Compiler Optimizations

Fan Long

University of Toronto

UNIVERSITY OF TORONTO

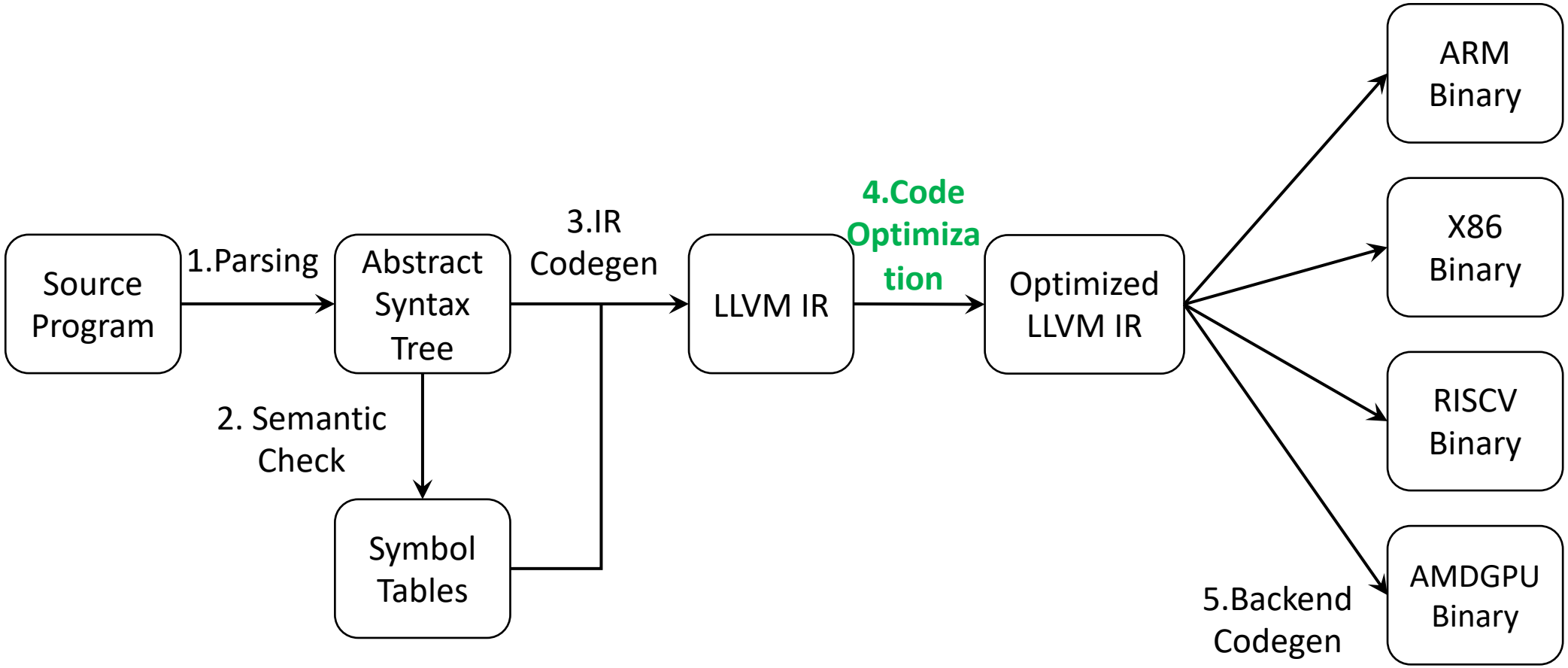
Optimization

- Goals for Optimization
  - Make the object program faster and/or smaller, and/or use less power without changing the meaning of the program!!!
  - To produce object code that is *optimal*, i.e., minimum execution time or minimum space. Usually an undecidable problem.
- Why optimization is needed
  - To improve on inefficient implementation of programming language constructs.
  - To mitigate the mismatch between high level programming language constructs and low-level machine instructions.
  - To compensate for sloppy programming practices.
  - To improve programs in ways that can't be expressed in high level programming languages.
- Optimization **is not** a substitute for good algorithm selection.

When Do Optimizations Happen?

- **Machine Independent Optimizations**: Operate on the IR (i.e., LLVM IR) and happen after the IR code generation.
  - Architecture independent optimizations only
  - Could be reused for different backend targets
- **Machine Dependent Optimizations**: Operate on the generated assembly code (i.e., x86 instructions) and happen after the backend code generation.
  - Can do architecture dependent optimizations
  - Local (peephole) optimizations only because reasoning global properties of assembly code is hard.
- We focus on machine independent optimizations, but many principles are the same.

Typical LLVM Compiler Workflow



Scope of Optimizations

- **peephole** Optimize over a few machine instructions
- **local** Optimize over a few statements.
- **Intraprocedural** Optimize over the body of one routine.
- **Interprocedural** Optimize over some collection of routines, e.g. all the member functions in a Class.
- **global** Optimize over an entire program.

When is Optimization Worthwhile ?

- Most expressions in typical programs are very simple.
- Big optimization gains come from optimizing loops and subscript calculation.
- Generating locally good code to begin with always wins.
- Most interesting optimization problems are NP-complete. There are useful heuristics in many cases.
- Historically, optimization is notorious for introducing bugs into a compiler.

Classical Optimizations

- Machine Independent Optimizations
  - Constant Folding
  - Common Subexpression Elimination
  - Memory to Register Promotion
  - Algebraic Simplification
  - Code Motion (hosting/sink)
  - Strength Reduction and Test Replacement
  - Dead Code Elimination
  - Loop Unrolling, Loop Fusion
  - Procedure/Function Integration

Classical Optimizations

- Machine Dependent Optimizations
  - Register Allocation, allocating variables in registers
  - Use of hardware idioms
  - Peephole optimization
  - Instruction scheduling
  - Cache-sensitive code generation
- Modern compilers will chain multiple optimizations together
  - Constant folding, loop unrolling, and algebraic simplification may create more common subexpressions.
  - Inline functions will create more opportunities for common

Optimization Inhibitors

- *Side Effects* Any construct that changes the value of a variable has a side effect on that variable. An optimizer needs to know when the value of a variable might change in order to optimize the use of variables. Example:

```
int I = 0 ;
int F( void ) {
    return ++I
}
```

```
J = F( ) + I + F(  ) ;
```

- *Exception Handler* An exception handling mechanism can be invoked asynchronously during the execution of a program and can in general cause side effects on variables. The occurrence of exception-caused side effects is extremely difficult to predict.

Optimization Inhibitors

- *Aliasing* An alias exists when two identifiers refer to the same storage location. An optimizer needs to know the effect of every assignment statement. An alias causes an unexpected side effect. Example:

```
int I, J , K , A[100] ;
void P( int &  X , int &  Y ) {
    // X and Y are passed
    // by reference (address)
    A[ X ] = A[ Y ] ;
    X = Y + A[ Y ] ;
    Y = I ;
}
```

```
// Alias on I
P( I, J ) ;

// Alias on A[ 47 ]
P( A[ 47 ] , A[ 47 ] ) ;

// Alias on A[J] iff J == K
P( J , K ) ;
```

How to Write Optimizations for LLVM IR

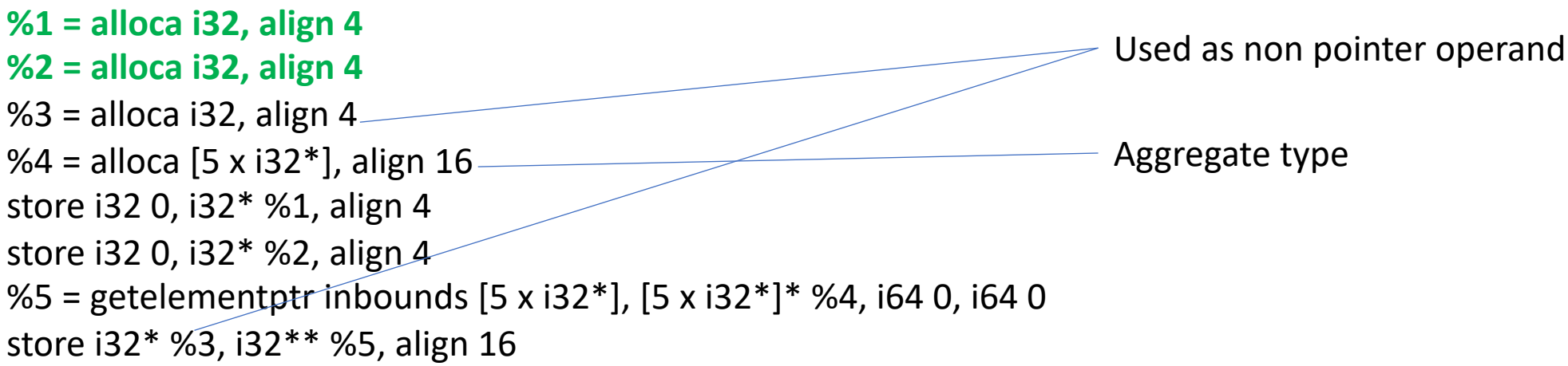
- In LLVM, optimizations are organized as transformation passses.
- Define a class that inherits FunctionPass/ModulePass
- The pass takes the LLVM IR of a module or a function as the input and produces the transformed IR of the module or the function.
- Implement runOnFunction() or runOnModule() manipulate the IR.
- PassManager in LLVM will chain multiple optimization and analysis passes to run together
  - -O3 is actually running a predefined set of optimization passes in LLVM
- **opt** is a LLVM tool that directly invoke a pass on a LLVM IR bytecode.

Register Promotion Optimization

- Memory loads/stores are significantly more expensive than register accesses on all architectures. It is therefore desirable to promote variables holding in memory to registers.
- Challenges:
  - Aliasing and address taken operator (&)
- LLVM pass **mem2reg** performs register promotion for LLVM IR.
- Why it is important? Previous analysis show that this is one of the most impactful optimizations among all -O1 optimization passes.

Safely Promote Local Variable to Registers

- Code translation pass often use **alloca** instructions to hold local variables. How to promote them?
- **Step 1**: Detect all **alloca** instructions that could be promoted. We do not try to promote an **alloca** instruction if:
  - Used at instructions other than load/store pointer operands (address taken or aliasing risk).
  - It is an aggregate type (i.e., do not promote array/struct).



Safely Promote Local Variable to Registers

- **Step 2**: Iterate over all basic blocks of the function. For each one:
  - For every load instruction that fetches a local variable to be promoted, we replace its value with the stored value of the last store instruction in the same basic block.
  - If there is no store instruction proceeding the load in the same block, create a PHI instruction for the variable in the basic block and replace the value of the load instruction with the PHI. (For entry block with no preceeding blocks, it will be an undef value).
  - Use Post[BB, v] to hold the representative value of the variable v at the end of the basic block BB
  - Post [BB, v] equals to the last stored value of v if any.
  - If there is no store instruction for v, Post [BB, v] equals to the PHI instruction created for v in BB

Safely Promote Local Variable to Registers

- **Step 3**: Fill incoming edges of all created PHI instructions based on the computed Post table.
  - If the PHI instruction for variable v has an incoming edge from the preceeding block BB, the incoming edge should equal to Post[BB, v].

- **Step 4:** Remove unused PHI instructions.
- **Step 5**: Remove all **load**/**store** instructions accessing promoted variables. They now should have no other use.
- **Step 6**: Remove all **alloca** instructions for promoted variables, they now should have no other use as well.

Safely Promote Local Variable to Registers

```
int foo(int n) {
  int s, i;
  for (i = 0; i < n; i = i + 1)
    s = s + i;
  return s;
}
```

All alloca can be promoted!

```
define i32 @foo(i32 %0) #0 {
  %n = alloca i32, align 4
  %s = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %0, i32* %n, align 4
  store i32 0, i32* %s, align 4
  store i32 0, i32* %i, align 4
  br label %5
5:                                ; preds = %13, %1
  %6 = load i32, i32* %i, align 4
  %7 = load i32, i32* %n, align 4
  %8 = icmp slt i32 %6, %7
  br i1 %8, label %9, label %16
9:                                ; preds = %5
  %10 = load i32, i32* %s, align 4
  %11 = load i32, i32* %i, align 4
  %12 = add nsw i32 %10, %11
  store i32 %12, i32* %s, align 4
  br label %13
13:                               ; preds = %9
  %14 = load i32, i32* %i, align 4
  %15 = add nsw i32 %14, 1
  store i32 %15, i32* %i, align 4
  br label %5
16:                               ; preds = %5
  %17 = load i32, i32* %s, align 4
  ret i32 %17
}
```

Safely Promote Local Variable to Registers

```
define i32 @foo(i32 %0) #0 {
  %n = alloca i32, align 4
  %s = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %0, i32* %n, align 4
  store i32 0, i32* %s, align 4
  store i32 0, i32* %i, align 4
  br label %5
5:                                                ; preds = %13, %1
  %n_5 = phi i32 [?, %13], [?, %entry]
  %s_5 = phi i32 [?, %13], [?, %entry]
  %i_5 = phi i32 [?, %13], [?, %entry]
  %6 = load i32, i32* %i, align 4
  %7 = load i32, i32* %n, align 4
  %8 = icmp slt i32 %i_5, %n_5
  br i1 %8, label %9, label %16
9:                                                ; preds = %5
  %n_9 = phi i32 [?, %5]
  %s_9 = phi i32 [?, %5]
  %i_9 = phi i32 [?, %5]
  %10 = load i32, i32* %s, align 4
  %11 = load i32, i32* %i, align 4
  %12 = add nsw i32 %s_9, %i_9
  store i32 %12, i32* %s, align 4
  br label %13
13:                                               ; preds = %9
  %n_13 = phi i32 [?, %9]
  %s_13 = phi i32 [?, %9]
  %i_13 = phi i32 [?, %9]
  %14 = load i32, i32* %i, align 4
  %15 = add nsw i32 %i_13, 1
  store i32 %15, i32* %i, align 4
  br label %5
16:                                               ; preds = %5
  %n_16 = phi i32 [?, %13]
  %s_16 = phi i32 [?, %13]
  %i_16 = phi i32 [?, %13]
  %17 = load i32, i32* %s, align 4
  ret i32 %s_16
}
```

Safely Promote Local Variable to Registers

```
define i32 @foo(i32 %0) #0 {
  %n = alloca i32, align 4
  %s = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %0, i32* %n, align 4
  store i32 0, i32* %s, align 4
  store i32 0, i32* %i, align 4
  br label %5
5:                                ; preds = %13, %1
  %n_5 = phi i32 [?, %13], [?, %entry]
  %s_5 = phi i32 [?, %13], [?, %entry]
  %i_5 = phi i32 [?, %13], [?, %entry]
  %6 = load i32, i32* %i, align 4
  %7 = load i32, i32* %n, align 4
  %8 = icmp slt i32 %i_5, %n_5
  br i1 %8, label %9, label %16
9:                                ; preds = %5
  %n_9 = phi i32 [?, %5]
  %s_9 = phi i32 [?, %5]
  %i_9 = phi i32 [?, %5]
  %10 = load i32, i32* %s, align 4
  %11 = load i32, i32* %i, align 4
  %12 = add nsw i32 %s_9, %i_9
  store i32 %12, i32* %s, align 4
  br label %13
13:                               ; preds = %9
  %n_13 = phi i32 [?, %9]
  %s_13 = phi i32 [?, %9]
  %i_13 = phi i32 [?, %9]
  %14 = load i32, i32* %i, align 4
  %15 = add nsw i32 %i_13, 1
  store i32 %15, i32* %i, align 4
  br label %5
16:                               ; preds = %5
  %n_16 = phi i32 [?, %13]
  %s_16 = phi i32 [?, %13]
  %i_16 = phi i32 [?, %13]
  %17 = load i32, i32* %s, align 4
  ret i32 %s_16
}
```

```
Post[%entry, n] = %0
Post[%entry, s] = 0
Post[%entry, i] = 0;
Post[%5, n] = %n_5
Post[%5, s] = %s_5
Post[%5, i] = %i_5
Post[%9, n] = %n_9
Post[%9, s] = %12
Post[%9, i] = %i_9
Post[%13, n] = %n_13
Post[%13, s] = %s_13
Post[%13, i] = %15
```

Safely Promote Local Variable to Registers

```
define i32 @foo(i32 %0) #0 {
  %n = alloca i32, align 4
  %s = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %0, i32* %n, align 4
  store i32 0, i32* %s, align 4
  store i32 0, i32* %i, align 4
  br label %5
5:                                ; preds = %13, %ent
  %n_5 = phi i32 [%n_13, %13], [%0, %entry]
  %s_5 = phi i32 [%s_13, %13], [0, %entry]
  %i_5 = phi i32 [%15, %13], [0, %entry]
  %6 = load i32, i32* %i, align 4
  %7 = load i32, i32* %n, align 4
  %8 = icmp slt i32 %i_5, %n_5
  br i1 %8, label %9, label %16
9:                                ; preds = %5
  %n_9 = phi i32 [%n_5, %5]
  %s_9 = phi i32 [%s_5, %5]
  %i_9 = phi i32 [%i_5, %5]
  %10 = load i32, i32* %s, align 4
  %11 = load i32, i32* %i, align 4
  %12 = add nsw i32 %s_9, %i_9
  store i32 %12, i32* %s, align 4
  br label %13
13:                               ; preds = %9
  %n_13 = phi i32 [%n_9, %9]
  %s_13 = phi i32 [%12, %9]
  %i_13 = phi i32 [%i_9, %9]
  %14 = load i32, i32* %i, align 4
  %15 = add nsw i32 %i_13, 1
  store i32 %15, i32* %i, align 4
  br label %5
16:                               ; preds = %5
  %n_16 = phi i32 [%n_5, %13]
  %s_16 = phi i32 [%s_5, %13]
  %i_16 = phi i32 [%i_5, %13]
  %17 = load i32, i32* %s, align 4
  ret i32 %s_16
}
```

```
Post[%entry, n] = %0
Post[%entry, s] = 0
Post[%entry, i] = 0;
Post[%5, n] = %n_5
Post[%5, s] = %s_5
Post[%5, i] = %i_5
Post[%9, n] = %n_9
Post[%9, s] = %12
Post[%9, i] = %i_9
Post[%13, n] = %n_13
Post[%13, s] = %s_13
Post[%13, i] = %15
```

Safely Promote Local Variable to Registers

```
define i32 @foo(i32 %0) #0 {
  %n = alloca i32, align 4
  %s = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %0, i32* %n, align 4
  store i32 0, i32* %s, align 4
  store i32 0, i32* %i, align 4
  br label %5
5:                                                ; preds = %13, %ent
  %n_5 = phi i32 [%n_13, %13], [%0, %entry]
  %s_5 = phi i32 [%s_13, %13], [0, %entry]
  %i_5 = phi i32 [%15, %13], [0, %entry]
  %6 = load i32, i32* %i, align 4
  %7 = load i32, i32* %n, align 4
  %8 = icmp slt i32 %i_5, %n_5
  br i1 %8, label %9, label %16
9:                                                ; preds = %5
  %n_9 = phi i32 [%n_5, %5]
  %s_9 = phi i32 [%s_5, %5]
  %i_9 = phi i32 [%i_5, %5]
  %10 = load i32, i32* %s, align 4
  %11 = load i32, i32* %i, align 4
  %12 = add nsw i32 %s_9, %i_9
  store i32 %12, i32* %s, align 4
  br label %13
13:                                               ; preds = %9
  %n_13 = phi i32 [%n_9, %9]
  %s_13 = phi i32 [%12, %9]
  %i_13 = phi i32 [%i_9, %9]
  %14 = load i32, i32* %i, align 4
  %15 = add nsw i32 %i_13, 1
  store i32 %15, i32* %i, align 4
  br label %5
16:                                               ; preds = %5
  %n_16 = phi i32 [%n_5, %13]
  %s_16 = phi i32 [%s_5, %13]
  %i_16 = phi i32 [%i_5, %13]
  %17 = load i32, i32* %s, align 4
  ret i32 %s_16
}
```

UNIVERSITY OF TORONTO

Safely Promote Local Variable to Registers

```
define i32 @foo(i32 %0) #0 {
  br label %5
5:                                ; preds = %13, %ent
  %n_5 = phi i32 [%n_13, %13], [%0, %entry]
  %s_5 = phi i32 [%s_13, %13], [0, %entry]
  %i_5 = phi i32 [%15, %13], [0, %entry]
  %8 = icmp slt i32 %i_5, %n_5
  br i1 %8, label %9, label %16
9:                                ; preds = %5
  %n_9 = phi i32 [%n_5, %5]
  %s_9 = phi i32 [%s_5, %5]
  %i_9 = phi i32 [%i_5, %5]
  %12 = add nsw i32 %s_9, %i_9
  br label %13
13:                               ; preds = %9
  %n_13 = phi i32 [%n_9, %9]
  %s_13 = phi i32 [%12, %9]
  %i_13 = phi i32 [%i_9, %9]
  %15 = add nsw i32 %i_13, 1
  br label %5
16:                               ; preds = %5
  %s_16 = phi i32 [%s_5, %13]
  ret i32 %s_16
}
```

Safely Promote Local Variable to Registers

- **Step 7**: Eliminate redundant PHI instructions.
  - If all incoming edges of a PHI instruction pointing to the same value (or self referencing), remove the PHI instruction and replace its references with this value throughout the program.
  - Iteratively perform the above step until no more PHI instruction can be eliminated.

- Note that this last step does not affect the correctness but remove redundant IR for speed.

Safely Promote Local Variable to Registers

```
define i32 @foo(i32 %0) #0 {
  br label %5
5:                                ; preds = %13, %ent
  %n_5 = phi i32 [%n_13, %13], [%0, %entry]
  %s_5 = phi i32 [%s_13, %13], [0, %entry]
  %i_5 = phi i32 [%15, %13], [0, %entry]
  %8 = icmp slt i32 %i_5, %n_5
  br i1 %8, label %9, label %16
9:                                ; preds = %5
  %n_9 = phi i32 [%n_5, %5]
  %s_9 = phi i32 [%s_5, %5]
  %i_9 = phi i32 [%i_5, %5]
  %12 = add nsw i32 %s_9, %i_9
  br label %13
13:                               ; preds = %9
  %n_13 = phi i32 [%n_9, %9]
  %s_13 = phi i32 [%12, %9]
  %i_13 = phi i32 [%i_9, %9]
  %15 = add nsw i32 %i_13, 1
  br label %5
16:                               ; preds = %5
  %s_16 = phi i32 [%s_5, %13]
  ret i32 %s_16
}
```

UNIVERSITY OF TORONTO

Safely Promote Local Variable to Registers

```
define i32 @foo(i32 %0) #0 {
  br label %5
5:                                ; preds = %13, %ent
  %n_5 = phi i32 [%n_13, %13], [%0, %entry]
  %s_5 = phi i32 [%s_13, %13], [0, %entry]
  %i_5 = phi i32 [%15, %13], [0, %entry]
  %8 = icmp slt i32 %i_5, %n_5
  br i1 %8, label %9, label %16
9:                                ; preds = %5
  %n_9 = phi i32 [%n_5, %5]
  %s_9 = phi i32 [%s_5, %5]
  %i_9 = phi i32 [%i_5, %5]
  %12 = add nsw i32 %s_9, %i_9
  br label %13
13:                               ; preds = %9
  %n_13 = phi i32 [%n_9, %9]
  %s_13 = phi i32 [%12, %9]
  %i_13 = phi i32 [%i_9, %9]
  %15 = add nsw i32 %i_13, 1
  br label %5
16:                               ; preds = %5
  ret i32 %s_5
}
```

UNIVERSITY OF TORONTO

Safely Promote Local Variable to Registers

```
define i32 @foo(i32 %0) #0 {
  br label %5
5:                                ; preds = %13, %ent
  %n_5 = phi i32 [%n_9, %13], [%0, %entry]
  %s_5 = phi i32 [%12, %13], [0, %entry]
  %i_5 = phi i32 [%15, %13], [0, %entry]
  %8 = icmp slt i32 %i_5, %n_5
  br i1 %8, label %9, label %16
9:                                ; preds = %5
  %n_9 = phi i32 [%n_5, %5]
  %s_9 = phi i32 [%s_5, %5]
  %i_9 = phi i32 [%i_5, %5]
  %12 = add nsw i32 %s_9, %i_9
  br label %13
13:                               ; preds = %9
  %15 = add nsw i32 %i_9, 1
  br label %5
16:                               ; preds = %5
  ret i32 %s_5
}
```

Safely Promote Local Variable to Registers

```
define i32 @foo(i32 %0) #0 {
  br label %5
5:                                ; preds = %13, %ent
  %n_5 = phi i32 [%n_5, %13], [%0, %entry]
  %s_5 = phi i32 [%12, %13], [0, %entry]
  %i_5 = phi i32 [%15, %13], [0, %entry]
  %8 = icmp slt i32 %i_5, %n_5
  br i1 %8, label %9, label %16
9:                                ; preds = %5
  %12 = add nsw i32 %s_5, %i_5
  br label %13
13:                               ; preds = %9
  %15 = add nsw i32 %i_5, 1
  br label %5
16:                               ; preds = %5
  ret i32 %s_5
}
```

Safely Promote Local Variable to Registers

```
define i32 @foo(i32 %0) #0 {
  br label %5
5:                                ; preds = %13, %ent
  %n_5 = phi i32 [%n_5, %13], [%0, %entry]
  %s_5 = phi i32 [%12, %13], [0, %entry]
  %i_5 = phi i32 [%15, %13], [0, %entry]
  %8 = icmp slt i32 %i_5, %n_5
  br i1 %8, label %9, label %16
9:                                ; preds = %5
  %12 = add nsw i32 %s_5, %i_5
  br label %13
13:                               ; preds = %9
  %15 = add nsw i32 %i_5, 1
  br label %5
16:                               ; preds = %5
  ret i32 %s_5
}
```

Safely Promote Local Variable to Registers

```
define i32 @foo(i32 %0) #0 {
  br label %5
5:                                ; preds = %13, %ent
  %s_5 = phi i32 [%12, %13], [0, %entry]
  %i_5 = phi i32 [%15, %13], [0, %entry]
  %8 = icmp slt i32 %i_5, %0
  br i1 %8, label %9, label %16
9:                                ; preds = %5
  %12 = add nsw i32 %s_5, %i_5
  br label %13
13:                               ; preds = %9
  %15 = add nsw i32 %i_5, 1
  br label %5
16:                               ; preds = %5
  ret i32 %s_5
}
```

Safely Promote Local Variable to Registers

```
define i32 @foo(i32 %0) #0 {
  br label %5
5:                                ; preds = %13, %ent
  %s_5 = phi i32 [%12, %13], [0, %entry]
  %i_5 = phi i32 [%15, %13], [0, %entry]
  %8 = icmp slt i32 %i_5, %0
  br i1 %8, label %9, label %16
9:                                ; preds = %5
  %12 = add nsw i32 %s_5, %i_5
  br label %13
13:                               ; preds = %9
  %15 = add nsw i32 %i_5, 1
  br label %5
16:                               ; preds = %5
  ret i32 %s_5
}
```

No more redundant PHI instructions!

Notation for Optimization Examples

- We use the operator @ as a C-style pointer dereferencing operator in these examples to avoid confusion with $*$ which is always multiply.
- We use a C-style assignment operator = that expects an l-value as its left operand and an r-value as its right operand
- &X is used for *address of* X
- Variables with names ending in P (e.g., AP, BP) are *pointer variables*.
- Most data items are assumed to be 4 bytes wide.

For most optimizations we will preferentially force address items into the form @ P + constant because this form fits well with the most common register + displacement addressing mode on many machines.

Constant Folding

- If an expression involves only constants or other values known to the compiler, calculate the value of the expression at compile time.
- Challenges:
  - Compiler must contain a calculator for expressions. For cross compiler must calculate in the arithmetic of the target machine.
  - May want to extend to common builtin functions, e.g. max, sqrt.
  - Beware of arithmetic faults during constant expression evaluation, e.g. overflow, underflow, divide by zero.
  - May need to reorder expressions to facilitate constant folding, e.g. 3 + A + 4

```
#define aSize 100
int A[ aSize ] ;
A[aSize-1] = aSize * aSize ;   @ (&A[0] + 4* (100-1)) = 100 * 100
                               @ (&A[0] + 396) = 10000
```

Common Subexpression Elimination

- If the same expression is calculated more than once, save the value of the first calculation and use it place of the repeated calculations.
- Challenges:
  - Detection of common subexpressions. Use heuristics, canonical ordering.
  - Must be able to detect when the values of the variables in an expression could be changed. The presence of functions with side effects, aliasing and exception handling make this more difficult.

A[J] = A[J] + 1 ;	@ (&A[0] + 4 * J) = @ (&A[0] + 4 * J) + 1
A[J + 1] = A[J] ;	@ (&A[0] + 4 * J + 4) = @ (&A[0] + 4 * J)
	AP = (&A[0] + 4 * J)
	@ AP = @ AP + 1
	@ (AP+4) = @ AP

Variable Folding

- If a variable is assigned a ”computationally simple” value, replace subsequent uses of the variable by the value. Creates new opportunities for constant folding and common subexpression elimination.
- Challenges: aliasing, side effects

```
J = 13 ;
A[ J ] = B[ J ] * C[ J ] ;        A[ 13 ] = B[ 13 ] * C[ 13 ] ;

K = J + 1 ;
A[ K ] = B[ K ] ;                 A[ J + 1 ] = B[ J + 1 ] ;
```

Algebraic Simplification

- Use algebraic identities to simplify or reorder expressions. Use commutativity and associativity of operators. Often used to facilitate constant folding and common subexpression elimination.
- Challenges:
  - Must be careful not to change the meaning of the program
  - Consider the corner cases of overflow and underflow

$X + 0 \Rightarrow X$	$X - 0 \Rightarrow X$	$Y * 1 \Rightarrow Y$
$Y / 1 \Rightarrow Y$	$X * 0 \Rightarrow 0$	$-X + -Y \Rightarrow -(X + Y)$
P **and** true $\Rightarrow$ P	P **and** false $\Rightarrow$ false	
P **or** true $\Rightarrow$ true	P **or** false $\Rightarrow$ P	

Example of Optimizations in LLVM IR

```
%0 = load i32, i32* @x, align 4
%1 = load i32, i32* @y, align 4
%2 = mul i32 %0, %1
%3 = load i32, i32* @y, align 4
%4 = load i32, i32* @x, align 4
%5 = mul i32 %3, %4
%6 = add i32 %2, %5
%7 = load i32, i32* @y, align 4
%8 = load i32, i32* @x, align 4
%9 = mul i32 %7, %8
%10 = load i32, i32* @y, align 4
%11 = mul i32 %9, %10
%12 = add i32 %6, %11
ret i32 %12
```

```
%0 = load i32, i32* @x, align 4
%1 = load i32, i32* @y, align 4
%2 = mul i32 %1, %0
%3 = add i32 %1, 2
%4 = mul i32 %2, %3
ret i32 %4
```

opt -O3 output.bc > output1.bc

After: Constant Folding + CSE + Algebraic Simplifications

```
a = (x * y) + (y * x) + (y * x * y);
return a;
```

UNIVERSITY OF TORONTO

Code Motion

- Move code (statements, expressions or fragments thereof) out of loops (forward or backward) to places where they are executed less frequently.
- Code that is moved must be *loop invariant*, i.e. independent of the loop indices.
- Challenges:
  - Correctness (safety) of code motion.
  - Influenced by side effects, aliasing and exception handling.
  - Must preserve semantics of loops that never execute.
  - may need fixup code after the loop.
- Some optimizers do code motion by copy insertion followed by common subexpression elimination.

Code Motion Example

```
int J, K ;
float A[100][100], B[100] ;
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        A[J][K] = B[J] ;
```

```
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        @ (&A[0][0] + 400 * J + 4 * K)
            = @ (&B[0] + 4 * J)
```

```
for( J = 0 ; J < 100 ; J++ ) {
    AP = &A[0][0] + 400 * J
    BV = @ (&B[0] + 4 * J)
    for( K = 0 ; K < 100 ; K++ )
        @ (AP + 4 * K) = BV
}
```

Strength Reduction

- Strength reduction is the replacement of ”slow” operations (e.g., multiply and divide) with ”faster” operations (e.g., add and subtract)
- Challenges:
  - Deciding which operations are sufficiently ”slow” as to warrant this optimization. Strength reduction was more important when multiply and divide were *slow* relative to add and subtract.
  - Need to be careful not to change the meaning of the program.
- Strength reduction is often used to eliminate multiplications in array subscript polynomials. Often a precursor to machine dependent optimizations.

Strength Reduction Example

```
int J, K ;
float A[100][100], B[100] ;
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        A[J][K] = B[J] ;
```

```
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        @ (&A[0][0] + 400 * J + 4 * K)
            = @ (&B[0] + 4 * J)
```

```
for( J4 = 0 , J400 = 0 , J = 0 ; J < 100 ;
                    J++ , J4 += 4, J400 += 400 ) {
    AP = &A[0][0] + J400
    BV = @ (&B[0] + J4 )
    for( K = 0 , K4 = 4 ; K < 100
                    ; K++, K4 += 4 )
        @ (AP + K4 ) = BV
}
```

Dead Code Elimination

- Compiler detects and eliminates code that can never be executed (dead code) and computations of values that are never subsequently used (useless computations).
- Challenges:
  - Detection of dead code requires analysis of control flow graph.
  - Useless calculations may result from programmer error or from other optimizations. Test replacement is a form of useless calculation elimination.

```
if( true )
    X = Y ;          |  X = Y ;
else
    X = Z ;
return ;             |  return ;
X = Y ;
```

Loop Unrolling

- Replicate the body of a loop and adjust loop index. Reduces loop overhead relative to loop body. Enables common subexpression and constant folding optimizations.
- Challenges:
  - detecting unrollable loops and considering side effects
  - loop carried dependencies
  - handling loop remainder

```
for( J = 0 ; J < 200 ; J++ )
    A[ J ] = B[ J ] * C[ J ] ;
```

```
for( J = 0 ; J < 200 ; J + = 4 ) {
    A[ J ] = B[ J ] * C[ J ] ;
    A[ J + 1 ] = B[ J + 1 ] * C[ J + 1 ] ;
    A[ J + 2 ] = B[ J + 2 ] * C[ J + 2 ] ;
    A[ J + 3 ] = B[ J + 3 ] * C[ J + 3 ] ;
}
```

Optimizations After Loop Unrolling

```
A[ J ] = B[ J ] * C[ J ] ;
A[ J + 1 ] = B[ J + 1 ] * C[ J + 1 ] ;
A[ J + 2 ] = B[ J + 2 ] * C[ J + 2 ] ;
A[ J + 3 ] = B[ J + 3 ] * C[ J + 3 ] ;
```

```
@ (&A[0]+4*J) =@ (&B[0]+4*J) + @ (&C[0] + 4*J)
@ (&A[0]+4*J+4) =@ (&B[0]+4*J+4) + @ (&C[0] + 4*J+4)
@ (&A[0]+4*J+8) =@ (&B[0]+4*J+8) + @ (&C[0] + 4*J+8)
@ (&A[0]+4*J+12) =@ (&B[0]+4*J+12) + @ (&C[0] + 4*J+12)
```

```
J4 = 4 * J
AJ4P = &A[0] + J4
BJ4P = &B[0] + J4
CJ4P = &C[0] + J4
@ AJ4P = @ BJ4P + @ CJ4P
@ (AJ4P + 4) = @ (BJ4P + 4) + @ (CJ4P + 4)
@ (AJ8P + 8) = @ (BJ8P + 8) + @ (CJ8P + 8)
@ (AJ12P + 12) = @ (BJ12P + 12) + @ (CJ12P + 12)
```

Loop Fusion (Jamming)

- Combine and simplify two or more loops that share a common index or range. Leads to common subexpression optimizations.
- Challenges:
  - detection of fusible loops
  - side effects and aliasing

```
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        A[ J ][ K ] = 0 ;
for( J = 0 ; J < 100 ; J++ )
    A[ J ][ J ] = 1 ;
```

```
for( J = 0 ; J < 100 ; J++ ) {
    for( K = 0 ; K < 100 ; K++ )
        A[ J ][ K ] = 0 ;
    A[ J ][ J ] = 1 ;
}
```

Routine Inlining

- Replace the call of a procedure or function with an inline expansion of the routine body with actual parameters substituted for the formal parameters.
- Eliminates routine call and return overhead.
- Challenges:
  - Aliasing and side effects. Recursive procedures.
  - Extreme care must be taken to preserve the semantics of parameter passing.
  - Only possible for leaf routines
  - Must rename variables/virtual registers to avoid name collision
- This is an important optimization for Object Oriented languages where Objects export a lot of very small Object access routines (e.g. set and get functions in Java).

Routine Inlining Example

```
void swap( int * A , int * B ) {
        int T = *A ;
        *A = *B ;
                *B = T ;
                return ;
}
...
int T , U ;
swap( &T , &U ) ;
```

```
{ int __T0023 = @ ( &T ) ;
        @ ( &T ) = @ ( & U ) ;
        @ ( &U ) = __T0023 ;
}
```

Tail Recursion Elimination

- If the body of a recursive routine ends with a recursive call to the routine, replace the call with a setting of parameters followed by a branch to the start of the routine body.
- Saves routine call and return overhead.
  Saves activation record allocation/deallocation overhead *and a huge amount of stack space*.
- Challenges:
  - Needs extreme care to preserve parameter passing semantics.
  - Often done where parameters are passed by value.
- This is a *really* important optimization in functional languages, e.g. Lisp, Scheme, ML

Tail Recusion Elimination

```
int Func( int A , int B ) {

    ...

    return Func( A - 1 , B + 1 );
}
```

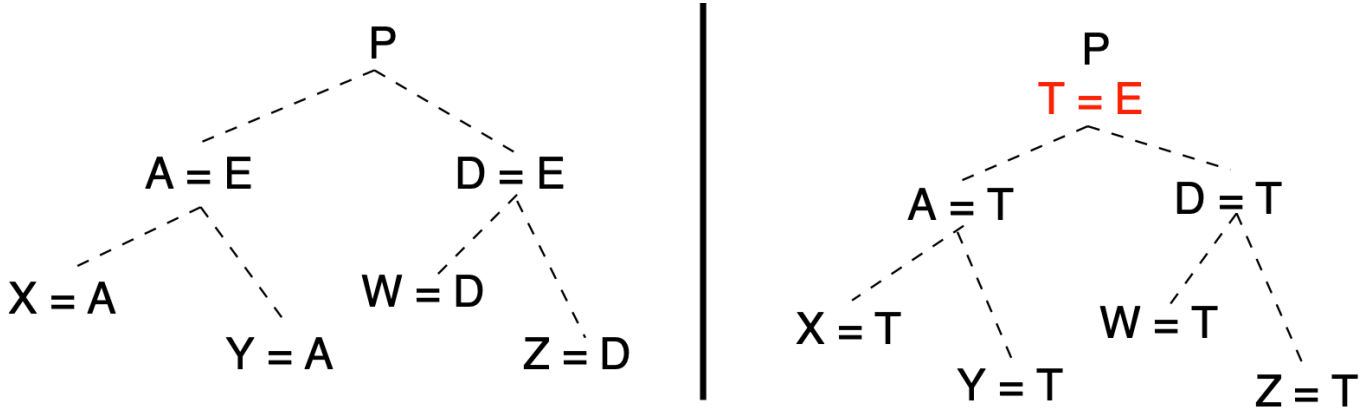
```
int Func( int A , int B ) {
    top:

    ...

    A = A - 1 ;
    B = B + 1 ;
    goto top ;
}
```

Code Hoisting

- If some expression is calculated on all paths leading *from* some point P in the programs control flow graph *and* the expression satisfies the conditions for common subexpression elimination then code for the expression can be moved (hoisted) to point P.
- Challenges:
  - Safety, aliasing, side effects.
  - Computationally expensive to detect hoisting opportunities.
  - Mostly useful for moving out of loops



Machine Depependent Optimizations

- Recognize specific programming language constructs that can be more efficiently implemented using specialized machine instructions
- Challenges:
  - Must maintain programming language semantics.
  - Be careful about interactions with other instructions, i.e., setting of condition codes.

I = I + 1 ;	INC I	PDP-11, VAX
J = 0 ;	CLR J	PDP-11
A[*] = B[*] ;	MVCL A,B	System/370
for(I = 1 ; I <= 100 ; I++)	BXLE	System/370
$X * 2^n$	SLA rX,n	System/370
C = 'A' ;	MVI 'A',C	System/370

Reading Assignments

- How LLVM Optimizes a Function: https://blog.regehr.org/archives/1603

Q/A?