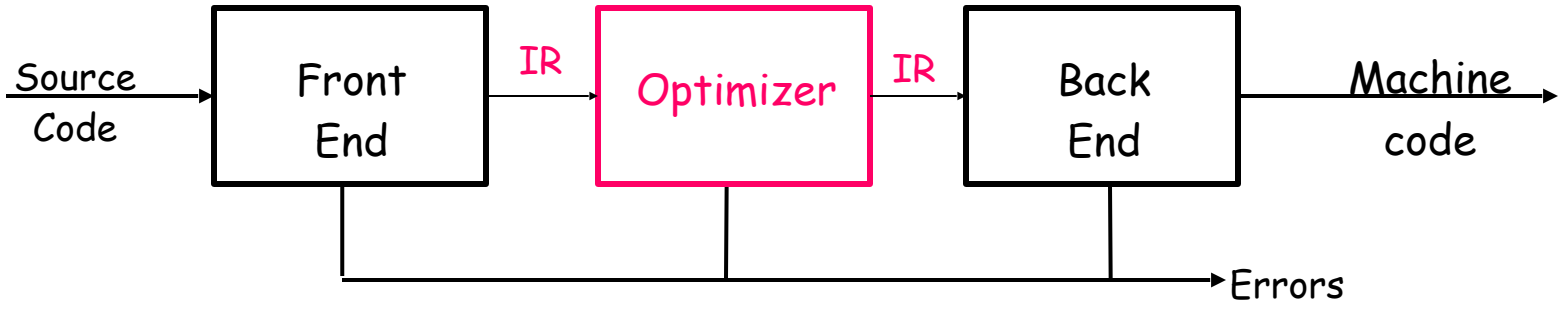
Introduction to Code Optimization

Traditional Three-Phase Compiler



Optimization (or Code Improvement)

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …

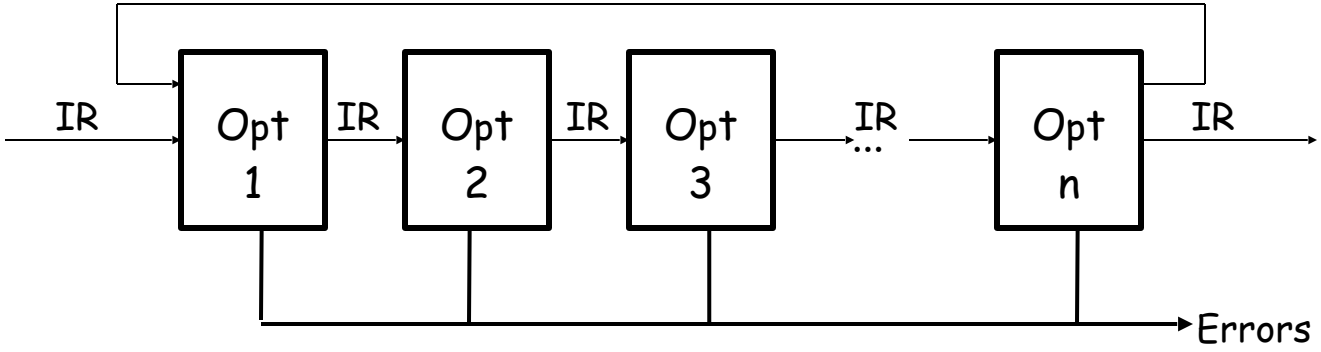
Transformations have to be:

- Safely applied and (it does not change the result of the running program)
- Applied when profit has expected

Background

- Until the early 1980s optimisation was a feature should be added to the compiler only after its other parts were working well
- Debugging compilers vs. optimising compilers
- After the development of RISC processors the demand for support from the compiler had increased

The Optimizer



Modern optimizers are structured as a series of passes

Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code

The Role of the Optimizer

- The compiler can implement a procedure in many ways
- The optimizer tries to find an implementation that is "better"
  - Speed, code size, data space, …

To accomplish this, it

- Analyzes the code to derive knowledge about run-time behavior
  - Data-flow analysis, pointer disambiguation, …
  - General term is "static analysis"
- Uses that knowledge in an attempt to improve the code
  - Literally hundreds of transformations have been proposed
  - Large amount of overlap between them

Nothing "optimal" about optimization

- Proofs of optimality assume restrictive & unrealistic conditions

Scope of Optimization

In scanning and parsing, "scope" refers to a region of the code that corresponds to a distinct name space.

In optimization "scope" refers to a region of the code that is subject to analysis and transformation.

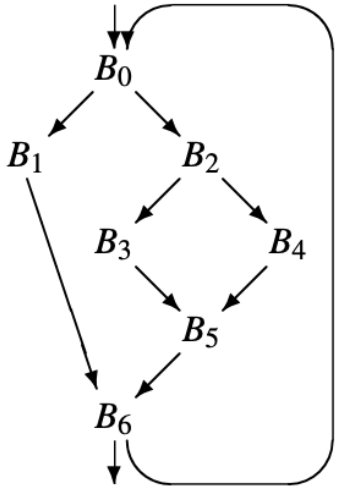
- Notions are somewhat related
- Connection is not necessarily intuitive

Different scopes introduces different challenges & different opportunities

Historically, optimization has been performed at several distinct scopes.

Scope of Optimization

CFG of basic blocks: BB is a maximal length sequence of straightline code.



Local optimization

- Operates entirely within a single basic block
- Properties of block lead to strong optimizations

Regional optimization

- Operate on a region in the CFG that contains multiple blocks
- Loops, trees, paths, extended basic blocks

new opportunities

Whole procedure optimization (intraprocedural)

- Operate on entire CFG for a procedure

Whole program optimization (interprocedural)

- Operate on some or all of the call graph (multiple procedures)
- Must contend with call/return & parameter binding

A Regional Technique

Loop Unrolling

Applications spend a lot of time in loops

- We can reduce loop overhead by unrolling the loop

```
do i = 1 to 100 by 1
    a(i) ← b(i) * c(i)
    end
```

Complete unrolling

```
a(1)   ← b(1) * c(1)
a(2)   ← b(2) * c(2)
a(3)   ← b(3) * c(3)
  ...
a(100) ← b(100) * c(100)
```

- Eliminated additions, tests and branches: reduce the number of operations. The resulting code can be subjected to strong local optimization!
- Only works with fixed loop bounds & few iterations
- The principle, however, is sound
- Unrolling is always safe, as long as we get the bounds right

Loop Unrolling

Unrolling by smaller factors can achieve much of the benefit

Example: unroll by 4 (8, 16, 32? depends on # of registers)

```
do i = 1 to 100 by 1
      a(i) ← b(i) * c(i)
   end
```

Unroll by 4

```
do i = 1 to 100 by 4
   a(i)   ← b(i) * c(i)
   a(i+1) ← b(i+1) * c(i+1)
   a(i+2) ← b(i+2) * c(i+2)
   a(i+3) ← b(i+3) * c(i+3)
   end
```

Achieves much of the savings with lower code growth

- Reduces tests & branches by 25%
- Less overhead per useful operation

But, it relied on knowledge of the loop bounds…

Loop Unrolling

Unrolling with unknown bounds

Need to generate guard loops

```
do i = 1 to n by 1
    a(i) ← b(i) * c(i)
    end
```

Unroll by 4

```
i ← 1
do while (i+3 < n )
    a(i)   ← b(i) * c(i)
    a(i+1) ← b(i+1) * c(i+1)
    a(i+2) ← b(i+2) * c(i+2)
    a(i+3) ← b(i+3) * c(i+3)
    i ←i + 4
    end
do while (i < n)
    a(i)   ← b(i) * c(i)
    i ← i + 1
    end
```

Achieves most of the savings

- Reduces tests & branches by 25%
- Guard loop takes some space

Can generalize to arbitrary upper & lower bounds, unroll factors

Loop Unrolling

i=1,…100 : a(i)=a(i)+b(i)+b(i-1)

One other unrolling trick

Eliminate copies at the end of a loop

```
t1 ← b(0)
do i = 1 to 100 by 1
        t2 ← b(i)
      a(i) ← a(i) + t1 + t2
   t1 ← t2
   end
```

Unroll and rename

```
t1 ← b(0)
do i = 1 to 100 by 2
        t2   ← b(i)
      a(i)   ← a(i) + t1 + t2
       t1    ← b(i+1)
    a(i+1) ← a(i+1) + t2 + t1
    end
```

Unroll

- Eliminates the copies, which were a naming artifact
- Achieves some of the benefits of unrolling
  - Lower overhead, longer blocks for local optimization
- Situation occurs in more cases than you might suspect

Sources of Degradation

- It increases the size of the code
- The unrolled loop may have more demand for registers
- If the demand for registers forces additional register spills (store and reloads) then the resulting memory traffic may overwhelm the potential benefits of unrolling