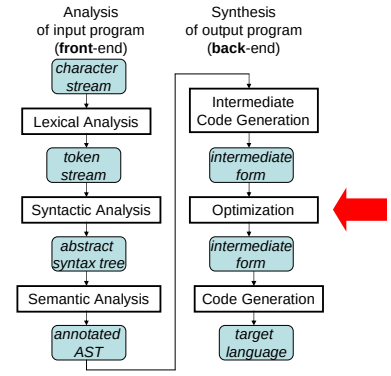
Optimization

Before and after generating machine code, devote one or more passes over the program to "improve" code quality

Compiler Passes



Optimizations

Identify inefficiencies in intermediate or target code

Replace with equivalent but better sequences

- equivalent = "has the same externally visible behavior"

Target-independent optimizations best done on IL code

Target-dependent optimizations best done on target 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, …

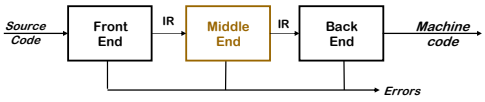
To accomplish this, it

- 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
- Better goal is to "usually improve"

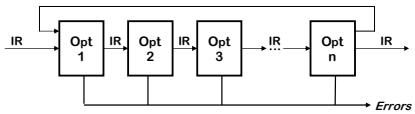
Traditional Three-pass Compiler



Middle End does Code Improvement (Optimization)

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …
- Must preserve "meaning" of the code
  - Measured by values of named variables
  - A course (or two) unto itself

The Optimizer (or Middle End)



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
- Encode an idiom in some particularly efficient form

Kinds of optimizations

Optimizations are characterized by which Transformation over what Scope. Typical scopes are:

peephole:
- look at adjacent instructions

• **local**:
- look at straight-line sequence of statements

• **global** (intraprocedural):
- look at entire procedure

• **whole program** (interprocedural):
- look across procedures

Larger scope => better optimization but more cost and complexity

7

Peephole Optimization

After target code generation, look at adjacent instructions (a "peephole" on the code stream)
- try to replace adjacent instructions with something faster

Example:

```
movl %eax, 12(%ebp)
movl 12(%ebp), %ebx
=>
movl %eax, 12(%ebp)
movl %eax, %ebx
```

8

Algebraic Simplification

"constant folding", "strength reduction"

```
z = 3 + 4;

z = x + 0;
z = x * 1;

z = x * 2;
z = x * 8;

z = x / 8;

double x, y, z;
z = (x + y) - y;
```

Can be done by peephole optimizer, or by code generator

9

Local Optimizations

Analysis and optimizations within a basic block

- Basic block: straight-line sequence of statements
  - no control flow into or out of middle of sequence
- Better than peephole
- Not too hard to implement

Machine-independent, if done on intermediate code

10

Local Constant Propagation

If variable assigned a constant value, replace downstream uses of the variable with the constant.

Can enable more constant folding

Example:

```
final int count = 10;
...
x = count * 5;
y = x ^ 3;
```

Unoptimized intermediate code:

```
t1 = 10;
t2 = 5;
t3 = t1 * t2;
x = t3;
t4 = x;
t5 = 3;
t6 = exp(t4, t5);
y = t6;
```

11

Local Dead Assignment (Store) Elimination

If l.h.s. of assignment never referenced again before being overwritten, then can delete assignment.

Example:

```
final int count = 10;
...
x = count * 5;
y = x ^ 3;
x = 7;
```

Primary use: clean-up after previous optimizations!

Intermediate code after constant propagation:

```
t1 = 10;
t2 = 5;
t3 = 50;
x = 50;
t4 = 50;
t5 = 3;
t6 = 125000;
y = 125000;
x = 7;
```

12

Local Common Subexpression Elimination (AKA Redundancy Elimination)

Avoid repeating the same calculation
- CSE of repeated loads: redundant load elimination

Keep track of available expressions

Source:
```
... a[i] + b[i] ...
```
Unoptimized intermediate code:
```
t1 = *(fp + ioffset);
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);
t5 = *(fp + ioffset);
t6 = t5 * 4;
t7 = fp + t6;
t8 = *(t7 + boffset);
t9 = t4 + t8;
```


Redundancy Elimination Implementation

An expression x+y is redundant if and only if, along every path from the procedure's entry, it has been evaluated, and its constituent subexpressions (x & y) have not been re-defined.

If the compiler can prove that an expression is redundant
- It can preserve the results of earlier evaluations
- It can replace the current evaluation with a reference

Two pieces to the problem
- Proving that x+y is redundant
- Rewriting the code to eliminate the redundant evaluation

One technique for accomplishing both is called value numbering

Value Numbering (An old idea)

The key notion (Balke 1968 or Ershov 1954)
- Assign an identifying number, V(n), to each expression
  - V(x+y) = V(j) iff x+y and j have the same value ∀ path
  - Use hashing over the value numbers to make it efficient
- Use these numbers to improve the code

Improving the code
- Replace redundant expressions
- Simplify algebraic identities
- Discover constant-valued expressions, fold & propagate them

This technique was invented for low-level, linear IRs
Equivalent methods exist for trees (*build a* ***DAG***)

Local Value Numbering

The Algorithm

For each operation *o* = <operator, o_1, o_2> in the block
1. Get value numbers for operands from hash lookup
2. Hash <operator,VN(o_1),VN(o_2)> to get a value number for *o*
3. If *o* already had a value number, replace *o* with a reference
4. If o_1 & o_2 are constant, evaluate it & replace with a load

If hashing behaves, the algorithm runs in linear time

Handling algebraic identities
- Case statement on operator type
- Handle special cases within each operator

Local Value Numbering

Example

Original Code	With VNs	Rewritten
a ← x + y	$a^3 \leftarrow x^1 + y^2$	$a^3 \leftarrow x^1 + y^2$
* b ← x + y	* $b^3 \leftarrow x^1 + y^2$	* $b^3 \leftarrow a^3$
a ← 17	$a^4 \leftarrow 17$	$a^4 \leftarrow 17$
* c ← x + y	* $c^3 \leftarrow x^1 + y^2$	* $c^3 \leftarrow a^3$ (oops!)

Two redundancies:
- Eliminate stmts with a *
- Coalesce results ?

Options:
- Use $c^3 \leftarrow b^3$
- Save a^3 in t^3
- Rename around it

Local Value Numbering

Example (continued)

Original Code	With VNs	Rewritten
$a_0 \leftarrow x_0 + y_0$	$a_0^3 \leftarrow x_0^1 + y_0^2$	$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0 \leftarrow x_0 + y_0$	* $b_0^3 \leftarrow x_0^1 + y_0^2$	* $b_0^3 \leftarrow a_0^3$
$a_1 \leftarrow 17$	$a_1^4 \leftarrow 17$	$a_1^4 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$	* $c_0^3 \leftarrow x_0^1 + y_0^2$	* $c_0^3 \leftarrow a_0^3$

Renaming:
- Give each value a unique name
- Makes it clear

Notation:
- While complex, the meaning is clear

Result:
- a_0^3 is available
- Rewriting just works

Simple Extensions to Value Numbering

Constant folding

- Add a bit that records when a value is constant
- Evaluate constant values at compile-time
- Replace with load immediate or immediate operand
- No stronger local algorithm

Identities:
x←y, x+0, x-0, x*1, x÷1, x-x, x*0, x÷x, x∨0, x ∧ 0xFF…FF, max(x,MAXINT), min(x,MININT), max(x,x), min(y,y), and so on …

Algebraic identities

- Must check (many) special cases
- Replace result with input VN
- Build a decision tree on operation

With values, not names

Intraprocedural (Global) optimizations

- Enlarge scope of analysis to entire procedure
  - more opportunities for optimization
  - have to deal with branches, merges, and loops
- Can do constant propagation, common subexpression elimination, etc. at global level
- Can do new things, e.g. loop optimizations

Optimizing compilers usually work at this level

Intraprocedural (Global) Optimizations

Two data structures are commonly used to help analyze the of procedure body.

Control flow graph (CFG) captures flow of control

- nodes are IL statements, or whole basic blocks
- edges represent control flow
- node with multiple successors = branch/switch
- node with multiple predecessors = merge
- loop in graph = loop

Data flow graph (DFG) capture flow of data

A common one is def/use chains:

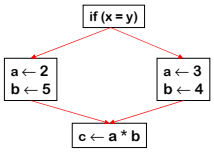
- nodes are def(inition)s and uses
- edge from def to use
- a def can reach multiple uses
- a use can have multiple reaching defs

Control-flow Graph

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example

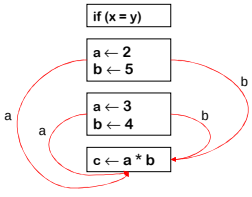


Data-flow Graph – Use/Def Chains

Models the transfer of data in the procedure

- Nodes in the graph are definitions and uses
- Edges in the graph represent data flow

Example



Analysis and Transformation

Each optimization is made up of

- some number of **analyses**
- followed by a **transformation**

Analyze CFG and/or DFG by propagating info forward or backward along CFG and/or DFG edges

- edges called **program points**
- merges in graph require combining info
- loops in graph require **iterative approximation**

Perform improving transformations based on info computed

- have to wait until any iterative approximation has converged

Analysis must be **conservative/safe/sound** so that transformations preserve program behavior

Data-flow Analysis

Data-flow analysis is a collection of techniques for *compile-time* reasoning about the *run-time* flow of values

- Almost always involves building a graph
  - Problems are trivial on a basic block
  - Global problems ⇒ control-flow graph (or derivative)
  - Whole program problems ⇒ call graph (or derivative)
- Usually formulated as a set of *simultaneous equations*
  - Sets attached to nodes and edges
  - Lattice (or semilattice) to describe values
- Desired result is usually *meet over all paths* solution
  - "What is true on every path from the entry?"
  - "Can this happen on any path from the entry?"
  - Related to the safety of optimization

Data-flow Analysis

Limitations

1. Precision – *"up to symbolic execution"*
   - Assume all paths are taken
2. Solution – cannot afford to compute MOP solution
   - Large class of problems where MOP = MFP= LFP
   - Not all problems of interest are in this class
3. Arrays – treated naively in classical analysis
   - Represent whole array with a single fact
4. Pointers – difficult (*and expensive*) to analyze
   - Imprecision rapidly adds up
   - Need to ask the right questions

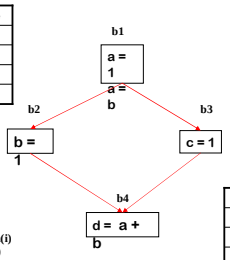
Good news:
Simple problems can carry us pretty far

Summary

For scalar values, we can quickly solve simple problems

Data-flow (Partial) Example

Block	Def	LiveUse
b1	{a}	{b}
b2	{b}	0
b3	{c}	0
b4	{d}	{a,b}



Block	LiveIn	LiveOut
b1	{b}	{b}
b2	{a}	{a,b}
b3	{a,b}	{a,b}
b4	{a,b}	0

$$\text{LiveOut(b)} = \bigcup_{i \in \text{Succ(b)}} \text{LiveIn(i)}$$

$$\text{LiveIn(b)} = \text{LiveUse(b)} \cup (\text{LiveOut(b)} - \text{Def(b)})$$

Example: Constant Propagation, Folding

Can use either the CFG or the DFG

CFG analysis info:

table mapping each variable in scope to one of

- a particular constant
- *NonConstant*
- *Undefined*

- Transformation: at each instruction:
  - if reference a variable that the table maps to a constant, then replace with that constant (constant propagation)
  - if r.h.s. expression involves only constants, and has no side-effects, then perform operation at compile-time and replace r.h.s. with constant result (constant folding)

For best analysis, do constant folding as part of analysis, to learn all constants in one pass

Example Program

```
x = 3;
y = x * x;
v = y - 2;
if (y > 10) {
   x = 5;
   y = y + 1;
} else {
    x = 6;
   y = x + 4;
}
w = y / v;
if (v > 20) {
  z = w * w;
  x = x - z;
  y = y - 1;
}
System.out.println(x);
```


Analysis of Loops

- How to analyze a loop?

```
i = 0;
x = 10;
y = 20;
while (...) {
  // what's true here?
  ...
  i = i + 1;
  y = 30;
}
// what's true here?
... x ... i ... y ...
```

A safe but imprecise approach:

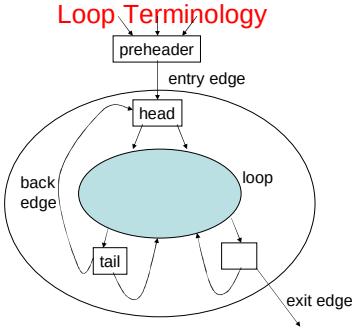
- forget everything when we enter or exit a loop

A precise but unsafe approach:

- keep everything when we enter or exit a loop

Can we do better?

Loop Terminology



31

Optimistic Iterative Analysis

1. Assuming info at loop head is same as info at loop entry
2. Then analyze loop body, computing info at back edge
3. Merge infos at loop back edge and loop entry
4. Test if merged info is same as original assumption
   a) If so, then we're done
   b) If not, then replace previous assumption with merged info, and goto step 2

32

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
   // what's true here?
   ...
   i = i + 1;
   y = 30; }
// what's true here?
... x ... i ... y ...
```

33

Why does optimistic iterative analysis work?

Why are the results always conservative?

Because if the algorithm stops, then

- the loop head info is at least as conservative as both the loop entry info and the loop back edge info
- the analysis within the loop body is conservative, given the assumption that the loop head info is conservative

Why does the algorithm terminate?

It might not!

But it does if:

- there are only a finite number of times we could merge values together without reaching the worst case info (e.g. NotConstant)

34

Loop Optimization - Code Motion

Goal: move loop-invariant calculations out of loops

Can do at source level or at intermediate code level

Source:

```
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + b[j];
  z = z + 10000;
}
```

Transformed source:

```
t1 = b[j];
t2 = 10000;
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + t1;
  z = z + t2;
}
```

35

Loop Optimization - Induction Variable Elimination

For-loop index is **induction variable**

- incremented each time around loop
- offsets & pointers calculated from it

If used only to index arrays, can rewrite with pointers

- compute initial offsets/pointers before loop
- increment offsets/pointers each time around loop
- no expensive scaling in loop

Source:

```
for (i = 0; i < 10; i = i+1) {
  a[i] = a[i] + x;
}
```

Transformed source:

```
for (p = &a[0]; p < &a[10]; p = p+4) {
  *p = *p + x;
}
```

then do loop-invariant code motion

36

Interprocedural ("Whole Program") Optimizations

- Expand scope of analysis to procedures calling each other
- Can do local & intraprocedural optimizations at larger scope
- Can do new optimizations, e.g. inlining

37

Inlining

Replace procedure call with body of called procedure

Source:

```
double pi = 3.1415927;
...
double circle_area(double radius) {
   return pi * (radius * radius);
}
...
double r = 5.0;
...
double a = circle_area(r);
```

After inlining:

```
...
double r = 5.0;
...
double a = pi * r * r;
```

(Then what?)

38

Summary

Enlarging scope of analysis yields better results

- today, most optimizing compilers work at the intraprocedural (global) level

Optimizations organized as collections of passes, each rewriting IL in place into better version

Presence of optimizations makes other parts of compiler (e.g. intermediate and target code generation) easier to write

39

Additional Material

40

Another example: live variable analysis

Want the set of live variables at each pt. in program

- live: *might be used later in the program*

Supports dead assignment elimination, register allocation

What info computed for each program point?

What is the requirement for this info to be conservative?

How to merge two infos conservatively?

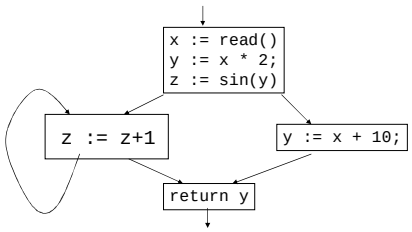
How to analyze an assignment, e.g. X := Y + Z?

- given *liveVars* before (or after?), what is computed after (or before?)

What is live at procedure entry (or exit?)?

41

Example



42

Peephole Optimization of Jumps

Eliminate jumps to jumps

Eliminate jumps after conditional branches

"Adjacent" instructions = "adjacent in control flow"

Source code:

```
if (a < b) {
   if (c < d) {
      // do nothing
   } else {
      stmt1;
   }
} else {
   stmt2;
}
```

IL:

43

Global Register Allocation

Try to allocate local variables to registers

If life times of two locals don't overlap, can give to same register

Try to allocate most-frequently-used variables to registers first

Example:

```
int foo(int n, int x) {
  int sum; int i; int t;
  sum = x;
  for (i = n; i > 0; i=i-1) {
    sum = sum + i;
  }
  t = sum * sum;
  return t;
}
```

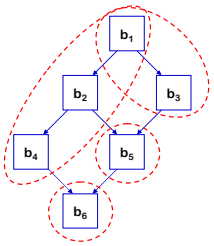
44

Handling Larger Scopes

Extended Basic Blocks

- Initialize table for b_i with table from b_{i-1}
- With single-assignment naming, can use scoped hash table

Otherwise, it is complex



The Plan:

→ Process b_1, b_2, b_4

Pop two levels

→ Process b_3 relative to b_1

→ Start clean with b_5

→ Start clean with b_6

→

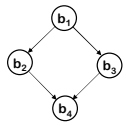
Using a scoped table makes doing the full tree of EBBs that share a common header efficient.

45

Handling Larger Scopes

To go further, we must deal with merge points

- Our simple naming scheme falls apart in b_4
- We need more powerful analysis tools
- Naming scheme becomes SSA



This requires global data-flow analysis

"Compile-time reasoning about the run-time flow of values"

1 Build a model of control-flow
2 Pose questions as sets of simultaneous equations
3 Solve the equations
4 Use solution to transform the code

Examples: LIVE, REACHES, AVAIL

46