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

"Optimize" overly optimistic

- "usually improve" better

An example

Source code:

```
x = a[i] + b[2];
c[i] = x - 5;
```

Intermediate code (if array indexing calculations explicit):

```
t1 = *(fp + ioffset);    // i
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);    // a[i]
t5 = 2;
t6 = t5 * 4;
t7 = fp + t6;
t8 = *(t7 + boffset);    // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;    // x = ...
t10 = *(fp + xoffset);  // x
t11 = 5;
t12 = t10 - t11;
t13 = *(fp + ioffset);  // i
t14 = t13 * 4;
t15 = fp + t14;
*(t15 + coffset) = t15;  // c[i] = ...
```


Kinds of optimizations

Scope of study for optimizations:

- **peephole**:
  look at adjacent instructions
- **local**:
  look at straight-line sequence of statements
- **global** (**intra**procedural):
  look at whole procedure
- **inter**procedural:
  look across procedures

Larger scope ⇒ better optimization, but more cost & complexity

Peephole optimization

After 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
```

More examples

```
subl 4, %esp
movl %eax, 0(%esp)
   ⇒
pushl %eax, %esp
```

```
movl 12(%ebp), %eax
addl 1, %eax
movl %eax, 12(%ebp)
   ⇒
incl 12(%ebp)
```

Do complex instruction selection through peephole optimization

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 code:

Algebraic simplifications

"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

Local optimization

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

Local constant propagation

If variable assigned a constant,
replace downstream uses of the variable with 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;
```


Local dead assignment elimination

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

E.g. clean-up after previous optimizations

Example:

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

Intermediate code after constant propagation:

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


Local common subexpression 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;
```


Intraprocedural ("global") optimizations

Enlarge scope of analysis to whole 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

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;
}
```


Code motion at intermediate code level

Source:

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

Unoptimized intermediate code:

```
   *(fp + ioffset) = 0;
label top;
   t0 = *(fp + ioffset);
   iffalse (t0 < 10) goto done;
   t1 = *(fp + joffset);
   t2 = t1 * 4;
   t3 = fp + t2;
   t4 = *(t3 + boffset);
   t5 = *(fp + ioffset);
   t6 = t5 * 4;
   t7 = fp + t6;
   *(t7 + aoffset) = t4;
   t9 = *(fp + ioffset);
   t10 = t9 + 1;
   *(fp + ioffset) = t10;
   goto top;
label done;
```


Loop 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

Global register allocation

Try to allocate local variables to registers

If lifetimes 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;
}
```

Inter*procedural* 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**

Inlining

Replace procedure call with body of called procedure

Source:

```
final 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?)

Summary

Enlarging scope of analysis yields better results

- today, most optimizing compilers work at the intraprocedural (aka 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

Implementing intraprocedural (global) optimizations

Construct convenient **representation** of procedure body

Control flow graph (**CFG**) captures flow of control

- nodes are IL statements (or whole basic blocks)
- edges represent possible 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

E.g. **def/use chains**:

- nodes are
  - *def*initions (assignments, writes) and
  - *uses* (reads)
- edge from def to use
- a def can *reach* multiple uses
- a use can have multiple reaching defs

Example program

```
x = 3;
y = x * x;
if (y > 10) {
   x = 5;
   y = y + 1;
} else {
   x = 6;
   y = x + 4;
}
w = y / 3;
while (y > 0) {
   z = w * w;
   x = x - z;
   y = y - 1;
}
System.out.println(x);
```


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 analysis until convergence

Perform transformations based on info computed

- have to wait until any iterative analysis has converged

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

Example: constant propagation & folding

Can use either the CFG or the DFG

- DFG: compute info about value flowing on that edge
- CFG: compute info about all variables in scope at that edge

If DFG, analysis info is one of

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

If CFG, analysis info is table mapping each variable in scope to DFG's info

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 results, 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 (z > 10) {
   x = 5;
   y = y + 1;
} else {
   y = x + 4;
}
w = y / v;
if (v > 20) {
   v = x - 1;
}
u = x + v;
```

Merging data flow analysis info

How to merge analysis info?

Constraint: merge results must be sound

- if something is believed true after the merge, then it must be true no matter which path we took into the merge
- only things true along all predecessors are true after the merge

To merge two maps of constant info, build map by merging corresponding variable infos

To merge two variable infos:

- if one is `Undefined`, keep the other
- if both same constant, keep that constant
- otherwise, degenerate to `NonConstant`

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:

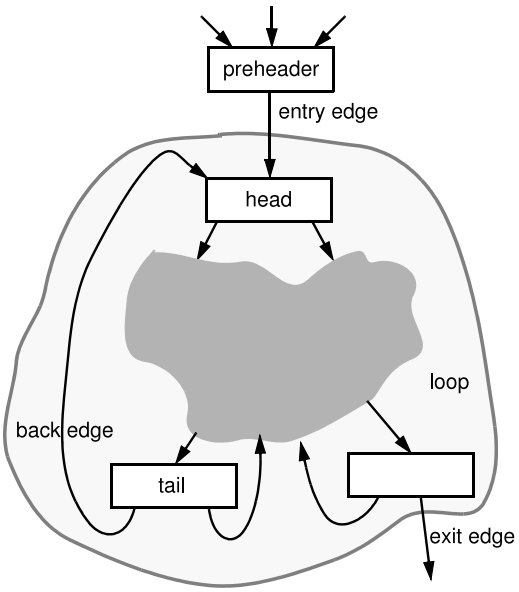
- forget everything when we enter or exit a loop

A precise but unsafe approach:

- keep everything when we enter or exit a loop

Can we do better?

Some loop terminology



Optimistic iterative analysis

1. Assume info after loop head merge node is same as info at loop entry edge

2. Then analyze loop body, computing info at back edge

3. Merge infos at loop back edge and loop entry edge

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

Example

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


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`)

Another example: live variables analysis

Want the set of variables that are **live** 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?

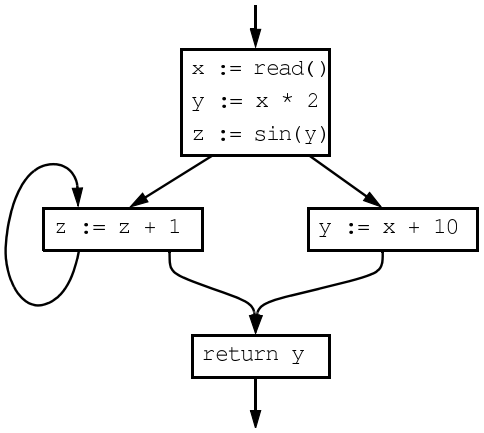
How to merge two infos conservatively?

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

- i.e., given `liveVars` before (or after), what is computed after (or before)?

What is live at procedure entry (or exit)?

Example



Intraprocedural Register Allocation

The problem:

assign machine resources (registers, stack locations)
to hold run-time data

Constraint:

simultaneously live data allocated to different locations

Goal:

minimize overhead of stack loads & stores
and register moves

Interference graph

Represent notion of "simultaneously live" using
interference graph

- nodes are "units of allocation"
- n_1 is linked by an edge to n_2 iff
  n_1 and n_2 are simultaneously live at some program point
- symmetric (undirected), not reflexive, not transitive

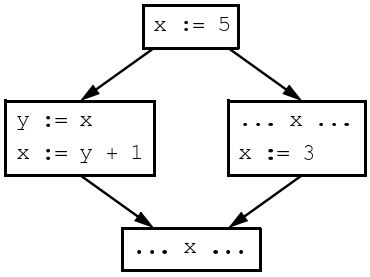
Two adjacent nodes must be allocated to distinct locations

Units of allocation

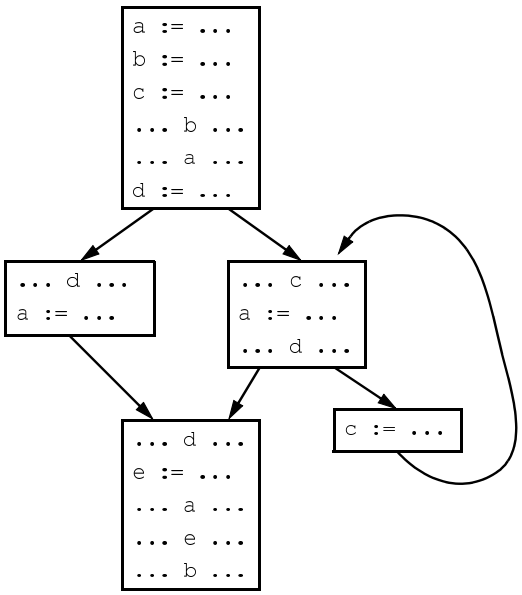
What are the units of allocation?

- option 1: variables
- option 2: distinct connected def/use chains (**live ranges**)

Example:



A bigger example



Computing interference graph

Construct interference graph as side-effect of
live variables analysis

- easy if variables are units of allocation

Construct incrementally as live vars sets modified

- when add a new var to live vars set,
  create edge from new var to all existing vars
- when merge two live vars sets,
  add one sets' vars to other set

Allocating registers using interference graph

Register allocation via graph coloring:
allocating variables to *k* registers
is equivalent to
finding a *k*-coloring of the interference graph

k-coloring: color nodes of graph using up to *k* colors,
so that adjacent nodes have different colors

Optimal graph coloring: NP-complete

- need algorithms + heuristics
  to do a decent job in reasonable time

Spilling

If can't find *k*-coloring of interference graph,
must **spill** some variables to stack,
until the resulting interference graph is *k*-colorable

Which to spill?

- least frequently accessed variables
- most conflicting variables (nodes with highest out-degree)

Weighted interference graph:
weight(*n*) =
sum over all references (uses and defs) *r* of *n*:
execution frequency of *r*

Try to spill nodes with lowest weight and highest out-degree,
if forced to spill

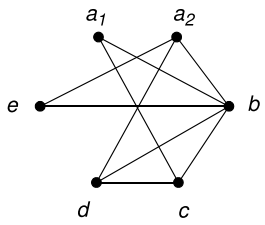
A simple greedy allocation algorithm

For all nodes, in decreasing order of weight:

- try to allocate node to a register, if possible
- if not, allocate to a stack location

Reserve 2-3 scratch registers to use when manipulating nodes
allocated to stack locations

Example



Weight Order:

c
d
a_2
b
a_1
e

Assume 3 registers available

Improvement: add simplification pre-phase

Key idea:

nodes with < *k* neighbors can be allocated
after all their neighbors, but still guaranteed a register

So remove them from the graph first

- reduces the degree of the remaining nodes

Must resort to spilling only when all remaining nodes have degree ≥ *k*

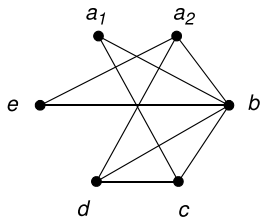
The algorithm

while interference graph not empty:
    while there exists a node with < *k* neighbors:
        remove it from the graph
        push it on a stack

    if all remaining nodes have *k* neighbors, then **blocked**:
        pick node with lowest weight/degree to spill
        remove it from the graph
        spill now, or push it on the stack to (maybe) spill later

while stack not empty:
    pop node from stack
    put back in graph
    allocate to register different from all its neighbors
        (spill to stack if none is available)

Example

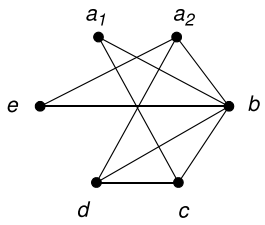


Weight Order:

c
d
a_2
b
a_1
e

Assume 3 registers available

Example

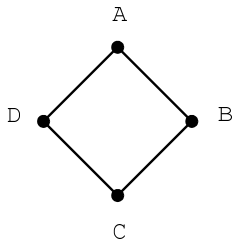


Weight Order:

c
d
a_2
b
a_1
e

Assume **2** registers available

Another example



Assume 2 registers available

Coalescing and preference hints

When generating code for copy statement like `x = y`,
if `x` and `y` were assigned same register,
then skip generating a move instruction

If register allocator sees `x = y`,
and `x` & `y` are not simultaneously live,
then it should **prefer** to assign `x` and `y` to same register

One implementation strategy:
coalesce `x` and `y` into same unit of allocation
(similar effect as copy propagation)

\+ avoids generating code for simple copies
– can cause more spilling

Another strategy: add preference hints that two things be allocated to same register

- can assign costs to (violating) preferences
- when picking a register,
  favor most preferred available register

Live range splitting

If a long live range cannot be allocated a register,
can split it into multiple separate live ranges,
linked by copy statements

- can allocate each separate piece separately
- since each piece is shorted, each may interfere with fewer things, and so be more allocatable

The reverse of coalescing

Pretty tricky to decide where to split, where to coalesce, etc.
to come up with good overall allocations

Handling calling conventions

How should register allocator deal w/ calling conventions?

Simple: calling-convention-oblivious register allocation

- spill all live caller-save registers before call, restore after call
- save all callee-save registers at entry, restore at return

Better: calling-convention-aware register allocation

- add preferred registers for formals, actuals, results
- variables live across a call interfere with caller-save regs
  - allocator knows to avoid these registers,
    save/restore code turns into normal spills
  - live range splitting into before/during/after call could be good
- procedure entry "assigns" to all callee-save registers,
  procedure exit "reads" all callee-save registers
  - simultaneously live with all variables in procedure
    $\Rightarrow$ allocator knows must spill these registers if used

Gives limited form of interprocedural register allocation

- leaf routines (try to) use only caller-save registers
- routines with calls use callee-save registers for
  variables live across calls