Compiler Design

Case Study on Optimization Techniques

Submitted by

CB.SC.U4CSE23535
CB.SC.U4CSE23547
CB.SC.U4CSE23538

Table of Contents

Introduction

A compiler takes source code and translates it into machine code the computer can execute. This happens in three main phases:

1. **Frontend:** Scans the code, tokenises it, checks grammar and type rules, then converts correct code into an Intermediate Representation (IR).
2. **Optimizer:** Takes the IR and improves it without changing the program's observable behaviour — making it run faster and use less memory.
3. **Backend:** Converts the optimized IR into target assembly code that the processor can execute directly.

The Optimizer Phase

The optimizer is critical because high-level code written for readability often introduces redundant operations, unused variables, and re-computable constants. The optimizer detects these patterns using formal analysis (liveness, alias, dataflow) and eliminates or simplifies them before code generation. This saves the CPU from wasting cycles at runtime on work that could have been resolved at compile time.

Different Techniques of Optimization

1. **Loop Unrolling:** Expands small loops into straight-line code, eliminating loop-condition checks and branching overhead.
2. **Loop Vectorization & Transformation:** Identifies array operations that can be processed in parallel using SIMD vector instructions, greatly speeding up bulk data processing.
3. **Dead Code Elimination (DCE):** Removes computations whose results are provably never used, reducing code size and execution time.
4. **Tail Call Optimization (TCO):** Converts tail-recursive calls into flat iterative loops, reusing the current stack frame and preventing stack overflow.
5. **Switch Statement Transformation (Jump Tables):** Converts sequential case comparisons into a direct formula or O(1) jump table when case values are dense and regular.
6. **Algebraic Simplification:** Resolves constant expressions and simplifies arithmetic at compile time so no runtime computation is needed.
7. **Bitwise Optimization:** Compresses static bit-level operations into minimal hardware instructions, often eliminating conditionals entirely.
8. **Interprocedural Constant Propagation (IPCP):** Crosses function boundaries; when a callee always receives static arguments, a specialised constant-propagated clone is generated.
9. **Strength Reduction:** Replaces expensive operations such as hardware division with cheaper equivalents such as reciprocal multiplication and bitwise shifts.

10. **Pointer Aliasing & Load/Store Optimization:** Uses alias analysis to eliminate redundant memory loads; if a value is proven unchanged, the memory fetch is replaced by a constant.
11. **Instruction Scheduling:** Reorders independent instructions so a superscalar CPU can dispatch multiple operations simultaneously through parallel pipelines.
12. **Partial Redundancy Elimination (PRE):** Moves loop-invariant computations out of loops so they execute once rather than on every iteration.
13. **Redundant Assignment Elimination:** Removes duplicate stores to the same address when no state change occurs between the two writes.

Analysis

1. Loop Unrolling

Original Program (C):

```
void loop_unroll(int *a) {
    for (int i = 0; i < 4; i++) {
        a[i] = i * 2;
    }
}
```

Assembly — -O0 (No Optimization):

```
loop_unroll:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-24], rdi
        mov     DWORD PTR [rbp-4], 0
        jmp     .L2
.L3:
        mov     eax, DWORD PTR [rbp-4]
        cdqe
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-24]
        add     rax, rdx
        mov     edx, DWORD PTR [rbp-4]
        add     edx, edx
        mov     DWORD PTR [rax], edx
        add     DWORD PTR [rbp-4], 1
.L2:
        cmp     DWORD PTR [rbp-4], 3
        jle     .L3
        nop
        nop
        pop     rbp
        ret
```

Assembly — -O1:

```
loop_unroll:
        mov     DWORD PTR [rdi], 0
        mov     DWORD PTR [rdi+4], 2
        mov     DWORD PTR [rdi+8], 4
        mov     DWORD PTR [rdi+12], 6
        ret
```

Assembly — -O2:

```
loop_unroll:
        movdqa  xmm0, XMMWORD PTR .LC0[rip]
        movups  XMMWORD PTR [rdi], xmm0
        ret
.LC0:
        .long   0
        .long   2
        .long   4
        .long   6
```

Assembly — -O3:

```
loop_unroll:
        movdqa  xmm0, XMMWORD PTR .LC0[rip]
        movups  XMMWORD PTR [rdi], xmm0
        ret
.LC0:
        .long   0
        .long   2
        .long   4
        .long   6
```

Assembly — -Os (Size Optimized):

```
loop_unroll:
        mov     eax, 1
        sal     rax, 33
        mov     QWORD PTR [rdi], rax
        movabs  rax, 25769803780
        mov     QWORD PTR [rdi+8], rax
        ret
```

Inference:

Optimization Level	Compiler Behaviour & Observations
`-O0`	Full loop with jmp/.L2/.L3 jump checks the end condition on every iteration. Heavy stack setup with pointer arithmetic required just to store four constant values — entirely unnecessary overhead.
`-O1`	The compiler detects the loop runs exactly four times and unrolls it completely. All four constant results (0, 2, 4, 6) are written directly to fixed memory offsets. No loop counter, no branch, no stack frame.
`-O2` *(Identical to -O3)*	Advances to SIMD vectorisation. Packs all four 32-bit values into a 128-bit xmm0 register from a read-only constant pool (.LC0) and writes the entire 16 bytes in one movups instruction. Maximum throughput — one instruction for the entire function body.

Optimization Level	Compiler Behaviour & Observations
`-O3` *(Identical to -O2)*	Same SIMD approach as -O2. The loop is already fully vectorised using a single 128-bit store; no further structural improvement is possible.
`-Os`	Encodes the same four values using bitwise shifting tricks: sal rax, 33 packs the first two values and movabs encodes the second two as a 64-bit immediate. Produces the minimum number of instruction bytes while preserving correctness.

2. Loop Vectorization & Transformation

Original Program (C):

```
void transform_data(int *restrict a, int *restrict b, int n) {
    for (int i = 0; i < n; i++) {
        a[i] = b[i] * 4;
    }
}
```

Assembly — -O0 (No Optimization):

```
transform_data:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-24], rdi
        mov     QWORD PTR [rbp-32], rsi
        mov     DWORD PTR [rbp-36], edx
        mov     DWORD PTR [rbp-4], 0
        jmp     .L2
.L3:
        mov     eax, DWORD PTR [rbp-4]
        cdqe
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-32]
        add     rax, rdx
        mov     edx, DWORD PTR [rax]
        mov     eax, DWORD PTR [rbp-4]
        cdqe
        lea     rcx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-24]
        add     rax, rcx
        sal     edx, 2
        mov     DWORD PTR [rax], edx
        add     DWORD PTR [rbp-4], 1
.L2:
        mov     eax, DWORD PTR [rbp-4]
        cmp     eax, DWORD PTR [rbp-36]
        jl      .L3
        nop
        nop
        pop     rbp
        ret
```

Assembly — -O1:

```
transform_data:
        test    edx, edx
        jle     .L1
        movsx   rdx, edx
        lea     rcx, [0+rdx*4]
        mov     eax, 0
.L3:
        mov     edx, DWORD PTR [rsi+rax]
        sal     edx, 2
        mov     DWORD PTR [rdi+rax], edx
        add     rax, 4
        cmp     rax, rcx
        jne     .L3
.L1:
        ret
```

Assembly — -O2:

```
transform_data:
        test    edx, edx
        jle     .L1
        movsx   rdx, edx
        xor     eax, eax
        sal     rdx, 2
.L3:
        mov     ecx, DWORD PTR [rsi+rax]
        sal     ecx, 2
        mov     DWORD PTR [rdi+rax], ecx
        add     rax, 4
        cmp     rax, rdx
        jne     .L3
.L1:
        ret
```

Assembly — -O3 (SIMD Vectorized):

```
transform_data:
        test    edx, edx
        jle     .L1
        lea     eax, [rdx-1]
        cmp     eax, 2
        jbe     .L8
        mov     ecx, edx
        xor     eax, eax
        shr     ecx, 2
        sal     rcx, 4
.L4:
        movdqu  xmm0, XMMWORD PTR [rsi+rax]
        pslld   xmm0, 2
        movups  XMMWORD PTR [rdi+rax], xmm0
        add     rax, 16
        cmp     rax, rcx
        jne     .L4
        ...
.L1:
        ret
```

Assembly — -Os (Size Optimized):

```
transform_data:
        mov     ecx, edx
        xor     eax, eax
.L2:
        cmp     ecx, eax
        jle     .L5
        mov     edx, DWORD PTR [rsi+rax*4]
        sal     edx, 2
        mov     DWORD PTR [rdi+rax*4], edx
        inc     rax
        jmp     .L2
.L5:
        ret
```

Inference:

Optimization Level	Compiler Behaviour & Observations
`-O0`	Full memory load pathways (mov, lea, add) are computed on every single iteration. Stack pointer arithmetic is recalculated each pass — extremely slow for large arrays.
`-O1`	Stack frame eliminated. The pointer multiplication is replaced with a direct shift-left by 2 (sal edx, 2), equivalent to multiply-by-4. Cleaner scalar loop with byte-offset iteration (add rax, 4).
`-O2`	Pre-computes the total byte length of the array (sal rdx, 2) before entering the loop so the loop counter does not require additional arithmetic per iteration. Minor instruction reordering for pipeline efficiency.
`-O3` *(SIMD vectorised)*	Transforms the loop using SSE vector instructions. Loads four 32-bit integers simultaneously (movdqu), shifts all four in a single instruction (pslld xmm0, 2), and stores all four at once (movups). Processes 4 elements per clock cycle instead of one — up to 4x throughput improvement.
`-Os`	Removes all vectorisation to minimise binary size. Reverts to a compact scalar loop using integer indexing (inc rax / jmp .L2) — fewest instruction bytes at the cost of throughput.

3. Dead Code Elimination (DCE)

Original Program (C):

```
int global_var = 0;

int dead_code_example(int x) {
    int a = x + 10;         // dead: result never read
    int b = a * 2;          // dead: overwritten before use
    global_var = 5;         // side effect: must be preserved
    b = b + 5;              // dead: overwritten on the very next line
    b = x + 5;              // only this value of b matters
    return b;
}
```

Assembly — -O0 (No Optimization):

```
dead_code_example:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     eax, DWORD PTR [rbp-20]
        add     eax, 10
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        add     eax, eax
        mov     DWORD PTR [rbp-8], eax
        mov     DWORD PTR global_var[rip], 5
        add     DWORD PTR [rbp-8], 5
        mov     eax, DWORD PTR [rbp-20]
        add     eax, 5
        mov     DWORD PTR [rbp-8], eax
        mov     eax, DWORD PTR [rbp-8]
        pop     rbp
        ret
```

Assembly — -O1:

```
dead_code_example:
        mov     DWORD PTR global_var[rip], 5
        lea     eax, [rdi+5]
        ret
global_var:
        .zero   4
```

Assembly — -O2:

```
dead_code_example:
        mov     DWORD PTR global_var[rip], 5
        lea     eax, [rdi+5]
        ret
global_var:
        .zero   4
```

Assembly — -O3:

```
dead_code_example:
        mov     DWORD PTR global_var[rip], 5
        lea     eax, [rdi+5]
        ret
global_var:
        .zero   4
```

Assembly — -Os (Size Optimized):

```
dead_code_example:
        mov     DWORD PTR global_var[rip], 5
        lea     eax, [rdi+5]
        ret
global_var:
        .zero   4
```

Inference:

Optimization Level	Compiler Behaviour & Observations
`-O0`	Every statement executes in order: a = x+10 is computed and stored, b = a*2 is computed and stored, global_var is updated, b is added to and then overwritten again. All intermediate dead results consume stack space and CPU cycles with no contribution to the final answer.
`-O1` *(DCE fully applied)*	Liveness analysis determines that a and all intermediate values of b are written but never subsequently read — they are provably dead. All are eliminated. global_var must be preserved as a mandatory observable side effect. The function collapses to exactly two instructions: update global_var (mov) and return x+5 (lea). A 15-instruction function becomes 2.
`-O2` *(Identical to -O1)*	Dead code was fully removed at -O1. No further dead assignments remain. Assembly output is identical.
`-O3` *(Identical to -O1)*	No additional dead code exists. Assembly output is identical to -O1 and -O2.
`-Os` *(Identical to -O1)*	Same elimination. The two-instruction result is already the minimum possible code size.

4. Tail Call Optimization (TCO)

Original Program (C):

```
int factorial_tail(int n, int accumulator) {
    if (n <= 0) return accumulator;
    return factorial_tail(n - 1, n * accumulator);  // tail call
}
```

Assembly — -O0 (No Optimization):

```
factorial_tail:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        cmp     DWORD PTR [rbp-4], 0
        jg      .L2
        mov     eax, DWORD PTR [rbp-8]
        jmp     .L3
.L2:
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-8]
        mov     edx, DWORD PTR [rbp-4]
        sub     edx, 1
        mov     esi, eax
        mov     edi, edx
        call    factorial_tail
.L3:
        leave
        ret
```

Assembly — -O1:

```
factorial_tail:
        mov     eax, esi
        test    edi, edi
        jle     .L5
        sub     rsp, 8
        imul    esi, edi
        sub     edi, 1
        call    factorial_tail
        add     rsp, 8
        ret
.L5:
        ret
```

Assembly — -O2 (TCO Applied):

```
factorial_tail:
        mov     eax, esi
        test    edi, edi
        jle     .L5
        test    dil, 1
        je      .L2
        imul    eax, edi
        sub     edi, 1
        test    edi, edi
        je      .L17
.L2:
        imul    eax, edi
        lea     edx, [rdi-1]
        imul    eax, edx
        sub     edi, 2
        jne     .L2
.L5:
        ret
.L17:
        ret
```

Assembly — -O3:

```
factorial_tail:
        mov     eax, esi
        test    edi, edi
        jle     .L5
        test    dil, 1
        je      .L2
        imul    eax, edi
        sub     edi, 1
        test    edi, edi
        je      .L17
.L2:
        imul    eax, edi
        lea     edx, [rdi-1]
        imul    eax, edx
        sub     edi, 2
        jne     .L2
.L5:
        ret
.L17:
        ret
```

Assembly — -Os (Flattest Iterative Form):

```
factorial_tail:
        mov     eax, esi
.L3:
        test    edi, edi
        jle     .L4
        imul    eax, edi
        dec     edi
        jmp     .L3
.L4:
        ret
```

Inference:

Optimization Level	Compiler Behaviour & Observations
`-O0`	Genuinely recursive. A new stack frame is pushed and call factorial_tail executes on every step. For large n this causes a stack overflow. Memory usage grows as O(n) with the recursion depth.
`-O1`	Optimises the function body (removes stack-based argument spilling) but still emits a call instruction. The recursion is not yet eliminated; stack depth continues to grow linearly with n.
`-O2` *(TCO applied)*	Proves the tail call can be safely replaced by a jump-back loop. The call instruction is removed entirely. Stack depth becomes constant O(1). A two-step unrolling pattern (sub edi, 2) processes two iterations per loop pass.
`-O3` *(Identical to -O2)*	Same tail-call elimination with the same two-step unrolling. No further structural improvement is identified.
`-Os` *(TCO, minimal form)*	Produces the cleanest iterative form: a single tight loop (imul / dec / jmp .L3). Fewest instruction bytes. Stack depth is O(1).

5. Switch Statement Transformation (Jump Tables)

Original Program (C):

```
int switch_example(int color_code) {
    switch (color_code) {
        case 0: return 100;
        case 1: return 200;
```

```
            case 2: return 300;
            case 3: return 400;
            default: return 0;
        }
    }
```

Assembly — -O0 (No Optimization):

```
switch_example:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        cmp     DWORD PTR [rbp-4], 3
        je      .L2
        cmp     DWORD PTR [rbp-4], 3
        jg      .L3
        cmp     DWORD PTR [rbp-4], 2
        je      .L4
        cmp     DWORD PTR [rbp-4], 2
        jg      .L3
        cmp     DWORD PTR [rbp-4], 0
        je      .L5
        cmp     DWORD PTR [rbp-4], 1
        je      .L6
        jmp     .L3
.L5:    mov     eax, 100
        jmp     .L7
.L6:    mov     eax, 200
        jmp     .L7
.L4:    mov     eax, 300
        jmp     .L7
.L2:    mov     eax, 400
        jmp     .L7
.L3:    mov     eax, 0
.L7:
        pop     rbp
        ret
```

Assembly — -O1:

```
switch_example:
        cmp     edi, 2
        je      .L4
        jg      .L3
        mov     eax, 100
        test    edi, edi
        je      .L1
        cmp     edi, 1
        mov     eax, 200
        mov     edx, 0
        cmovne  eax, edx
        ret
.L3:
        cmp     edi, 3
        mov     eax, 400
        mov     edx, 0
        cmovne  eax, edx
        ret
.L4:
        mov     eax, 300
.L1:
```

```
                ret
```

Assembly — -O2 (Algebraic Formula):

```
switch_example:
        lea     edx, [rdi+1]
        xor     eax, eax
        imul    edx, edx, 100
        cmp     edi, 3
        cmovbe  eax, edx
        ret
```

Assembly — -O3:

```
switch_example:
        lea     edx, [rdi+1]
        xor     eax, eax
        imul    edx, edx, 100
        cmp     edi, 3
        cmovbe  eax, edx
        ret
```

Assembly — -Os (Size Optimized):

```
switch_example:
        lea     eax, [rdi+1]
        xor     edx, edx
        imul    eax, eax, 100
        cmp     edi, 4
        cmovnb  eax, edx
        ret
```

Inference:

Optimization Level	Compiler Behaviour & Observations
`-O0`	A sequential waterfall of cmp + je/jg pairs tests every case individually until a match is found. Up to six comparisons and jumps may execute for a single lookup. No use of the mathematical relationship between the case values and their return values.
`-O1`	Applies a binary-search ordering of cases and uses conditional-move instructions (cmovne) to reduce comparison depth. Still branch-based, but structured more efficiently than the linear waterfall.
`-O2` *(Algebraic formula)*	Recognises that cases 0, 1, 2, 3 map linearly to (x+1)*100. All branches are replaced with a pure algebraic expression: lea computes x+1, imul multiplies by 100, and cmovbe returns 0 for out-of-range inputs. Completely branchless — executes in constant time regardless of input.
`-O3` *(Identical to -O2)*	Same algebraic formula. The function is already at its optimal form.

Optimization Level	Compiler Behaviour & Observations
-Os *(Algebraic formula, minimal bytes)*	Same algebraic approach with a slightly different boundary encoding (cmp edi, 4 / cmovnb) chosen to minimise instruction byte count.

6. Algebraic Expression & Constant Folding

Original Program (C):

```
#include <stdio.h>
#include <linux/types.h>
#define BIGEDIAN

int main() {
    __u32 tmp[2] = {
#ifdef BIGEDIAN
        0x9D8BC888, 0x00000764
#else
        0x00000764, 0x8B9D8C88
#endif
    };
    __u64 *val = (__u64 *)&tmp[0];
    printf("Value is %llu.\n.", *val);
    return 0;
}
```

Assembly — -O0 (No Optimization):

```
.LC0:
        .string "Value is %llu.\n."
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-16], -1651783544
        mov     DWORD PTR [rbp-12], 1892
        lea     rax, [rbp-16]
        mov     QWORD PTR [rbp-8], rax
        mov     rax, QWORD PTR [rbp-8]
        mov     rax, QWORD PTR [rax]
        mov     rsi, rax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        leave
        ret
```

Assembly — -O1:

```
.LC0:
        .string "Value is %llu.\n."
main:
        sub     rsp, 8
        movabs  rsi, 8128721307784
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        add     rsp, 8
        ret
```

Assembly — -O2:

```
.LC0:
        .string "Value is %llu.\n."
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        movabs  rsi, 8128721307784
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Assembly — -O3:

```
.LC0:
        .string "Value is %llu.\n."
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        movabs  rsi, 8128721307784
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Assembly — -Os (Size Optimized):

```
.LC0:
        .string "Value is %llu.\n."
main:
        push    rax
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        movabs  rsi, 8128721307784
        call    printf
        xor     eax, eax
        pop     rdx
        ret
```

Inference:

Optimization Level	Compiler Behaviour & Observations
`-O0`	Allocates a 16-byte stack frame and independently stores the two 32-bit halves (0x9D8BC888 = -1651783544 and 0x00000764 = 1892) into memory. Computes the array base address via lea, stores it as a pointer, then dereferences that pointer back again just to pass the value to printf. Fully redundant memory round-trips for a value that was always static.
`-O1` *(Constant fold)*	Recognises at compile time that the two adjacent 32-bit integers constitute a single 64-bit value: 8128721307784. The stack array, the pointer, and all memory operations are eliminated entirely. The constant is loaded directly into the rsi argument register using movabs — zero runtime memory allocation.
`-O2`	Same constant folding as -O1. Additionally replaces mov eax, 0 with the faster two-byte xor eax, eax for clearing the return register, and reorders argument setup for better pipeline scheduling.
`-O3` *(Identical to -O2)*	No further improvements possible. Constant folding and instruction scheduling were already applied at -O2.
`-Os`	Same constant folding. Replaces the sub rsp, 8 / add rsp, 8 stack-alignment pair with a single push rax / pop rdx, saving two instruction bytes.

7. Bitwise Optimization & Constant Folding

Original Program (C):

```
#include <stdio.h>
#include <linux/types.h>
#define FLAG_ONE   1 << 0
#define FLAG_TWO   1 << 1
#define FLAG_THREE 1 << 2
#define COMBINED   0x07

int main() {
    __u32 flags = 0;
    flags |= FLAG_ONE;
    flags |= FLAG_TWO;
    flags |= FLAG_THREE;
    if ((flags & COMBINED) == COMBINED)
        printf("Flags set correctly!\n");
    return 0;
}
```

Assembly — -O0 (No Optimization):

```
.LC0:
        .string "Flags set correctly!"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 0
        or      DWORD PTR [rbp-4], 1
        or      DWORD PTR [rbp-4], 2
        or      DWORD PTR [rbp-4], 4
        mov     eax, DWORD PTR [rbp-4]
        and     eax, 7
        cmp     eax, 7
        jne     .L2
        mov     edi, OFFSET FLAT:.LC0
        call    puts
.L2:
        mov     eax, 0
        leave
        ret
```

Assembly — -O1:

```
.LC0:
        .string "Flags set correctly!"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0
        call    puts
        mov     eax, 0
        add     rsp, 8
        ret
```

Assembly — -O2:

```
.LC0:
        .string "Flags set correctly!"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0
        call    puts
        xor     eax, eax
        add     rsp, 8
        ret
```

Assembly — -O3:

```
.LC0:
        .string "Flags set correctly!"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0
        call    puts
        xor     eax, eax
        add     rsp, 8
        ret
```

Assembly — -Os (Size Optimized):

```
.LC0:
        .string "Flags set correctly!"
main:
        push    rax
        mov     edi, OFFSET FLAT:.LC0
        call    puts
        xor     eax, eax
        pop     rdx
        ret
```

Inference:

Optimization Level	Compiler Behaviour & Observations
-O0	Initialises flags to 0 in memory, then applies three separate OR operations (1, 2, 4) to that memory location. Copies the result to eax, applies an AND mask against 7, runs a CMP against 7, and conditionally jumps. Fifteen instructions to determine an outcome that is mathematically certain at compile time.
-O1 *(Full constant fold)*	Evaluates 1|2|4 = 7 and 7&7 == 7 statically. The branch is provably always taken. The variable, all bitwise operations, the mask, the comparison, and the conditional jump are eliminated entirely. Execution falls straight to call puts.
-O2	Same constant fold as -O1. Additionally replaces mov eax, 0 with xor eax, eax for a faster, smaller return-value clear.
-O3 *(Identical to -O2)*	No further improvements. The conditional logic was fully eliminated at -O1.
-Os	Same constant fold. Stack alignment managed with push rax / pop rdx instead of sub/add rsp to reduce instruction byte count.

8. Interprocedural Constant Propagation (IPCP)

Original Program (C):

```
__attribute__((noinline))
int compute_complex(int x, int check) {
    if (check > 10) return x * 100;
    return x + 5;
}

int wrapper() {
    return compute_complex(10, 5);  // always called with constants 10 and 5
}
```

Assembly — -O0 (No Optimization):

```
compute_complex:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        cmp     DWORD PTR [rbp-8], 10
        jle     .L2
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax, 100
        jmp     .L3
```

```
.L2:
        mov     eax, DWORD PTR [rbp-4]
        add     eax, 5
.L3:
        pop     rbp
        ret
wrapper:
        push    rbp
        mov     rbp, rsp
        mov     esi, 5
        mov     edi, 10
        call    compute_complex
        pop     rbp
        ret
```

Assembly — -O1:

```
compute_complex:
        imul    eax, edi, 100
        add     edi, 5
        cmp     esi, 10
        cmovle  eax, edi
        ret
wrapper:
        mov     esi, 5
        mov     edi, 10
        call    compute_complex
        ret
```

Assembly — -O2:

```
compute_complex:
        imul    eax, edi, 100
        add     edi, 5
        cmp     esi, 10
        cmovle  eax, edi
        ret
wrapper:
        mov     esi, 5
        mov     edi, 10
        jmp     compute_complex
```

Assembly — -O3 (IPCP Applied):

```
compute_complex.constprop.0:
        mov     eax, 15
        ret
compute_complex:
        imul    eax, edi, 100
        add     edi, 5
        cmp     esi, 10
        cmovle  eax, edi
        ret
wrapper:
        mov     eax, 15
        ret
```

Assembly — -Os (Size Optimized):

```
compute_complex:
        imul    eax, edi, 100
        add     edi, 5
        cmp     esi, 10
        cmovle  eax, edi
        ret
wrapper:
        mov     esi, 5
        mov     edi, 10
        jmp     compute_complex
```

Inference:

Optimization Level	Compiler Behaviour & Observations
-O0	wrapper allocates a full stack frame, loads constants 10 and 5 into registers, and calls compute_complex. Inside compute_complex, a cmp and conditional jump evaluate the branch dynamically on every invocation — no compile-time knowledge is applied.
-O1	compute_complex is rewritten using a branchless conditional-move (cmovle). wrapper still uses a call instruction — the function boundary is not yet crossed for propagation purposes.
-O2	wrapper replaces the call with a direct jmp, eliminating the return overhead (tail-call elimination). compute_complex body is unchanged.
-O3 *(IPCP applied)*	The compiler crosses the function boundary. It proves wrapper always passes the constants 10 and 5, determines if (5 > 10) is permanently false, and generates a specialised clone compute_complex.constprop.0 that contains only mov eax, 15; ret. wrapper is rewritten to return 15 directly. The entire conditional logic of compute_complex is eliminated for this call site.
-Os	Same as -O2: tail-call jmp applied. The IPCP clone is not generated because duplicating the function body would increase code size.

9. Strength Reduction & Constant Folding

Original Program (C):

```
#include <stdio.h>
#include <stdint.h>

int main() {
    uint8_t value = 0;
    value += 30;
    value /= 10;
    printf("Value is %d.\n", value);
    return 0;
}
```

Assembly — -O0 (Strength Reduction visible):

```
.LC0:
        .string "Value is %d.\n"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     BYTE PTR [rbp-1], 0
        add     BYTE PTR [rbp-1], 30
        movzx   eax, BYTE PTR [rbp-1]
        mov     edx, -51
        mul     dl
        shr     ax, 8
        shr     al, 3
        mov     BYTE PTR [rbp-1], al
        movzx   eax, BYTE PTR [rbp-1]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        leave
        ret
```

Assembly — -O1 (Constant Folding):

```
.LC0:
        .string "Value is %d.\n"
main:
        sub     rsp, 8
        mov     esi, 3
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        add     rsp, 8
        ret
```

Assembly — -O2:

```
.LC0:
        .string "Value is %d.\n"
main:
        sub     rsp, 8
        mov     esi, 3
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Assembly — -O3:

```
.LC0:
        .string "Value is %d.\n"
main:
        sub     rsp, 8
        mov     esi, 3
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Assembly — -Os (Size Optimized):

```
.LC0:
        .string "Value is %d.\n"
main:
        push    rax
        mov     esi, 3
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        xor     eax, eax
        pop     rdx
        ret
```

Inference:

Optimization Level	Compiler Behaviour & Observations
-O0 *(Strength Reduction)*	The CPU has no fast integer-division path for 8-bit values. Even without optimisation flags, the compiler applies Strength Reduction automatically: division by 10 is replaced with a reciprocal multiplication (mul dl using the magic constant -51) combined with bitwise shifts (shr ax, 8 and shr al, 3). This is mathematically equivalent to dividing by 10 but uses only cheap multiply and shift hardware.
-O1 *(Constant Folding)*	Evaluates the entire computation statically: 0 + 30 = 30, 30 / 10 = 3. The stack variable, the Strength Reduction multiply-shift sequence, and all arithmetic are eliminated. Only mov esi, 3 remains — the answer is a hardcoded compile-time constant.
-O2	Same constant folding as -O1. Replaces mov eax, 0 with xor eax, eax for faster return-register clearing, and reorders argument loads for better pipeline utilisation.
-O3 *(Identical to -O2)*	No further improvements. Compile-time evaluation was complete at -O1.
-Os	Same constant fold. Replaces sub rsp, 8 / add rsp, 8 with push rax / pop rdx to minimise instruction byte count.

10. Pointer Aliasing & Load/Store Optimization

Original Program (C):

```
#include <stdio.h>

int main() {
    int i = 1;
    int *x = &i;
    fprintf(stdout, "i = %d. x = %d.\n", i, *x);
    return 0;
}
```

Assembly — -O0 (No Optimization):

```
.LC0:
        .string "i = %d. x = %d.\n"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-12], 1
        lea     rax, [rbp-12]
        mov     QWORD PTR [rbp-8], rax
        mov     rax, QWORD PTR [rbp-8]
        mov     ecx, DWORD PTR [rax]
        mov     edx, DWORD PTR [rbp-12]
        mov     rax, QWORD PTR stdout[rip]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, rax
        mov     eax, 0
        call    fprintf
        mov     eax, 0
        leave
        ret
```

Assembly — -O1:

```
.LC0:
        .string "i = %d. x = %d.\n"
main:
        sub     rsp, 8
        mov     ecx, 1
        mov     edx, 1
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stdout[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 0
        add     rsp, 8
        ret
```

Assembly — -O2:

```
.LC0:
        .string "i = %d. x = %d.\n"
main:
        sub     rsp, 8
        mov     ecx, 1
        mov     edx, 1
        xor     eax, eax
        mov     rdi, QWORD PTR stdout[rip]
        mov     esi, OFFSET FLAT:.LC0
        call    fprintf
        xor     eax, eax
        add     rsp, 8
        ret
```

Assembly — -O3:

```
.LC0:
        .string "i = %d. x = %d.\n"
main:
        sub     rsp, 8
        mov     ecx, 1
        mov     edx, 1
        xor     eax, eax
        mov     rdi, QWORD PTR stdout[rip]
        mov     esi, OFFSET FLAT:.LC0
        call    fprintf
        xor     eax, eax
        add     rsp, 8
        ret
```

Assembly — -Os (Size Optimized):

```
.LC0:
        .string "i = %d. x = %d.\n"
main:
        push    rax
        mov     rdi, QWORD PTR stdout[rip]
        mov     edx, 1
        xor     eax, eax
        mov     ecx, 1
        mov     esi, OFFSET FLAT:.LC0
        call    fprintf
        xor     eax, eax
        pop     rdx
        ret
```

Inference:

Optimization Level	Compiler Behaviour & Observations
`-O0`	Stores i = 1 in memory at [rbp-12]. Separately computes its address via lea and stores that address as a pointer at [rbp-8]. Then dereferences the pointer (mov ecx, DWORD PTR [rax]) to read back the value 1. Two redundant memory accesses for a value that is a compile-time constant and never modified.
`-O1` *(Alias analysis)*	Alias analysis proves that *x and i always refer to the same memory location and that location is never written after initialisation. The pointer, the stack allocation, the lea instruction, and all memory loads are eliminated. Both i and *x are replaced by the constant 1 in-register: mov ecx, 1 and mov edx, 1.
`-O2`	Same alias-based elimination as -O1. Additionally uses xor eax, eax for a faster return-register clear and reorders argument-register assignments for better pipeline scheduling.
`-O3` *(Identical to -O2)*	No further improvements. Load elimination was complete at -O1.

Optimization Level	Compiler Behaviour & Observations
`-Os`	Same alias analysis applied. Uses push rax / pop rdx stack management and a size-optimal register assignment order to minimise instruction byte count.

11. Instruction Scheduling

Original Program (C):

```
int scheduling_example(int a, int b, int c, int d) {
    int x = a * b;   // independent of y
    int y = c * d;   // independent of x
    return x + y;
}
```

Assembly — -O0 (No Optimization):

```
scheduling_example:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-24], esi
        mov     DWORD PTR [rbp-28], edx
        mov     DWORD PTR [rbp-32], ecx
        mov     eax, DWORD PTR [rbp-20]
        imul    eax, DWORD PTR [rbp-24]
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-28]
        imul    eax, DWORD PTR [rbp-32]
        mov     DWORD PTR [rbp-8], eax
        mov     edx, DWORD PTR [rbp-4]
        mov     eax, DWORD PTR [rbp-8]
        add     eax, edx
        pop     rbp
        ret
```

Assembly — -O1:

```
scheduling_example:
        imul    edi, esi
        imul    edx, ecx
        lea     eax, [rdi+rdx]
        ret
```

Assembly — -O2:

```
scheduling_example:
        imul    edi, esi
        imul    edx, ecx
        lea     eax, [rdi+rdx]
        ret
```

Assembly — -O3:

```
scheduling_example:
        imul    edi, esi
        imul    edx, ecx
        lea     eax, [rdi+rdx]
        ret
```

Assembly — -Os (Size Optimized):

```
scheduling_example:
        imul    edi, esi
        imul    edx, ecx
        lea     eax, [rdi+rdx]
        ret
```

Inference:

Optimization Level	Compiler Behaviour & Observations
`-O0`	All four parameters are spilled to the stack. a*b is computed and stored to [rbp-4], then c*d is computed and stored to [rbp-8], then both are reloaded and added. Each step stalls until the previous completes — sequential execution with no pipeline parallelism possible.
`-O1` *(Scheduling applied)*	Dependency graph analysis determines that imul edi, esi (computing a*b) and imul edx, ecx (computing c*d) operate on entirely different register sets with no data dependency between them. They are scheduled back-to-back so a superscalar CPU can issue both to separate execution units in the same clock cycle. The sum is folded into a single lea instruction. The stack frame is eliminated entirely — four registers in, one register out.
`-O2` *(Identical to -O1)*	The three-instruction form is already optimal. No further scheduling improvements are possible for this function.
`-O3` *(Identical to -O1)*	Same result. The function body is as compact and parallel as the architecture allows.
`-Os` *(Identical to -O1)*	The three-instruction form is also the minimum binary size. No modifications needed.

12. Partial Redundancy Elimination (PRE)

Original Program (C):

```
void pre_example(int *arr, int x, int y, int n) {
    for (int i = 0; i < n; i++) {
        if (n > 10) {
            arr[i] = (x * y) + i;   // x*y is loop-invariant
        } else {
            arr[i] = i;
        }
    }
}
```

Assembly — -O0 (No Optimization):

```
pre_example:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-24], rdi
        mov     DWORD PTR [rbp-28], esi
        mov     DWORD PTR [rbp-32], edx
        mov     DWORD PTR [rbp-36], ecx
        mov     DWORD PTR [rbp-4], 0
        jmp     .L2
.L5:
        cmp     DWORD PTR [rbp-36], 10
        jle     .L3
        mov     eax, DWORD PTR [rbp-28]
        imul    eax, DWORD PTR [rbp-32]    ; x*y recomputed every iteration
        mov     ecx, eax
        ...
.L2:
        cmp     eax, DWORD PTR [rbp-36]
        jl      .L5
        pop     rbp
        ret
```

Assembly — -O1 (Invariant Hoisted):

```
pre_example:
        test    ecx, ecx
        jle     .L1
        imul    edx, esi             ; x*y computed ONCE before the loop
        movsx   rsi, ecx
        mov     eax, 0
.L5:
        lea     r8d, [rdx+rax]
        cmp     ecx, 10
        cmovle  r8d, eax
        mov     DWORD PTR [rdi+rax*4], r8d
        add     rax, 1
        cmp     rax, rsi
        jne     .L5
.L1:
        ret
```

Assembly — -O2 (Loop Split):

```
pre_example:
        mov     eax, ecx
        test    ecx, ecx
        jle     .L1
        movsx   rcx, ecx
        cmp     eax, 10
        jle     .L6                 ; branch evaluated once; separate loop paths
        imul    edx, esi
        xor     eax, eax
        xor     esi, esi
.L4:
        add     esi, edx
        mov     DWORD PTR [rdi+rax*4], esi
        ...
.L6:
        xor     edx, edx
        xor     eax, eax
.L3:
        mov     DWORD PTR [rdi+rax*4], edx
        ...
```

Assembly — -O3 (Vectorized):

```
pre_example:
        test    ecx, ecx
        jle     .L1
        cmp     ecx, 10
        jg      .L3                ; n>10 vectorised SIMD path
        ...
.L8:
        pshufd  xmm1, xmm4, 0
        paddd   xmm1, xmm0         ; 4 elements processed per cycle via SSE
        movups  XMMWORD PTR [rax-16], xmm1
        cmp     rax, rsi
        jne     .L8
        ...
.L1:
        ret
```

Assembly — -Os (Size Optimized):

```
pre_example:
        imul    esi, edx            ; x*y hoisted before loop
        xor     eax, eax
.L2:
        cmp     ecx, eax
        jle     .L7
        lea     edx, [rsi+rax]
        cmp     ecx, 10
        cmovle  edx, eax
        mov     DWORD PTR [rdi+rax*4], edx
        inc     rax
        jmp     .L2
.L7:
        ret
```

Inference:

Optimization Level	Compiler Behaviour & Observations
`-O0`	x*y is recomputed on every loop iteration via imul eax inside the loop body, even though both x and y are invariant throughout the loop. The n>10 branch condition is also re-evaluated on every pass. This is maximally redundant — identical work repeated n times.
`-O1` *(PRE + LICM)*	Loop Invariant Code Motion (LICM) proves x*y does not change across iterations. It is hoisted above the loop (imul edx, esi before .L5) and computed exactly once. The n>10 condition remains inside the loop but is handled branchlessly via cmovle.

Optimization Level	Compiler Behaviour & Observations
`-O2` *(Loop split)*	The n>10 check is evaluated once before entering the loop. The loop is split into two completely separate code paths: one for n>10 (with x*y pre-added each step) and one for n<=10 (simple index store). No conditional instruction exists inside either loop.
`-O3` *(SIMD vectorised)*	Both loop paths are vectorised using SSE2 instructions (paddd xmm0, pshufd, movups). The n>10 path processes four array elements per clock cycle via 128-bit packed addition.
`-Os`	Keeps the PRE invariant hoist (imul esi, edx before the loop) but skips loop splitting and vectorisation. A compact single loop with cmovle inside provides correct behaviour at minimum code size.

13. Redundant Assignment Elimination

Original Program (C):

```
#include <stdio.h>

int main() {
    int i = 1;
    int *x = &i;
    x = &i;       // redundant: identical to the assignment above
    fprintf(stdout, "i = %d. x = %d.\n", i, *x);
    return 0;
}
```

Assembly — -O0 (No Optimization):

```
.LC0:
        .string "i = %d. x = %d.\n"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-12], 1
        lea     rax, [rbp-12]
        mov     QWORD PTR [rbp-8], rax     ; first:  x = &i
        lea     rax, [rbp-12]
        mov     QWORD PTR [rbp-8], rax     ; second: x = &i  (wasted write)
        mov     rax, QWORD PTR [rbp-8]
        mov     ecx, DWORD PTR [rax]
        mov     edx, DWORD PTR [rbp-12]
        mov     rax, QWORD PTR stdout[rip]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, rax
        mov     eax, 0
        call    fprintf
        mov     eax, 0
        leave
        ret
```

Assembly — -O1:

```
.LC0:
        .string "i = %d. x = %d.\n"
main:
        sub     rsp, 8
        mov     ecx, 1
        mov     edx, 1
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stdout[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 0
        add     rsp, 8
        ret
```

Assembly — -O2:

```
.LC0:
        .string "i = %d. x = %d.\n"
main:
        sub     rsp, 8
        mov     ecx, 1
        mov     edx, 1
        xor     eax, eax
        mov     rdi, QWORD PTR stdout[rip]
        mov     esi, OFFSET FLAT:.LC0
        call    fprintf
        xor     eax, eax
        add     rsp, 8
        ret
```

Assembly — -O3:

```
.LC0:
        .string "i = %d. x = %d.\n"
main:
        sub     rsp, 8
        mov     ecx, 1
        mov     edx, 1
        xor     eax, eax
        mov     rdi, QWORD PTR stdout[rip]
        mov     esi, OFFSET FLAT:.LC0
        call    fprintf
        xor     eax, eax
        add     rsp, 8
        ret
```

Assembly — -Os (Size Optimized):

```
.LC0:
        .string "i = %d. x = %d.\n"
main:
        push    rax
        mov     rdi, QWORD PTR stdout[rip]
        mov     edx, 1
        xor     eax, eax
        mov     ecx, 1
        mov     esi, OFFSET FLAT:.LC0
        call    fprintf
        xor     eax, eax
        pop     rdx
        ret
```

Inference:

Optimization Level	Compiler Behaviour & Observations
-O0	The instruction sequence lea rax, [rbp-12] followed by mov QWORD PTR [rbp-8], rax appears identically twice back-to-back. This is a direct translation of the two C assignments. The second write to [rbp-8] stores exactly the same address that was written on the immediately preceding instruction — CPU cycles wasted with zero change to program state.
-O1 *(Redundant assignment eliminated)*	Alias propagation proves the second x = &i is a duplicate: same source, same destination, no intervening modifications. The redundant store is deleted. The combined alias and load-store analysis (as in Technique 10) further proves *x and i are always 1, eliminating the pointer, the stack allocation, and all memory operations. Both arguments become compile-time constants: mov ecx, 1 and mov edx, 1.
-O2	Same redundant-assignment elimination and alias folding as -O1. Additionally uses xor eax, eax instead of mov eax, 0 and reorders argument register assignments for better pipeline scheduling.
-O3 *(Identical to -O2)*	No further redundant stores remain to eliminate. Assembly output is identical to -O2.
-Os	Same elimination applied. Stack frame managed with push rax / pop rdx and argument registers loaded in a size-optimal order to minimise instruction byte count.

Complete Optimization Comparison Table

Technique	-O0	-O1	-O2	-O3	-Os	Status Map
Dead Code Elimination (DCE)	**No**	**Yes**	**Yes**	**Yes**	**Yes**	O1 = O2 = O3 = Os
Tail Call Optimization (TCO)	**No**	**No**	**Yes**	**Yes**	**No**	O2 = O3
Loop Unrolling	**No**	**No**	**Yes**	**Yes**	**No**	O2 = O3
Loop Vectorization	**No**	**No**	**No**	**Yes**	**No**	O3 only
Switch Statement Transformation	**No**	**No**	**Yes**	**Yes**	**No**	O2 = O3
Algebraic / Constant Folding	**No**	**No**	**Yes**	**Yes**	**No**	O2 = O3
Bitwise Optimization	**No**	**No**	**Yes**	**Yes**	**No**	O2 = O3

Technique	-O0	-O1	-O2	-O3	-Os	Status Map
IPCP	**No**	**No**	**No**	**Yes**	**No**	O3 only
Arithmetic Strength Reduction	**No**	**No**	**Yes**	**Yes**	**No**	O2 = O3
Pointer Aliasing Optimization	**No**	**No**	**Yes**	**Yes**	**No**	O2 = O3
Instruction Scheduling	**No**	**Yes**	**Yes**	**Yes**	**Yes**	O1 = O2 = O3 = Os
Partial Redundancy Elimination (PRE)	**No**	**Yes**	**Yes**	**Yes**	**Yes**	O1 = O2 = O3 = Os
Redundant Assignment Elimination	**No**	**Yes**	**Yes**	**Yes**	**Yes**	O1 = O2 = O3 = Os

Conclusion

Throughout this compiler design case study, we evaluated 13 fundamental optimization techniques applied by GCC across different optimization levels (-O0 through -Os). The transformation gradients from -O0 to -O3 demonstrate the compiler's ability to eliminate dead computations, hoist loop-invariant expressions, flatten recursive call chains, fold constants at compile time, and vectorize sequential operations over SIMD channels.

Each technique targets a specific category of inefficiency. Dead Code Elimination and Constant Folding operate on provably static facts; Tail Call Optimization and Partial Redundancy Elimination reshape control flow to reduce overhead; Instruction Scheduling and Loop Vectorization exploit the parallel execution capabilities of modern superscalar processors. Together, these passes transform verbose, readable high-level source code into tightly packed machine code that makes full use of available hardware resources.

The assembly comparisons clearly illustrate the gap between naive code generation and deeply optimized output. What begins as fifteen or more instructions at -O0 can reduce to as few as two at -O1, and in some cases to a single precomputed constant at -O3. This underlines why compiler optimization is an indispensable component of modern systems programming and why understanding it is essential for writing performance-critical software.