

6.172  
Performance  
Engineering  
of Software  
Systems



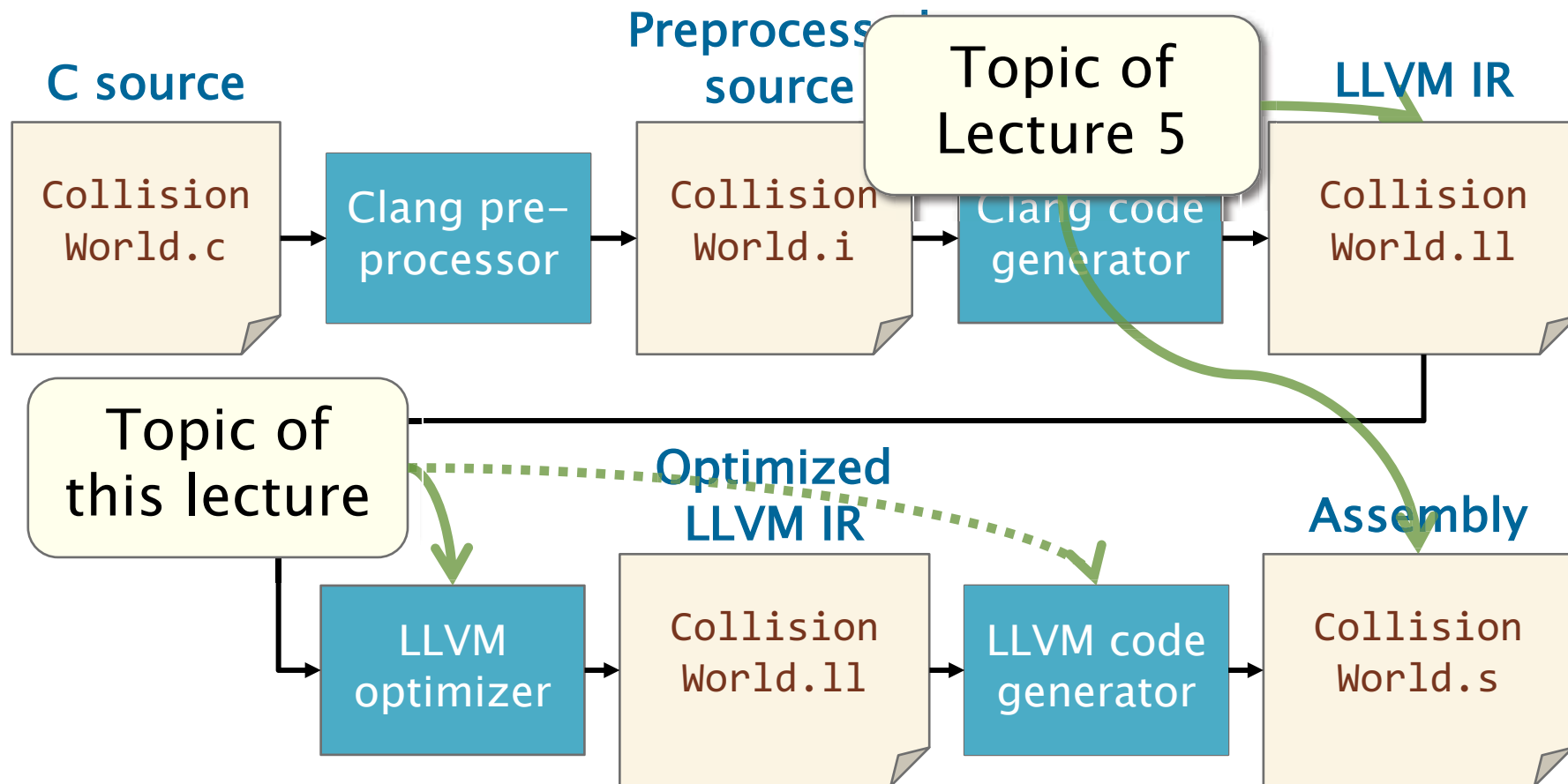
LECTURE 9

What Compilers Can and  
Cannot Do

Tao B. Schardl

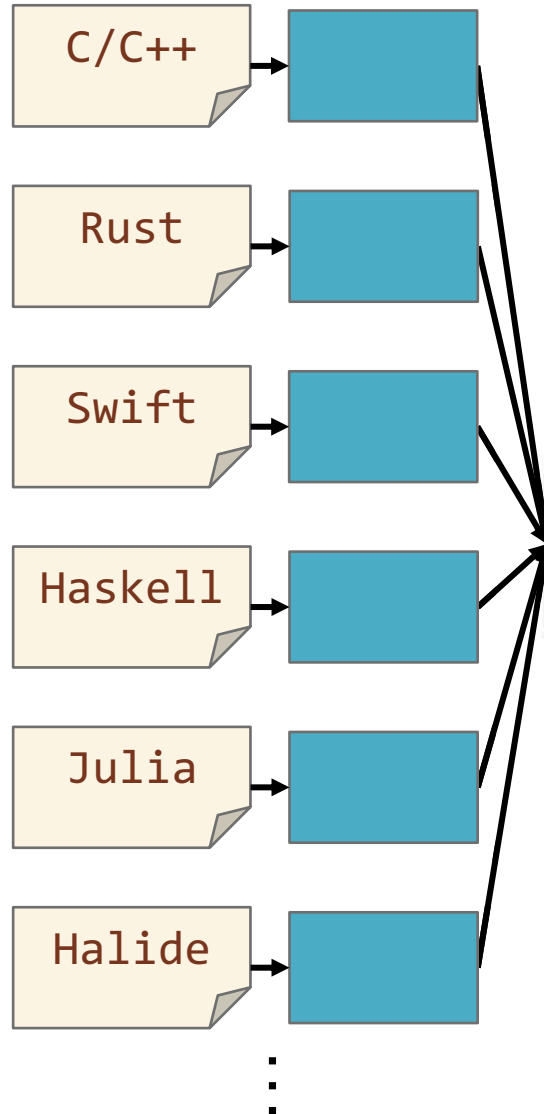
# Clang/LLVM Compilation Pipeline

This lecture completes more of the story from Lecture 5 about the compilation process.



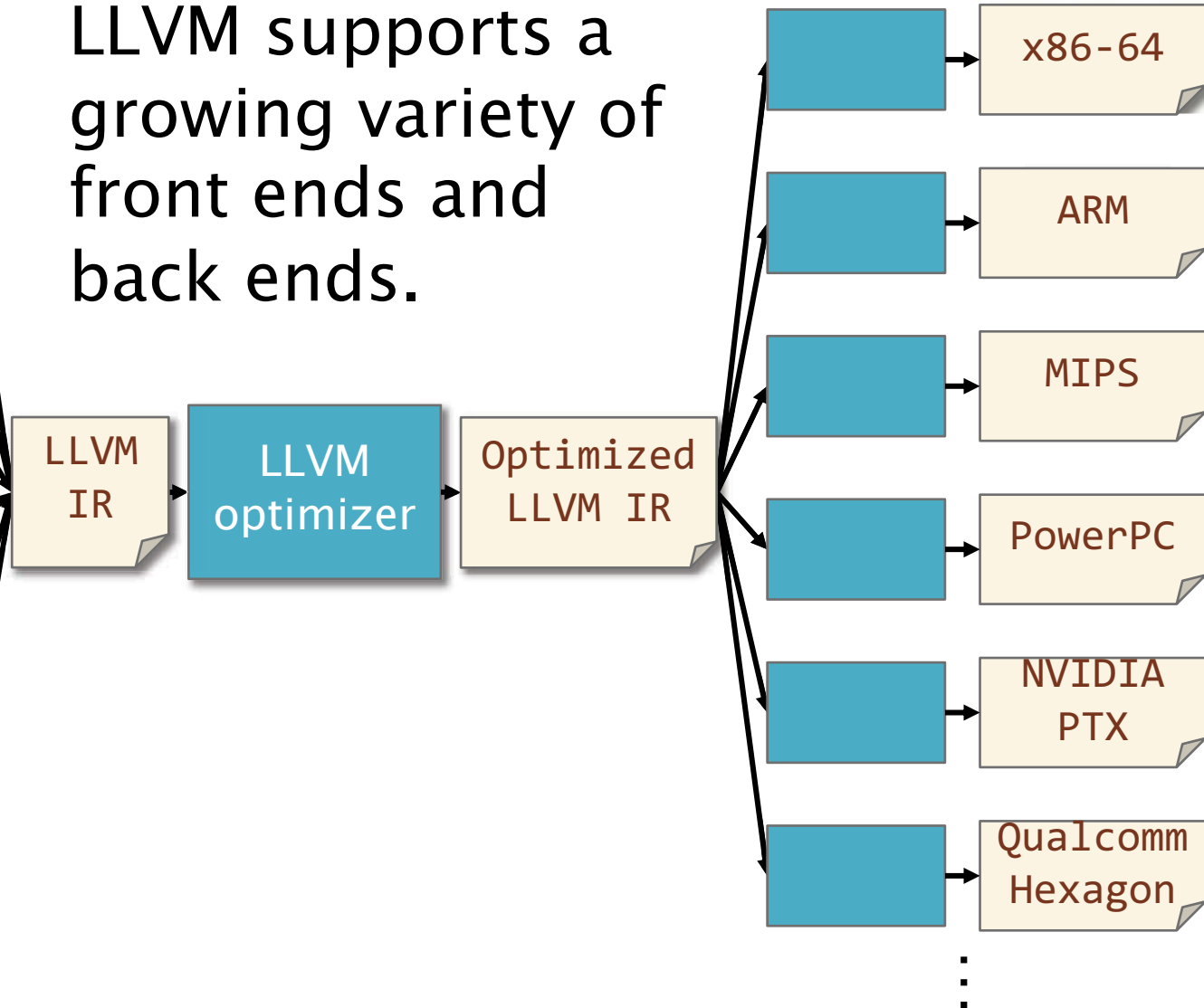
# Larger Context of the Compiler

## Front ends



LLVM supports a growing variety of front ends and back ends.

## Back ends

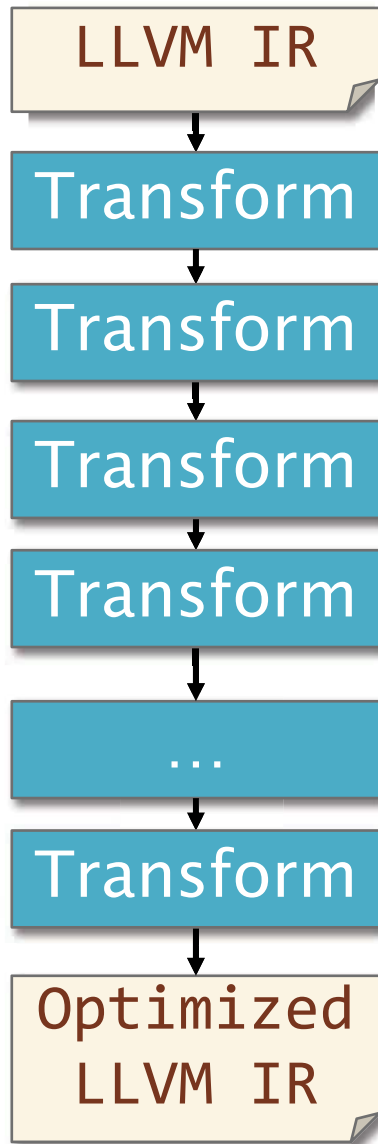


# Why Look Inside the Compiler?

Why study the compiler optimizations?

- Compilers can have a **big impact** on software performance.
- Compilers can **save** you performance-engineering work.
- Compilers help ensure that simple, readable, and maintainable code is **fast**.
- You can understand the **differences** between the source code and the IR or assembly.
- Compilers can make **mistakes**.
- Understanding compilers can help you use them more **effectively**.

# Simple Model of the Compiler



An optimizing compiler performs a sequence of **transformation passes** on the code.

- Each transformation pass **analyzes and edits** the code to try to **optimize** the code's performance.
- A transformation pass might run **multiple times**.
- Passes run in a **predetermined order** that seems to work well most of the time.

# Compiler Reports

Clang/LLVM can produce **reports** for many of its transformation passes, not just vectorization:

- Rpass**=<string>: Produces reports of which optimizations matching <string> were successful.
- Rpass-missed**=<string>: Produces reports of which optimizations matching <string> were not successful.
- Rpass-analysis**=<string>: Produces reports of the analyses performed by optimizations matching <string>.

The argument <string> is a **regular expression**. To see the whole report, use “**.\***” as the string.

# An Example Compiler Report

```
$ clang -O3 -std=gnu99 -c CollisionWorld.c -Rpass=.* -Rpass-analysis=.*
```

```
...  
CollisionWorld.c:92:39: remark: hoisting load [-Rpass=licm]
```

```
for (int i = 0; i < collisionWorld->numOfLines; i++) {
```

^

```
CollisionWorld.c:91:6: remark: load of type i32 eliminated [-Rpass=gvn]
```

```
void CollisionWorld_lineWallCollision(CollisionWorld* collisionWorld) {
```

^

```
CollisionWorld.c:79:3: remark: CollisionWorld_lineWallCollision can be inlined into  
CollisionWorld_updateLines with cost=245 (threshold=250) [-Rpass-analysis=inline]
```

```
CollisionWorld_lineWallCollision(collisionWorld);
```

^

```
CollisionWorld.c:79:3: remark: CollisionWorld_lineWallCollision inlined into  
CollisionWorld_updateLines [-Rpass=inline]
```

```
CollisionWorld.c:135:5: remark: loop not vectorized: loop control flow is not  
understood by vectorizer [-Rpass-analysis=loop-vectorize]
```

```
for (int j = i + 1; j < collisionWorld->numOfLines; j++) {
```

^

```
...
```

# Compiler Reports: Good and Bad

The **good news**: The compiler can tell you a lot about what it's doing.

- Many transformation passes in LLVM can report places where they successfully transform code.
- Many can also report the conclusions of their analysis.

The **bad news**: Reports can be hard to understand.

- The reports can be **long** and use LLVM **jargon**.
- Not all transformation passes generate reports.
- Reports don't always tell the whole story.

We want context for understanding these reports.



# Outline

- Example compiler optimizations
  - Optimizing a scalar
  - Optimizing a structure
  - Optimizing function calls
  - Optimizing loops
- Diagnosing failures
  - Case Study 1
  - Case Study 2
  - Case Study 3

# OVERVIEW OF COMPILER OPTIMIZATIONS



# New Bentley Rules

## Compiler Optimizations

### Data structures

- ~~● Packing and encoding~~
- ~~● Augmentation~~
- ~~● Precomputation~~
- ~~● Compile-time initialization~~
- ~~● Caching~~
- ~~● Lazy evaluation~~
- ~~● Sparsity~~

### Loops

- Hoisting
- ~~● Sentinels~~
- Loop unrolling
- Loop fusion\*
- Eliminating wasted iterations\*

\*Restrictions may apply.

### Logic

- Constant folding and propagation
- Common-subexpression elimination
- Algebraic identities
- Short-circuiting
- Ordering tests\*
- ~~● Creating a fast path~~
- Combining tests\*

### Functions

- Inlining
- Tail-recursion elimination
- ~~● Coarsening recursion~~

# More Compiler Optimizations

## Data structures

- Register allocation
- Memory to registers
- Scalar replacement of aggregates
- Alignment

## Loops

- Vectorization
- Unswitching
- Idiom replacement
- Loop fission\*
- Loop skewing\*
- Loop tiling\*
- Loop interchange\*

\*In development in Clang/LLVM.

## Logic

- Elimination of redundant instructions
- Strength reduction
- Dead-code elimination
- Idiom replacement
- Branch reordering
- Global value numbering

## Functions

- Unswitching
- Argument elimination

**Moving target:** Compiler developers implement new optimizations over time.

# Arithmetic Opt's: C vs. LLVM IR

Most compiler optimizations happen on the compiler's **intermediate representation (IR)**, although not all of them.

EXAMPLE: Let `n` be a `uint32_t`.

## C code

```
uint32_t x = n * 8;
```

```
uint32_t y = n * 15;
```

```
uint32_t z = n / 71;
```

## LLVM IR

```
%2 = shl nsw i32 %0, 3
```

```
%3 = mul nsw i32 %0, 15
```

```
%4 = udiv i32 %0, 71
```

Register `%0` holds the value of `n`.

# Arithmetic Opt's: C vs. Assembly

Most compiler optimizations happen on the compiler's **intermediate representation (IR)**, although not all of them.

EXAMPLE: Let `n` be a `uint32_t`.

Register `%rdi` holds the value of `n`.

## C code

```
uint32_t x = n * 8;
```

```
uint32_t y = n * 15;
```

```
uint32_t z = n / 71;
```

**Magic number**  
equal to  $2^{38}/71+1$ .

## Assembly

```
leal    (,%rdi,8), %eax
```

```
leal    (%rdi,%rdi,4), %eax  
leal    (%rax,%rax,2), %eax
```

```
movl    %edi, %eax  
movl    $3871519817, %ecx  
imulq   %rax, %rcx  
shrq    $38, %rcx
```

# Example: N-Body Simulation

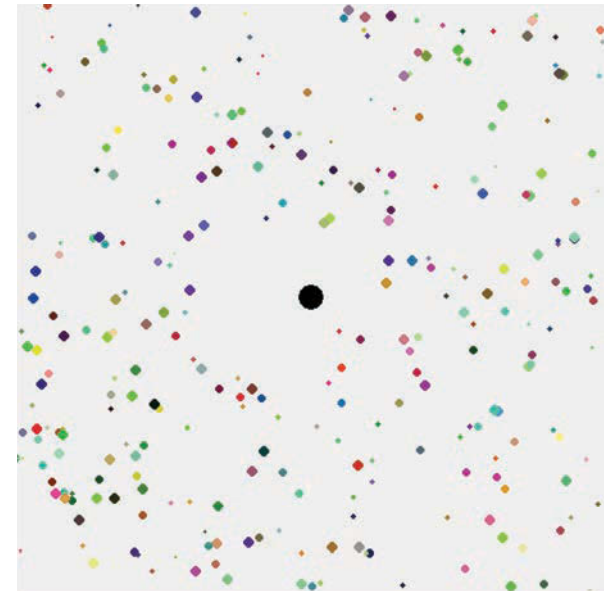
A lot of the compiler's capability comes from **combining** optimizations.

Let's work through an example.

**PROBLEM:** Simulate the behavior of  $n$  massive bodies in 2D space under the influence of gravity.

**LAW OF GRAVITATION:**

$$\mathbf{F}_{21} = (G m_1 m_2 / |\mathbf{r}_{12}|^2) \text{unit}(\mathbf{r}_{21})$$



# N-Body Simulation Code

Primary  
routine

```
void simulate(body_t *bodies, int nbodies,  
             int nsteps, int time_quantum) {  
    for (int i = 0; i < nsteps; ++i) {  
        calculate_forces(nbodies, bodies);  
        update_positions(nbodies, bodies,  
                        time_quantum);  
    }  
}
```

Data  
structures

```
typedef struct body_t {  
    // Position vector  
    vec_t position;  
    // Velocity vector  
    vec_t velocity;  
    // Force vector  
    vec_t force;  
    // Mass  
    double mass;  
} body_t;
```

```
typedef struct vec_t {  
    double x, y;  
} vec_t;
```



# Key Routine in N-Body Simulation

```
void update_positions(int nbodies, body_t *bodies,
                    double time_quantum) {
    for (int i = 0; i < nbodies; ++i) {
        // Compute the new velocity of bodies[i].
        vec_t new_velocity =
            vec_scale(bodies[i].force,
                    time_quantum / bodies[i].mass);
        // Update the position of bodies[i] based on
        // the average of its old and new velocity.
        bodies[i].position =
            vec_add(bodies[i].position,
                    vec_scale(vec_add(bodies[i].velocity,
                                    new_velocity),
                            time_quantum / 2.0));
        // Set the new velocity of bodies[i].
        bodies[i].velocity = new_velocity;
    }
}
```

# Basic Routines for 2D Vectors

The top-level methods invoke a few simple routines on 2D vectors.

```
typedef struct vec_t {
    double x, y;
} vec_t;

static vec_t vec_add(vec_t a, vec_t b) {
    vec_t sum = { a.x + b.x, a.y + b.y };
    return sum;
}

static vec_t vec_scale(vec_t v, double a) {
    vec_t scaled = { v.x * a, v.y * a };
    return scaled;
}

static double vec_length2(vec_t v) {
    return v.x * v.x + v.y * v.y;
}
```

# Compiling with No Optimizations

```
typedef struct vec_t { double x, y; } vec_t;
```

C code

```
static vec_t vec_scale(vec_t v, double s) {  
    vec_t scaled = {  
        s*v.x, s*v.y  
    };  
    return scaled;  
}
```

LLVM IR  
compiled  
at -O0

```
define internal @vec_scale(double, double, double) #0 {  
    %4 = alloca @struct.vec_t, align 8  
    %5 = alloca @struct.vec_t, align 8  
    %6 = alloca double, align 8  
    %7 = alloca @struct.vec_t, align 8  
    %8 = bitcast @struct.vec_t* %5 to { double, double }*  
    %9 = getelementptr inbounds { double, double }, { double, double }* %8, i32 0, i32 0  
    store double %0, double* %9, align 8  
    %10 = getelementptr inbounds { double, double }, { double, double }* %8, i32 0, i32 1  
    store double %1, double* %10, align 8  
    store double %2, double* %6, align 8  
    %11 = getelementptr inbounds @struct.vec_t, @struct.vec_t* %7, i32 0, i32 0  
    %12 = getelementptr inbounds @struct.vec_t, @struct.vec_t* %5, i32 0, i32 0  
    %13 = load double, double* %12, align 8  
    %14 = load double, double* %6, align 8  
    %15 = fmul double %13, %14  
    store double %15, double* %11, align 8  
    %16 = getelementptr inbounds @struct.vec_t, @struct.vec_t* %7, i32 0, i32 1  
    %17 = getelementptr inbounds @struct.vec_t, @struct.vec_t* %5, i32 0, i32 1  
    %18 = load double, double* %17, align 8  
    %19 = load double, double* %6, align 8  
    %20 = fmul double %18, %19  
    store double %20, double* %16, align 8  
    %21 = bitcast @struct.vec_t* %4 to i8*  
    %22 = bitcast @struct.vec_t* %7 to i8*  
    call void @llvm.memcpy.p0i8.p0i8.i64(i8* %21, i8* %22, i64 16, i32 8, i1 false)  
    %23 = bitcast @struct.vec_t* %4 to { double, double }*  
    %24 = load { double, double }, { double, double }* %23, align 8  
    ret { double, double } %24  
}
```

# Compiling with -O1

```
typedef struct vec_t { double x, y; } vec_t;  
  
static vec_t vec_scale(vec_t v, double a) {  
    vec_t scaled = { v.x * a, v.y * a };
```

## C code

Let's see how the compiler optimizes the original code.

Struct argument occupies two function parameters.

Compute the fields of the new struct.

LLVM IR  
compiled  
at -O1

```
@vec_scale @llvm.global_ctors { double, double, double }  
%4 = fmul double %0, %2  
%5 = fmul double %1, %2  
%.fca.0.insert = insertvalue { double, double }  
    undef, double %4, 0  
%.fca.1.insert = insertvalue { double, double }  
    %.fca.0.insert, double %5, 1  
ret { double, double } %fca.1.insert  
}
```

Insert fields into destination register.

# OPTIMIZING A SCALAR



# Handling One Argument, -O0 Code

```
static vec_t vec_scale(vec_t v, double a) {  
    vec_t scaled = { v.x * a, v.y * a };  
    return scaled;  
}
```

```
define internal { double, double }  
@vec_scale(double, double, double) #0 {
```

```
...  
%6 = alloca double, align 8
```

```
...  
store double %2, double* %6, align 8
```

```
...  
%14 = load double, double* %6, align 8  
%15 = fmul double %13, %14
```

```
...  
%19 = load double, double* %6, align 8  
%20 = fmul double %18, %19  
...  
}
```

Let's examine the parameter `a` in `vec_scale` at `-O0`.

Allocate stack storage.

Store `a` onto the stack.

Load `a` from the stack.

# Promoting Memory to Registers

**IDEA:** Replace the stack-allocated variable with the copy in the register.

```
define internal { double, double }  
@vec_scale(double, double, double) #0 {  
  ...  
  %6 = alloca double, align 8  
  ...  
  store double %2, double* %6, align 8  
  ...  
  %14 = load double, double* %6, align 8  
  %15 = fmul double %13, %14  
  ...  
  %19 = load double, double* %6, align 8  
  %20 = fmul double %18, %19  
  ...  
}
```

**Step 1:** Replace loaded values with original register.

**Step 2:** Remove dead code.



# Improvement So Far

## Before

```
define internal @vec_scale(double, double, double) #0 {
  %4 = alloca %struct.vec_t, align 8
  %5 = alloca %struct.vec_t, align 8
  %6 = alloca double, align 8
  %7 = alloca %struct.vec_t, align 8
  %8 = bitcast %struct.vec_t* %5 to { double, double }*
  %9 = getelementptr inbounds { double, double }, { double, double }* %8, i32 0, i32 0
  store double %0, double* %9, align 8
  %10 = getelementptr inbounds { double, double }, { double, double }* %8, i32 0, i32 0
  store double %1, double* %10, align 8
  %11 = getelementptr inbounds { double, double }, { double, double }* %8, i32 0, i32 1
  store double %2, double* %11, align 8
  %12 = getelementptr inbounds { double, double }, { double, double }* %8, i32 0, i32 0
  %13 = load double, double* %12, align 8
  %14 = load double, double* %11, align 8
  %15 = fmul double %13, double %14
  store double %15, double* %10, align 8
  %16 = getelementptr inbounds { double, double }, { double, double }* %8, i32 0, i32 1
  %17 = getelementptr inbounds { double, double }, { double, double }* %8, i32 0, i32 0
  %18 = load double, double* %17, align 8
  %19 = load double, double* %16, align 8
  %20 = fmul double %18, double %19
  store double %20, double* %11, align 8
  %21 = bitcast %struct.vec_t* %4 to { double, double }*
  %22 = bitcast %struct.vec_t* %6 to { double, double }*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %21, i8* %22, i64 16, i32 8, i1 false)
  %23 = bitcast %struct.vec_t* %4 to { double, double }*
  %24 = load { double, double }, { double, double }* %23, align 8
  ret { double, double } %24
}
```

## After

```
define internal @vec_scale(double, double, double) #0 {
  %4 = alloca %struct.vec_t, align 8
  %5 = alloca %struct.vec_t, align 8
  %6 = alloca %struct.vec_t, align 8
  %7 = bitcast %struct.vec_t* %5 to { double, double }*
  %8 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 0
  store double %0, double* %8, align 8
  %9 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 1
  store double %1, double* %9, align 8
  %10 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %6, i32 0, i32 0
  %11 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %6, i32 0, i32 1
  %12 = load double, double* %11, align 8
  %13 = fmul double %12, %2
  store double %13, double* %10, align 8
  %14 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %6, i32 0, i32 0
  %15 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %6, i32 0, i32 1
  %16 = load double, double* %15, align 8
  %17 = fmul double %16, %2
  store double %17, double* %14, align 8
  %18 = bitcast %struct.vec_t* %4 to { double, double }*
  %19 = bitcast %struct.vec_t* %6 to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %18, i8* %19, i64 16, i32 8, i1 false)
  %20 = bitcast %struct.vec_t* %4 to { double, double }*
  %21 = load { double, double }, { double, double }* %20, align 8
  ret { double, double } %21
}
```

Idea: Eliminate struct-type arguments and local variables as well.



# OPTIMIZING A STRUCTURE



# Removing Structures

**PROBLEM:** Structures are harder to handle because code operates on individual fields.

```
define internal { double, double } @vec_scale(double, double)
```

```
...  
%5 = alloca %struct.vec_t, align 8
```

Allocate stack storage for a struct.

```
...  
%7 = bitcast %struct.vec_t* %5 to { double, double }*  
%8 = getelementptr inbounds { double, double },  
    { double, double }* %7, i32 0, i32 0  
store double %0, double* %8, align 8
```

Store the first field.

```
%9 = getelementptr inbounds { double, double },  
    { double, double }* %7, i32 0, i32 1  
store double %1, double* %9, align 8
```

Store the second field.

```
...  
%11 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 0  
%12 = load double, double* %11, align 8  
%13 = fmul double %12, %2
```

Load the first field.

```
...  
%15 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 1  
%16 = load double, double* %15, align 8  
%17 = fmul double %16, %2
```

Load the second field.

```
...
```

# Removing Structures

**PROBLEM:** Structures are harder to handle because code operates on individual fields.

```
define internal { double, double } @vec_scale(double, double, double) #0 {  
  ...  
  %5 = alloca %struct.vec_t, align 8  
  ...  
  %7 = bitcast %struct.vec_t* %5 to { double, double }*  
  %8 = getelementptr inbounds { double, double },  
    { double, double }* %7, i32 0, i32 0  
  store double %0, double* %8, align 8  
  %9 = getelementptr inbounds { double, double },  
    { double, double }* %7, i32 0, i32 1  
  store double %1, double* %9, align 8  
  ...  
  %11 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 0  
  %12 = load double, double* %11, align 8  
  %13 = fmul double %12, %2  
  ...  
  %15 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 1  
  %16 = load double, double* %15, align 8  
  %17 = fmul double %16, %2  
  ...  
  ...  
  ...
```

**Idea:** Optimize individual fields of the aggregate type.

# Scalar Replacement of Aggregates

Let's consider just the first field.

```
define internal { double, double }
@vec_scale(double, double, double) #0 {
  ...
  %5 = alloca %struct.vec_t, align 8
  ...
  %7 = bitcast %struct.vec_t* %5 to { double, double }*
  %8 = getelementptr inbounds { double, double },
    { double, double }* %7, i32 0, i32 0
  store double %0, double* %8, align 8
  ...
  %11 = getelementptr inbounds %struct.vec_t,
    %struct.vec_t* %5, i32 0, i32 0
  %12 = load double, double* %11, align 8
  %13 = fmul double %12, %2
  ...
}
```

Both address calculations refer to the same struct field.

Question: What value will the load retrieve?

Answer: %0.

Replace the use of that field with the original register value.

# Scalar Replacement of Aggregates

Let's consider just the first field.

```
define internal { double, double }
@vec_scale(double, double, double) #0 {
  ...
  %5 = alloca %struct.vec_t, align 8
  ...
  %7 = bitcast %struct.vec_t* %5 to { double, double }*
%8 = getelementptr inbounds { double, double },
  { double, double }* %7, i32 0, i32 0
store double %0, double* %8, align 8
  ...
%11 = getelementptr inbounds %struct.vec_t,
  %struct.vec_t* %5, i32 0, i32 0
%12 = load double, double* %11, align 8
  %13 = fmul double %0, %2
  ...
}
```

Remove now-dead code.

# Scalar Replacement of Aggregates

The second field can be optimized similarly.

```
define internal { double, double }
@vec_scale(double, double, double) #0 {
  ...
  %5 = alloca %struct.vec_t, align 8
  ...
  %7 = bitcast %struct.vec_t* %5 to { double, double }*
  %9 = getelementptr inbounds { double, double },
    { double, double }* %7, i32 0, i32 1
  store double %1, double* %9, align 8
  ...
  %13 = fmul double %0, %2
  %15 = getelementptr inbounds %struct.vec_t,
    %struct.vec_t* %5, i32 0, i32 1
  %16 = load double, double* %15, align 8
  %17 = fmul double %16, %2
  ...
  %1
}
```



# Scalar Replacement of Aggregates

A similar but more complicated optimization can optimize the **return-value structure**.

```
define internal { double, double }
@vec_scale(double, double, double) #0 {
    %6 = alloca %struct.vec_t, align 8
    %10 = getelementptr inbounds %struct.vec_t,
        %struct.vec_t* %6, i32 0, i32 0
    %13 = fmul double %0, %2
    store double %13, double* %10, align 8
    %14 = getelementptr inbounds %struct.vec_t,
        %struct.vec_t* %6, i32 0, i32 1
    %17 = fmul double %1, %2
    store double %17, double* %14, align 8
    %20 = bitcast %struct.vec_t* %6 to { double, double }*
    %21 = load { double, double },
        { double, double }* %20, align 8
    ret { double, double } %21
}
```

# Result of Optimizations

## Result of optimizing all aggregate variables

```
define internal { double, double }  
@vec_scale(double, double, double) #0 {  
    %4 = fmul double %0, %2  
    %5 = fmul double %1, %2  
    %.fca.0.insert = insertvalue { double, double }  
        undef, double %4, 0  
    %.fca.1.insert = insertvalue { double, double }  
        %.fca.0.insert, double %5, 1  
    ret { double, double } %.fca.1.insert  
}
```

**SUMMARY:** Compilers transform data structures to store as much as possible in registers.



# OPTIMIZING FUNCTION CALLS




# Example: Updating Positions

Let's see how compilers optimize function calls.

C code from `update_positions`

```
vec_add(bodies[i].position,  
        vec_scale(vec_add(bodies[i].velocity,  
                          new_velocity),  
                  time_quantum / 2.0));
```



LLVM IR

```
%24 = extractvalue { double, double } %23, 0  
%25 = extractvalue { double, double } %23, 1  
%26 = fmul double %2, 5.000000e-01  
%27 = call { double, double } @vec_scale(double %24,  
double %25, double %26)
```

# Function Inlining

## LLVM IR snippet from update\_positions

```
%24 = extractvalue { double, double } %23, 0
%25 = extractvalue { double, double } %23, 1
%26 = fmul double %2, 5.000000e-01
%27 = call { double, double } @vec_scale(double %24,
double %25, double %26)
```

## LLVM IR for vec\_scale

```
define internal { double, double }
@vec_scale(double, double, double) #0
    %4 = fmul double %0, %2
    %5 = fmul double %1, %2
    %.fca.0.insert = insertvalue { double, double } undef,
double %4, 0
    %.fca.1.insert = insertvalue { double, double }
%.fca.0.insert, double %5, 1
    ret { double, double } %.fca.1.insert
}
```

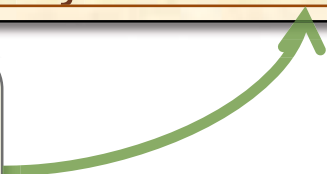
**Idea:** The code for `vec_scale` is small, so **copy-paste** it into the **call site**.

# Function Inlining

## LLVM IR snippet from update\_positions

```
%24 = extractvalue { double, double } %23, 0
%25 = extractvalue { double, double } %23, 1
%26 = fmul double %2, 5.000000e-01
%27 = call { double, double }
  @vec_scale(double %24, double %25, double %26)
%4.in = fmul double %24, %26
%5.in = fmul double %25, %26
%27 = insertvalue { double, double } undef, double %4.in, 0
%27 = insertvalue { double, double } %27, double %5.in, 1
ret { double, double } %.fca.1.insert
```

Step 1: Copy  
code from  
vec\_scale.



Step 2: Remove  
call and return.

# Further Optimization

Function inlining enables more optimizations.

LLVM IR snippet from `update_positions`

```
%24 = extractvalue { double, double } %2, 0
%25 = extractvalue { double, double } %2, 1
%26 = fmul double %2, 5.000000
%4.in = fmul double %24, %26
%5.in = fmul double %25, %26
```

These instructions pack struct fields and then immediately unpack them.

```
%27 = insertvalue { double, double } undef, double %4.in, 0
%27 = insertvalue { double, double } %27, double %5.in, 1
%28 = extractvalue { double, double } %27, 0
%29 = extractvalue { double, double } %27, 1
```

Idea: Remove these useless operations.

# Sequences of Function Calls

C code

```
vec_add(bodies[i].position,  
        vec_scale(vec_add(bodies[i].velocity,  
                        new_velocity),  
                  time_quantum / 2.0));
```

LLVM IR

```
%23 = call { double, double } @vec_add(double %20,  
    double %22, double %17, double %18)
```

```
%24 = extractvalue { double, double } %23, 0
```

```
%25 = extractvalue { double, double } %23, 1
```

```
%26 = fmul double %2, 5.000000e-01
```

```
%4.in = fmul double %24, %26
```

```
%5.in = fmul double %25, %26
```

...

```
%34 = call { double, double } @vec_add(double %33,  
    double %4.in, double %5.in)
```

Idea: Inline  
vec\_add as well.

...and then remove  
additional useless  
instructions...

# Sequences of Function Calls

C code

```
vec_add(bodies[i].position,  
        vec_scale(vec_add(bodies[i].velocity,  
                        new_velocity),  
                  time_quantum / 2.0));
```

Optimized LLVM IR

```
%22 = fadd double %19, %16  
%23 = fadd double %21, %17  
%26 = fmul double %2, 5.000000e-01  
%4.in = fmul double %24, %26  
%5.in = fmul double %25, %26  
...  
%31 = fadd double %28, %4.in  
%32 = fadd double %30, %5.in
```

# Equivalent C Code

C code

```
vec_add(bodies[i].position,  
        vec_scale(vec_add(bodies[i].velocity,  
                          new_velocity),  
                  time_quantum / 2.0));
```

C equivalent of optimized LLVM IR

```
double scale = time_quantum / 2.0  
double xv = bodies[i].velocity.x + new_velocity.x;  
double yv = bodies[i].velocity.y + new_velocity.y;  
double sxv = xv * scale;  
double syv = yv * scale;  
double new_x = bodies[i].position.x + sxv;  
double new_y = bodies[i].position.y + syv;
```

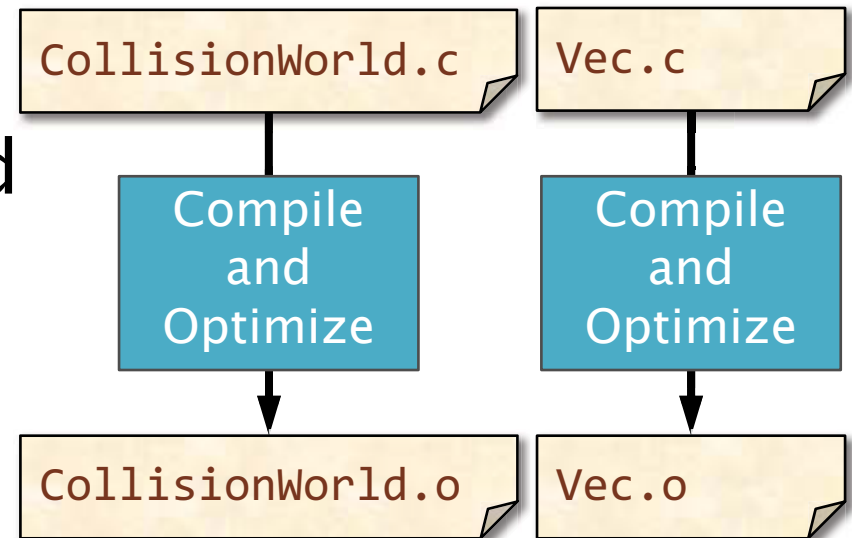
**SUMMARY:** Function inlining and additional transformations can **eliminate** the cost of the function abstraction.



# Problems with Function Inlining

Why doesn't the compiler inline **all** function calls?

- Some function calls, such as recursive calls, **cannot be inlined** except in special cases, e.g., “recursive tail calls.”
- The compiler cannot inline a function defined in another **compilation unit** unless one uses **whole-program optimization**.
- Function inlining can **increase code size**, which can hurt performance.



# Controlling Function Inlining

**QUESTION:** How does the compiler **know** whether or not inlining a function will hurt performance?

**ANSWER:** It doesn't know. It makes a **best guess** based on **heuristics**, such as the function's size.

Tips for controlling function inlining:

- Mark functions that should **always** be inlined with `__attribute__((always_inline))`.
- Mark functions that should **never** be inlined with `__attribute__((no_inline))`.
- Use **link-time optimization (LTO)** to enable whole-program optimization.

# LOOP OPTIMIZATIONS



# Loop Optimizations

Compilers also perform a variety of transformations on **loops**. Why?

Loops account for a lot of execution time.

Consider this thought experiment:

- Consider a **2** GHz processor with **16** cores executing **1** instruction per cycle.
- Suppose a program contains  $2^{40}$  instructions and ample parallelism for **16** cores, but it's all **simple straight-line code**, i.e., no loops.
- **QUESTION:** How long does the code take to run?

**ANSWER:** 32 seconds!

# Example: Calculating Forces

You have already seen vectorization. Let us look at another common optimization on loops: **code hoisting**, a.k.a., **loop-invariant-code motion (LICM)**.

C code from n-body simulation

```
void calculate_forces(int nbodies, body_t *bodies) {
    for (int i = 0; i < nbodies; ++i) {
        for (int j = 0; j < nbodies; ++j) {
            if (i == j) continue;
            add_force(&bodies[i],
                    calculate_force(&bodies[i], &bodies[j]));
        }
    }
}
```

# Calculating Forces: LLVM IR

## Doubly-nested loop in LLVM IR

```
; <label>:7:          ; preds = %9, %4  
%8 = phi i64 [ 0, %4 ], [ %10, %9 ]  
br label %12
```

```
; <label>:12:         ; preds = %48, %7  
%13 = phi i64 [ 0, %7 ], [ %49, %48 ]  
%14 = icmp eq i64 %8, %13  
br i1 %14, label %48, label %15
```

```
; <label>:15:         ; preds = %12  
%16 = getelementptr inbounds %struct.body_t,  
%struct.body_t* %1, i64 %8, i32 0, i32 0  
%17 = load double, double* %16, align 8
```

```
br label %48
```

```
; <label>:48:        ; preds = %15, %12  
%49 = add nuw nsw i64 %13, 1  
%50 = icmp eq i64 %49, %5  
br i1 %50, label %9, label %12
```

```
; <label>:9:         ; preds = %48  
%10 = add nuw nsw i64 %8, 1  
%11 = icmp eq i64 %10, %5  
br i1 %11, label %6, label %7
```

Loop body

Outer-loop control

Inner-loop control

# Zooming in on the LLVM IR

Top part of doubly-nested loop.

```
; <label>:7: ; preds = %9, %4  
%8 = phi i64 [ 0, %4 ], [ %10, %9 ]  
br label %12
```

```
; <label>:12: ; preds = %48, %7  
%13 = phi i64 [ 0, %7 ], [ %49, %48 ]  
%14 = icmp eq i64 %8, %13  
br i1 %14, label %48, label %15
```

Outer-loop iteration variable

Inner-loop iteration variable

```
; <label>:15: ; preds = %12  
%16 = getelementptr inbounds %struct.body_t, %struct.body_t*  
%1, i64 %8, i32 0, i32 0  
%18 = getelementptr inbounds %struct.body_t, %struct.body_t*  
%1, i64 %8, i32 0, i32 1  
%20 = getelementptr inbounds %struct.body_t, %struct.body_t*
```

Some address calculations in the loop body only depend on the outer-loop iteration variable.

Idea: Move them out of the inner-loop body!

# Hoisting Address Calculations

Top part of transformed doubly-nested loop.

```
; <label>:7:                                ; preds = %9, %4  
%8 = phi i64 [ 0, %4 ], [ %10, %9 ]  
%16 = getelementptr inbounds %struct.body_t,  
%struct.body_t* %1, i64 %8, i32 0, i32 0  
%18 = getelementptr inbounds %struct.body_t,  
%struct.body_t* %1, i64 %8, i32 0, i32 1  
br label %12
```

```
; <label>:12:                                ; preds = %48, %7  
%13 = phi i64 [ 0, %7 ], [ %10, %9 ]  
%14 = icmp eq i64 %8, %13  
br i1 %14, label %48, label %15
```

```
; <label>:15:  
%20 = getelementptr inbounds %struct.body_t*  
%1, i64 %13, i32 0, i32 0  
%22 = getelementptr inbounds %struct.body_t, %struct.body_t*  
%1, i64 %13, i32 0, i32 1  
...
```

Hoisted calculations are performed just once per iteration of the outer loop.



# Code Hoisting: Equivalent C

Before

```
for (int i = 0; i < nbodies; ++i) {  
    for (int j = 0; j < nbodies; ++j) {  
        if (i == j) continue;  
        add_force(&bodies[i],  
                 calculate_force(&bodies[i], &bodies[j]));  
    }  
}
```

After

```
for (int i = 0; i < nbodies; ++i) {  
    body_t *bi = &bodies[i];  
    for (int j = 0; j < nbodies; ++j) {  
        if (i == j) continue;  
        add_force(bi, calculate_force(bi, &bodies[j]));  
    }  
}
```

In general, if the compiler can prove some calculation is **loop-invariant**, it will attempt to hoist the code out of the loop.

# Summary: What the Compiler Does

Compilers transform code through a sequence of **transformation passes**.

- The compiler looks through the code and applies **mechanical** transformations where it can.
- Many transformations resemble **Bentley-rule** work optimizations.
- One transformation can **enable** other transformations.

Compilers perform many more transformations than those shown in this lecture.

# Something the Compiler Cannot Do

```
// Calculate forces between all of the bodies in the
// simulation for all pairs.
void calculate_forces(int nbodies, body_t *bodies) {
    for (int i = 0; i < nbodies; ++i) {
        for (int j = 0; j < nbodies; ++j) {
            // Update the force vector on bodies[i]
            // exerted by bodies[j].
            if (i == j) continue;
            add_force(&bodies[i],
                    calculate_force(&bodies[i], &bodies[j]));
        }
    }
}
```

The compiler is unlikely to automatically exploit **symmetry** in this problem, i.e., that  $\mathbf{F}_{12} = -\mathbf{F}_{21}$ .

# DIAGNOSING FAILURES: CASE STUDY 1



# Vectorization

QUESTION: Does the following loop vectorize?

C code

```
void daxpy(double *y, double a,
           double *x, int64_t n) {
    for (int64_t i = 0; i < n; ++i)
        y[i] += a * x[i];
}
```

What does the report say?

```
$ clang -O3 -c daxpy.c -Rpass=vector -Rpass-analysis=vector
daxpy.c:6:3: remark: vectorized loop (vectorization width: 2, interleaved count: 2) [-Rpass=loop-vectorize]
    for (int64_t i = 0; i < n; ++i)
    ^
```

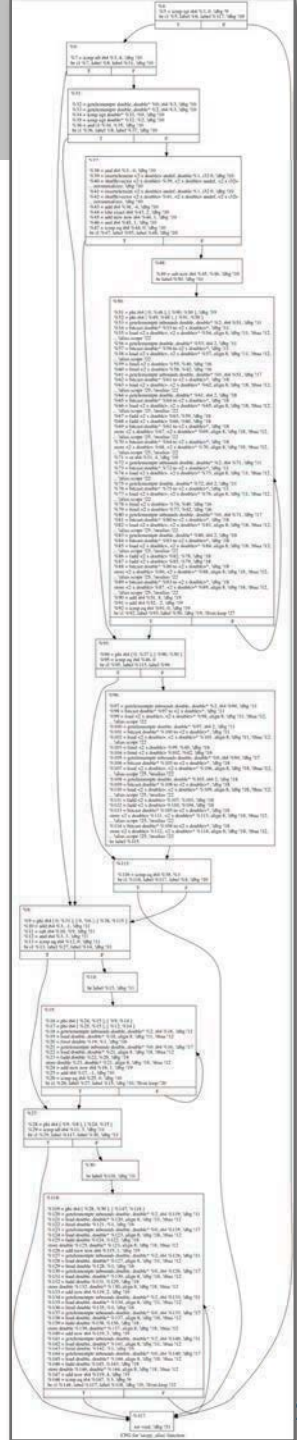
# Actual Compiled Code

The code generated by `-O2` optimization is **complicated**.

## C code

```
void daxpy(double *y, double a,
           double *x, int64_t n) {
    for (int64_t i = 0; i < n; ++i)
        y[i] += a * x[i];
}
```

## Actual control-flow graph



# Multiple Loops

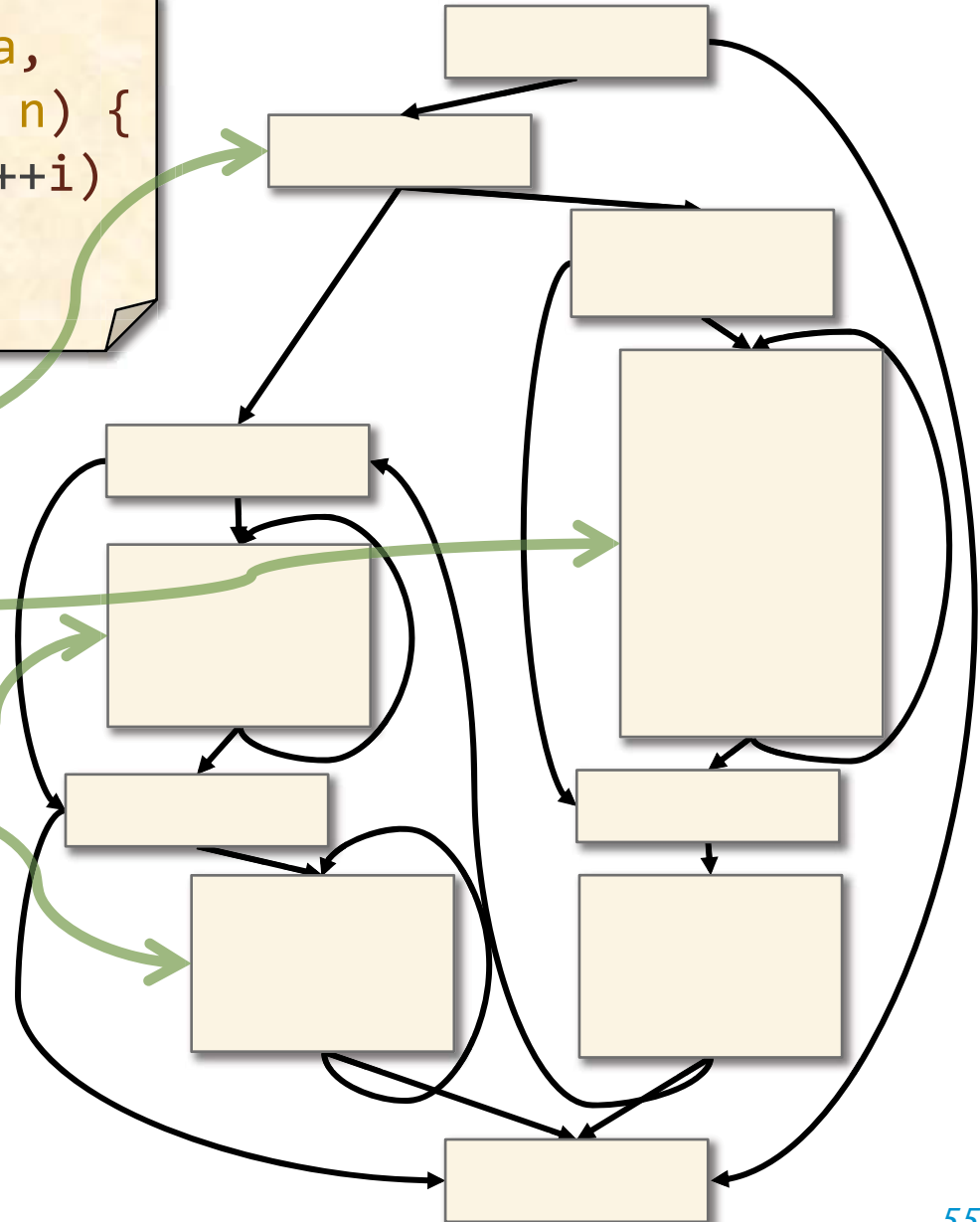
```
void daxpy(double *y, double a,  
          double *x, int64_t n) {  
  for (int64_t i = 0; i < n; ++i)  
    y[i] += a * x[i];  
}
```

Conditional branch

Vectorized loop

Not vectorized loops

Simplified control-flow graph structure

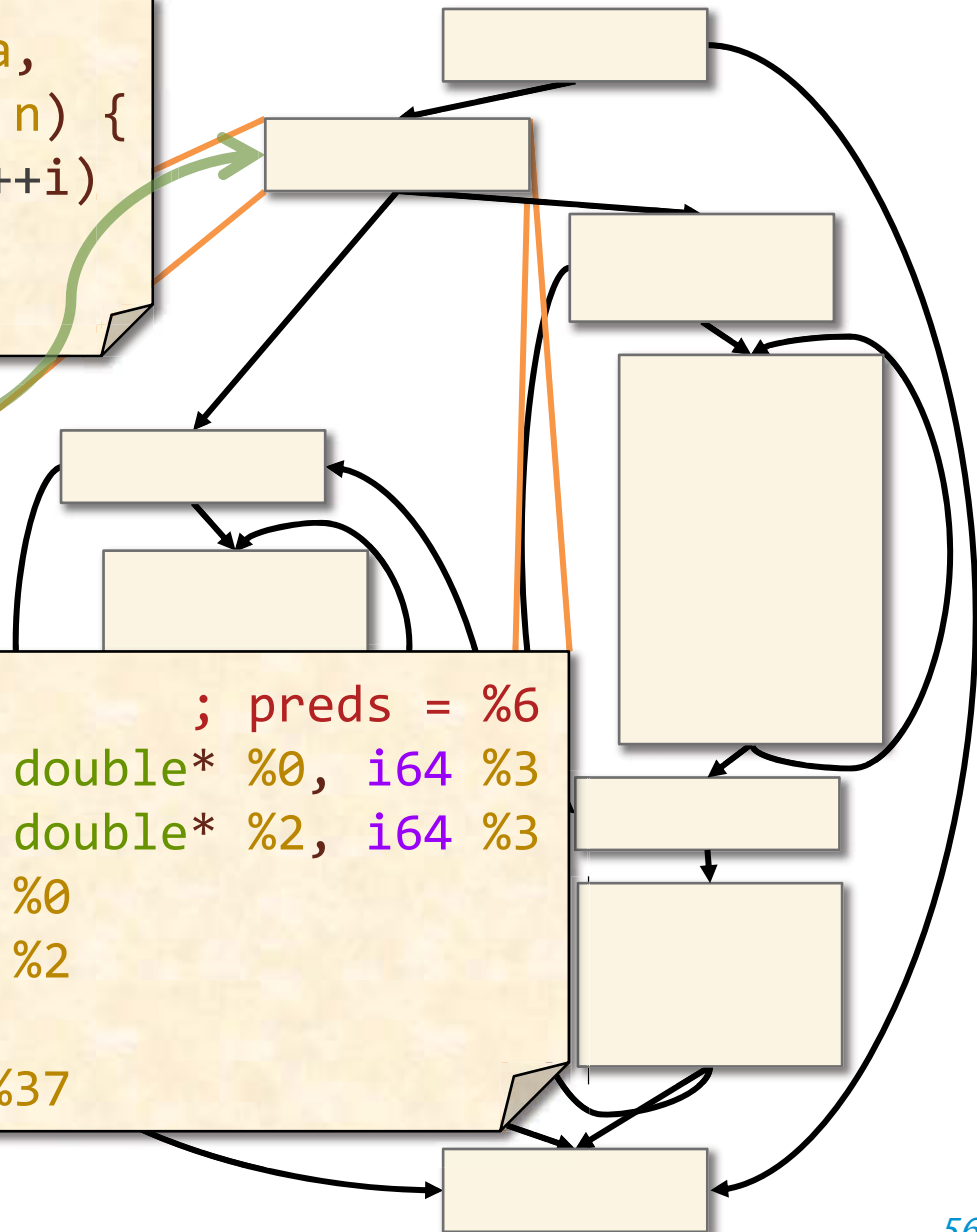


# Choosing Between Loops

```
void daxpy(double *y, double a,  
          double *x, int64_t n) {  
    for (int64_t i = 0; i < n; ++i)  
        y[i] += a * x[i];  
}
```

Conditional branch

```
; <label>:31:                                ; preds = %6  
%32 = getelementptr double, double* %0, i64 %3  
%33 = getelementptr double, double* %2, i64 %3  
%34 = icmp ugt double* %33, %0  
%35 = icmp ugt double* %32, %2  
%36 = and i1 %34, %35  
br i1 %36, label %8, label %37
```





# What's The Condition?

```
void daxpy(double *y, double a,
           double *x, int64_t n) {
    for (int64_t i = 0; i < n; ++i)
        y[i] += a * x[i];
}
```

Computes  $y + n$

Register `%0` stores `y`, register `%2` stores `x`, and register `%3` stores `n`.

```
; <label>:31:                                ; preds = %6
%32 = getelementptr double, double* %0, i64 %3
%33 = getelementptr double, double* %2, i64 %3
%34 = icmp ugt double* %33, %0
%35 = icmp ugt double* %32, %2
%36 = and i1 %34, %35
br i1 %36, label %8, label %37
```

# What's The Condition?

```
void daxpy(double *y, double a,  
          double *x, int64_t n) {  
    for (int64_t i = 0; i < n; ++i)  
        y[i] += a * x[i];  
}
```

Computes  $x + n$

Register `%0` stores `y`, register `%2` stores `x`, and register `%3` stores `n`.

```
; <label>:31:                                ; preds = %6  
%32 = getelementptr double, double* %0, i64 %3  
%33 = getelementptr double, double* %2, i64 %3  
%34 = icmp ugt double* %33, %0  
%35 = icmp ugt double* %32, %2  
%36 = and i1 %34, %35  
br i1 %36, label %8, label %37
```

# What's The Condition?

```
void daxpy(double *y, double a,  
          double *x, int64_t n) {  
    for (int64_t i = 0; i < n; ++i)  
        y[i] += a * x[i];  
}
```

Compares  $x + n > y$

Register %0 stores  $y$ , register %2 stores  $x$ , and register %3 stores  $n$ .

```
; <label>:31:                                ; preds = %6  
%32 = getelementptr double, double* %0, i64 %3  
%33 = getelementptr double, double* %2, i64 %3  
%34 = icmp ugt double* %33, %0  
%35 = icmp ugt double* %32, %2  
%36 = and i1 %34, %35  
br i1 %36, label %8, label %37
```

# What's The Condition?

```
void daxpy(double *y, double a,
           double *x, int64_t n) {
    for (int64_t i = 0; i < n; ++i)
        y[i] += a * x[i];
}
```

Compares  $y + n > x$

Register %0 stores  $y$ , register %2 stores  $x$ , and register %3 stores  $n$ .

```
; <label>:31:                                ; preds = %6
%32 = getelementptr double, double* %0, i64 %3
%33 = getelementptr double, double* %2, i64 %3
%34 = icmp ugt double* %33, %0
%35 = icmp ugt double* %32, %2
%36 = and i1 %34, %35
br i1 %36, label %8, label %37
```

# What's The Condition?

```
void daxpy(double *y, double a,  
           double *x, int64_t n) {  
    for (int64_t i = 0; i < n; ++i)  
        y[i] += a * x[i];  
}
```

Combines the comparisons  
to compute  
 $(y + n > x) \ \& \ (x + n > y)$

Register %0 stores  $y$ , register %2 stores  $x$ , and  
register %3 stores  $n$ .

```
; <label>:31:                                ; preds = %6  
%32 = getelementptr double, double* %0, i64 %3  
%33 = getelementptr double, double* %2, i64 %3  
%34 = icmp ugt double* %33, %0  
%35 = icmp ugt double* %32, %2  
%36 = and i1 %34, %35  
br i1 %36, label %8, label %37
```

**QUESTION:** What does  
the result of this  
condition mean?

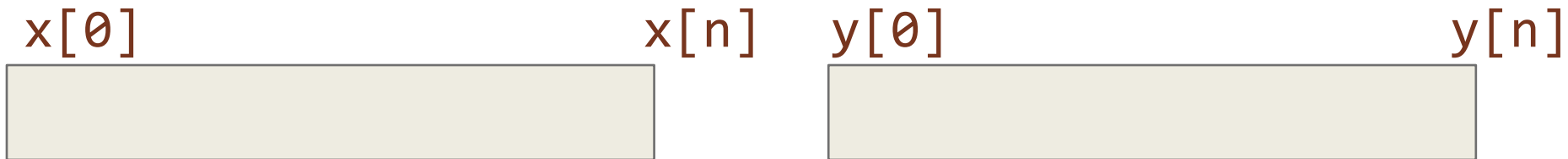
# What's The Condition?

```
void daxpy(double *y, double a,  
           double *x, int64_t n) {  
    for (int64_t i = 0; i < n; ++i)  
        y[i] += a * x[i];  
}
```

Condition:

$(y + n > x) \ \& \ (x + n > y)$

Arrays  $x$  and  $y$  in memory:



The condition is **false** if  $x$  appears before  $y$  in memory or vice versa.

# What's The Condition?

```
void daxpy(double *y, double a,  
          double *x, int64_t n) {  
    for (int64_t i = 0; i < n; ++i)  
        y[i] += a * x[i];  
}
```

Condition:

$(y + n > x) \ \& \ (x + n > y)$

Arrays  $x$  and  $y$  in memory:



The condition is **true** if arrays  $x$  and  $y$  *alias*, meaning that they **overlap** in memory.

# Condition In Context

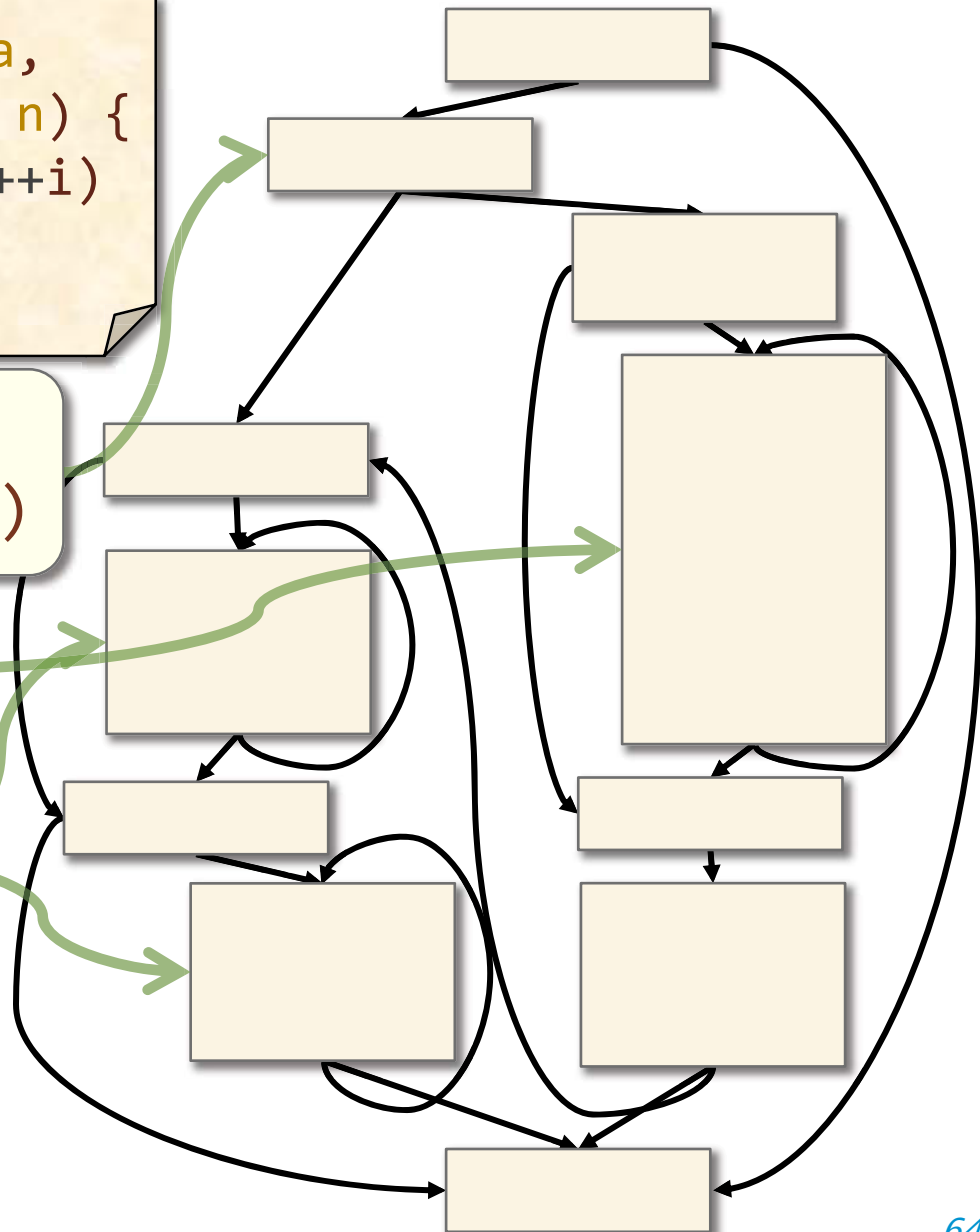
```
void daxpy(double *y, double a,  
          double *x, int64_t n) {  
  for (int64_t i = 0; i < n; ++i)  
    y[i] += a * x[i];  
}
```

Branch based on  
 $(y + n > x) \ \& \ (x + n > y)$

Vectorized loop

Not vectorized  
loops

Simplified control-  
flow graph structure



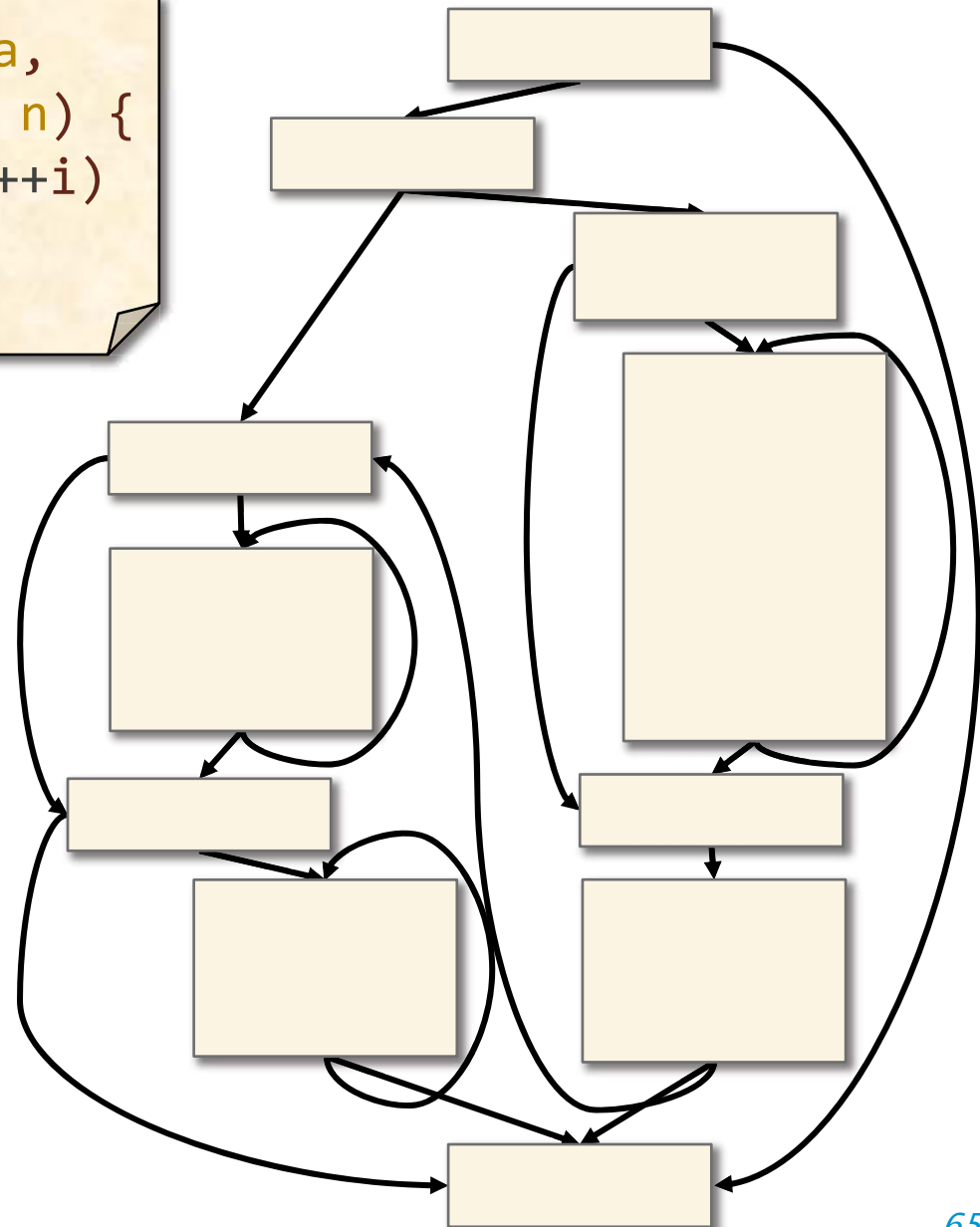


# Multiple Versions

```
void daxpy(double *y, double a,  
          double *x, int64_t n) {  
    for (int64_t i = 0; i < n; ++i)  
        y[i] += a * x[i];  
}
```

**QUESTION:** Does the following loop vectorize?

**ANSWER:** Yes and no.  
The compiler generated **multiple versions** of the loop, due to uncertainty about **memory aliasing**.



# Memory Aliasing

Many compiler optimizations act **conservatively** when **memory aliasing** is possible.

```
void mm_base(double *C, int n_C, double *A, int n_A,
             double *B, int n_B, int n) {
    for (int i = 0; i < n; ++i)
        for (int k = 0; k < n; ++k)
            for (int j = 0; j < n; ++j)
                C[i*n_C+j] += A[i*n_A+k] * B[k*n_B+j];
}
```

```
$ clang -O3 -c mm_base.c -Rpass-missed=.*
mm_base.c:20:23: remark: failed to move load with loop-invariant
address because the loop may invalidate its value [-Rpass-
missed=licm]
```

```
    C[i*n_C+j] += A[i*n_A+k] * B[k*n_B+j];
                   ^
```

# Dealing with Memory Aliasing

Compilers perform **alias analysis** to determine which addresses computed off of different pointers might refer to the same location.

- In general, alias analysis is **undecidable** [HU79, R94].
- Compilers use a variety of tricks to get useful alias-analysis results in practice.
- **EXAMPLE:** Clang uses **metadata** to track alias information derived from various sources, such as type information in the source code.

```
%34 = load double, double* %33, align 8, !tbaa !3,  
!alias.scope !12, !noalias !9
```

# What You Can Do About Aliasing

**SOLUTION:** Annotate your pointers.

- The `restrict` keyword allows the compiler to assume that address calculations based on a pointer will not alias those based on other pointers.
- The `const` keyword indicates that addresses based on a particular pointer will only be read.

```
void daxpy(double *restrict y, double a,
           const double *restrict x, int64_t n) {
    for (int64_t i = 0; i < n; ++i)
        y[i] += a * x[i];
}
```

# DIAGNOSING FAILURES: CASE STUDY 2



# Example Code: Normalize

Sometimes it's not enough to just annotate pointers.

**EXAMPLE:** Normalizing a vector

```
double norm(const double *X, int n);

void normalize(double *restrict Y,
              const double *restrict X,
              int n) {
    for (int i = 0; i < n; ++i)
        Y[i] = X[i] / norm(X, n);
}
```

Compute the norm of the input vector.

Divide each input element by the norm.

**IDEA:** The `norm` call always returns the same value, so the compiler can move it out of the loop.

# Normalize in LLVM IR

C code

```
for (int i = 0; i < n; ++i)
    Y[i] = X[i] / norm(X, n);
```

LLVM IR of loop body  
with -O2 optimization

```
; <label>:8:
%9 = phi i64 [ 0, %5 ], [ %15, %8 ]
%10 = getelementptr inbounds double, double* %1, i64 %9
%11 = load double, double* %10, align 8, !dbg !12
%12 = tail call double @norm(double* %1, i32 %2) #2
%13 = fdiv double %11, %12
%14 = getelementptr inbounds double, double* %1, i64 %9
store double %13, double* %14, align 8, !dbg !12
%15 = add nuw nsw i64 %9, 1
%16 = icmp eq i64 %15, %6
br i1 %16, label %7, label %8
```

LLVM didn't  
move the call  
to norm.

What went  
wrong?

# Fixing Normalize

**PROBLEM:** The compiler does not know that the `norm` function does not modify memory.

**EXAMPLE:** Normalizing a vector

```
__attribute__((const))  
double norm(const double *X, int n);  
  
void normalize(double *restrict Y,  
              const double *restrict X,  
              int n) {  
    for  
    Y  
}
```

Allows LLVM to assume that `norm` does not modify any memory.

**SOLUTION:** Annotate the `norm` function.

For instance, `norm` might modify a global variable.



# DIAGNOSING FAILURES: CASE STUDY 3



# Example: Explicit Unrolling

QUESTION: Does the following loop vectorize?

```
void daxpy4(double *restrict z, double a,
            const double *restrict x,
            const double *restrict y,
            size_t n) {
    for (size_t i = 0; i < n; i+=4) {
        z[i] = a * x[i] + y[i];
        z[i+1] = a * x[i+1] + y[i+1];
        z[i+2] = a * x[i+2] + y[i+2];
        z[i+3] = a * x[i+3] + y[i+3];
    }
}
```

What went wrong?

```
$ clang -O3 -c daxpy.c -Rpass=vector -Rpass-analysis=vector
```

```
daxpy.c:21:3: remark: loop not vectorized: could not determine number of loop iterations [-Rpass-analysis=loop-vectorize]
```

```
for (size_t i = 0; i < n; i+=4) {
```

```
^
```

# Unsigned Overflow

**PROBLEM:** In C, the behavior of unsigned-integer overflow is to wrap to 0.

```
void daxpy4(double *restrict z, double a,  
            const double *restrict x,  
            const double *restrict y,  
            size_t n) {  
    for (size_t i = 0; i < n; i+=4) {  
        z[i] = a * x[i] + y[i];  
        z[i+1] = a * x[i+1] + y[i+1];  
        z[i+2] = a * x[i+2] + y[i+2];  
        z[i+3] = a * x[i+3] + y[i+3];  
    }  
}
```

Implemented as an unsigned 64-bit integer.

**QUESTION:** How many iterations are in this loop?

**ANSWER:** Either  $\lfloor n/4 \rfloor$  or infinity.

# Signed Versus Unsigned

**SOLUTION:** Use **signed integer types**, unless you absolutely need an unsigned type, e.g., for bit-hacking.

```
void daxpy4(double *restrict z, double a,
            const double *restrict x,
            const double *restrict y,
            int64_t n) {
    for (int64_t i = 0; i < n; i+=4) {
        z[i] = a * x[i] + y[i];
        z[i+1] = a * x[i+1] + y[i+1];
        z[i+2] = a * x[i+2] + y[i+2];
        z[i+3] = a * x[i+3] + y[i+3];
    }
}
```

```
$ clang -O3 -c daxpy.c -Rpass=vector
```

```
daxpy.c:21:3: remark: vectorized loop (vectorization width: 2, interleaved count: 1) [-Rpass=loop-vectorize]
```

```
for (int64_t i = 0; i < n; i+=4) {
```

```
^
```

# Signed Overflow

**WHY IT WORKS:** In C, signed-integer overflow has **undefined behavior**.

- As a result, when analyzing code, the compiler is allowed to assume that signed-integer arithmetic **never** overflows.

Why is signed-integer overflow undefined behavior?

- Not all architectures implement signed overflow the same way.
- Programmers generally don't write code that explicitly accommodates signed overflow.
- So the compiler and language **compromise**.

# Summary

Compilers are **powerful** tools for optimizing code.

Compiler optimizations can be **fragile**, because analysis can be **difficult** and the compiler must act **conservatively**.

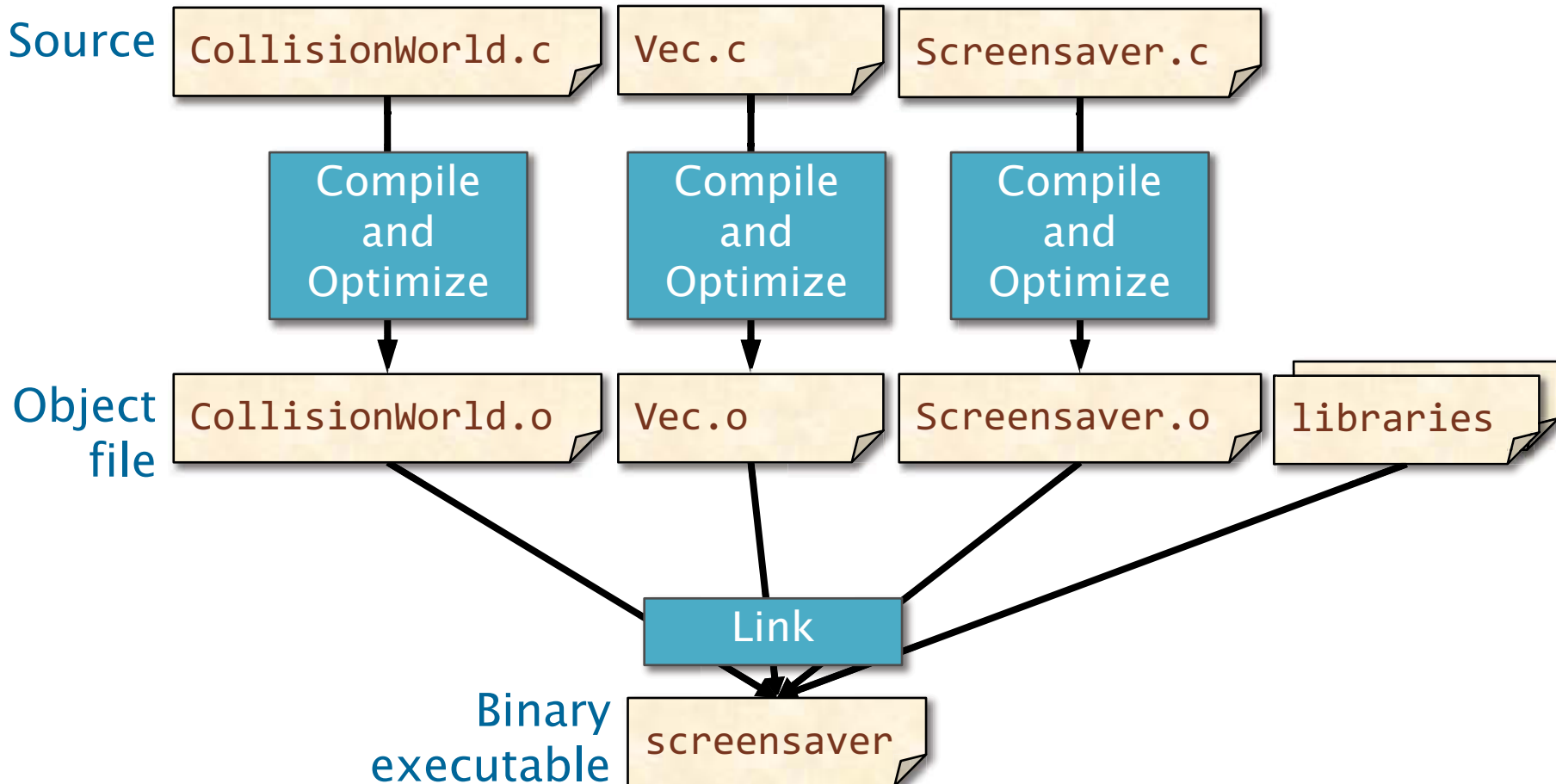
Take a look at the **compiler reports** and the **LLVM IR** to see what the compiler does and figure out how you can help it out.

# BACKUP SLIDES: LINK-TIME OPTIMIZATION



# Ordinary Compilation

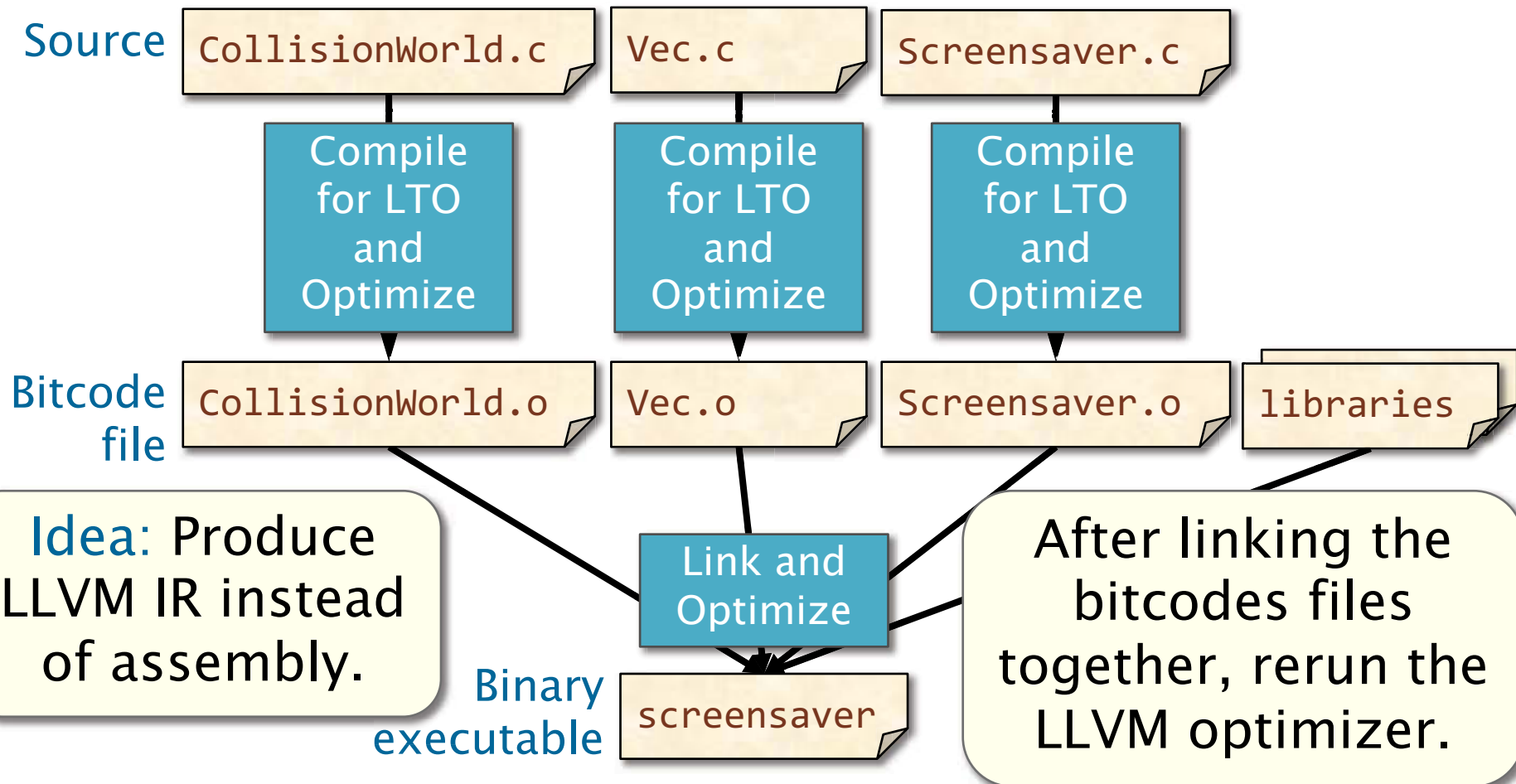
**PROBLEM:** The compiler only transforms code within a single file, or **compilation unit**.





# Optimizing Across Files

Solution: Modern compilers support **link-time optimization (LTO)**.



# LTO in Clang/LLVM

Clang/LLVM supports link-time optimization through the **compiler flag** `-flto`.

- Use the `-flto` flag to compile source code into LLVM bitcode.
- Use the **Gold linker** to link LLVM bitcode files together, via the flags:  
`-flto -fuse-ld=gold`

MIT OpenCourseWare  
<https://ocw.mit.edu>

6.172 Performance Engineering of Software Systems  
Fall 2018

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.