = 0;

   while(i < 100) {

      a[i] = b[i];

      i++;

   }
```

```
    return 0;
}
```

Analysis of Unoptimized code :

For every iteration:

- 1 array assignment
- 1 increment (i++)
- 1 comparison (i < 100)
- 1 branch operation

Total iterations: 100

Optimized Code (Loop Unrolling Factor = 2) :

```
#include <stdio.h>
int main() {
    int a[100], b[100];
    int i = 0;
    while(i < 100) {
        a[i] = b[i];
        i++;
        a[i] = b[i];
        i++;
    }
    return 0;
}
```

Key Transformation *:*

- Two assignments executed per iteration
- Total loop iterations reduced from 100→50

Runtime Analysis :

Runtime of Unoptimized code :

Each iteration performs:

- 1 memory copy
- 1 increment
- 1 comparison
- 1 branch

Total runtime:

T(unoptimized)=n[C(copy)+C(increment)+C(compare) +C(branch)]

Runtime of Optimized code :

Each iteration performs:

- 2 memory copies
- 2 increments
- 1 comparison
- 1 branch

Total runtime:

T(unoptimized)=(n/2)[2C(copy)+2C(increment) +2C(compare)+C(branch)]

Performance Difference **:**

Branch and comparison operations are reduced approximately by 50%
Thus:

$$\Delta T \approx (n/2) [C(cmpare)+C(branch)]$$

Compile-Time Analysis :

Loop Unrolling increases compiler workload because the compiler must:

- Replicate loop body
- Adjust loop structure
- Handle remainder iterations

Compile-time complexity remains approximately: O(n)
However:

- Slightly increased code generation time
- Increased optimization analysis cost

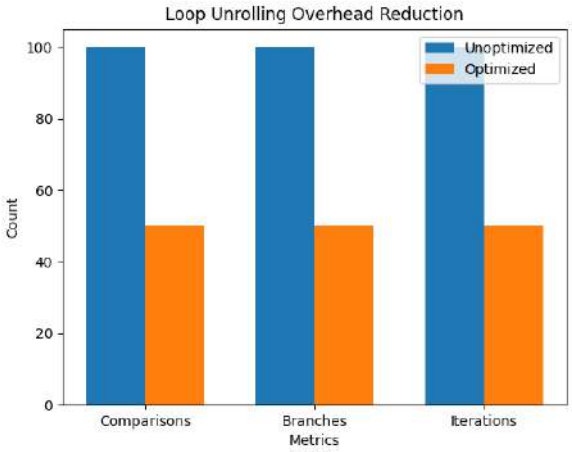
Compared to runtime savings, compile-time overhead is generally negligible.

Instruction-Level Comparison **:**

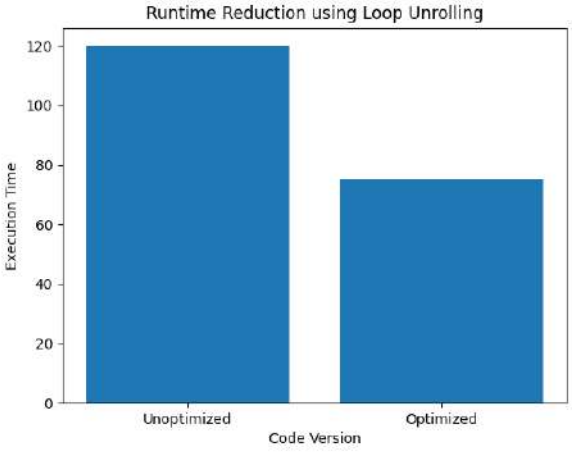
Metric	Unoptimized	Optimized
Loop Iterations	100	50
Comparisons	100	50f
Branch Instructions	100	50
Increment Operations	100	100
Memory Copy Operations	100	100
Runtime Performance	Lower	Higher
Compile-Time Complexity	Lower	Slightly Higher

Graphical Analysis and Comparison between Unoptimized and Optimized Code :

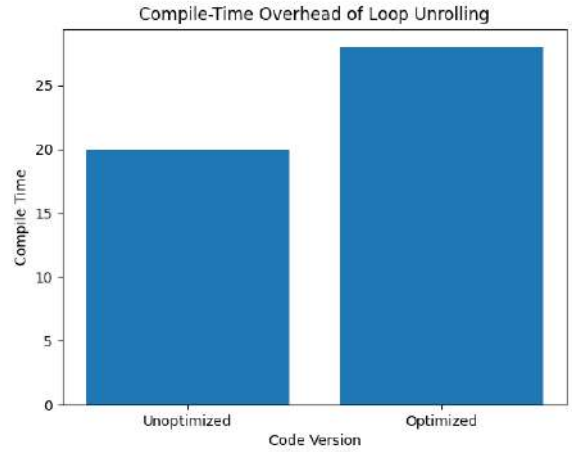
Loop Overhead Comparison :



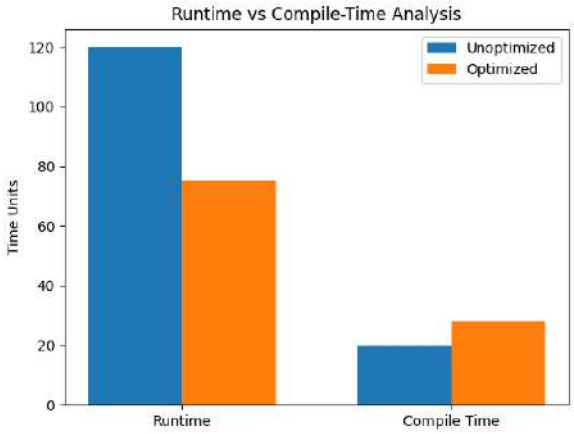
Runtime Comparison :



Compile -Time Comparison :



Runtime vs Compile-Time Combined Analysis:



Observations :

* Loop unrolling significantly reduces:
  - Branch instructions
  - Comparison operations
  - Loop control overhead
* Runtime performance improves because:
  - Fewer control instructions are executed
  - Better instruction scheduling becomes possible
* Compile-time increases slightly due to:
  - Additional compiler analysis
  - Code replication

Microarchitectural Impact :

1. Reduced Branch Overhead :

Fewer loop branches improve:
- Branch prediction accuracy
- Pipeline flow

2**. Improved Instruction-Level Parallelism (ILP)** :

Unrolled loops expose more independent instructions:

Copy 1
Copy 2
Increment 1
Increment 2

Processors can execute these in parallel.

3. Better Pipeline Utilization :

Reduced control instructions decrease:
- Pipeline stalls
- Control hazards

4. Improved Memory Throughput :

Sequential memory accesses improve:

- Cache prefetching
- Memory bandwidth utilization

Interaction with Other Optimizations **:**

Loop Unrolling strongly interacts with:

- Loop Invariant Code Motion (LICM)
- Strength Reduction
- Vectorization
- Instruction Scheduling

Combined effect:

- Significant performance acceleration

Register Pressure **:**

One limitation of Loop Unrolling is increased register usage.
Example:

- Multiple active variables exist simultaneously
- May lead to:
  - Register spilling
  - Increased memory traffic

Thus, excessive unrolling may reduce performance.

Remainder Handling **:**

If loop count is not divisible by unrolling factor:
for(i = 0; i < n-1; i += 2)
Compiler generates:

- Main unrolled loop
- Cleanup loop for remaining iterations

Loop Unrolling in Modern Compilers **:**

Modern compilers such as GCC and LLVM apply Loop Unrolling automatically at optimization levels:

- -O2
- -O3

Heuristics determine:

- Unrolling factor
- Code size trade-offs
- Hardware suitability

Limitations :

1. Code Size Increase :
Unrolling duplicates instructions:

- Larger binary size
- Potential instruction cache pressure

2. Diminishing Returns
Excessive unrolling may:

- Increase register pressure
- Reduce cache efficiency

3. Hardware Dependency
Performance gains depend on:

- Pipeline depth
- Cache size
- Superscalar architecture

Practical Significance **:**

Loop Unrolling is widely used in:

- Scientific computing
- Multimedia processing
- High-performance computing (HPC)
- Numerical simulations

It is particularly beneficial in loops with:

- Simple bodies
- High iteration counts

V. CONCLUSION

This research paper presented a detailed study and performance analysis of fundamental compiler optimization techniques including Constant Folding, Constant Propagation, Variable Propagation, Common Subexpression Elimination using Directed Acyclic Graph (DAG) construction, Strength Reduction, Dead Code Elimination, Loop Invariant Computation, and Loop Unrolling. The analysis demonstrated how these techniques improve program efficiency by reducing redundant computations, minimizing instruction count, decreasing loop overhead, and improving overall execution performance. Through optimized and unoptimized code comparisons, runtime and compile-time analysis, and graphical evaluation, the study highlighted the importance of compiler optimizations in modern computing systems. The research also emphasized the role of modern compilers such as GCC and LLVM in implementing advanced optimization strategies for efficient code generation. Overall, the work concludes that compiler optimization techniques play a vital role in enhancing execution speed, resource utilization, and system performance, making them essential components of modern compiler design and high-performance computing.

VI. ACKNOWLEDGMENT

The author would like to express sincere gratitude to the Department of Computer Science and Engineering for providing the academic support and learning environment necessary for completing this research work. The author also acknowledges the valuable contributions of researchers and educators in the fields of Compiler Design and Computer Architecture whose pioneering studies on compiler optimization techniques greatly inspired this work. Special thanks are extended to the open-source communities behind GCC and LLVM for providing extensive documentation and technical resources that supported the theoretical and analytical aspects of this research. Finally,

the author sincerely appreciates all academic resources and scientific communities that continue to make advanced technical knowledge accessible to students and independent researchers worldwide.

VII. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA, USA: Pearson, 2006.

[2] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann, 1997.

[3] K. D. Cooper and L. Torczon, *Engineering a Compiler*, 2nd ed. Burlington, MA, USA: Morgan Kaufmann, 2011.

[4] S. Chandra, "Compiler Optimization Techniques and Their Impact on Program Performance," *International Journal of Computer Applications*, vol. 182, no. 44, pp. 15–22, 2019.

[5] J. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA, USA: Morgan Kaufmann, 2002.

[6] F. E. Allen, "Control Flow Analysis," in *Proceedings of a Symposium on Compiler Optimization*, New York, NY, USA, 1970, pp. 1–19.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[8] S. Wegman and F. K. Zadeck, "Constant Propagation with Conditional Branches," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 2, pp. 181–210, Apr. 1991.

[9] D. Bernstein, "Compiler Optimization Techniques for High Performance Computing," *IEEE Micro*, vol. 24, no. 5, pp. 22–29, 2004.

[10] M. Wolfe, *High Performance Compilers for Parallel Computing*. Redwood City, CA, USA: Addison-Wesley, 1995.

[11] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1988.

[12] LLVM Compiler Infrastructure Project

[13] GNU Compiler Collection (GCC) Documentation

[14] S. Muchnick and N. Jones, "Data Flow Analysis and Compiler Optimizations," *Journal of Programming Languages*, vol. 5, no. 3, pp. 101–130, 1998.

[15] A. Appel, *Modern Compiler Implementation in C*. Cambridge, U.K.: Cambridge University Press, 1998.

[16] V. Bala and E. Duesterwald, "Redundancy Elimination Revisited," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999, pp. 12–23.

[17] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004, pp. 75–86.

[18] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Burlington, MA, USA: Morgan Kaufmann, 2017.

[19] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Press, 2023.

[20] A. Fog, *Optimizing Software in C++: An Optimization Guide for Windows, Linux and Mac Platforms*, Copenhagen, Denmark, 2022.