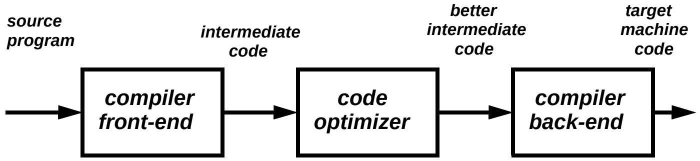
Code Optimizations

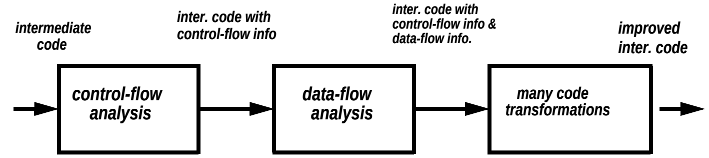


- *The intermediate code (e.g., IR tree) generated by the front-end is often* ***not efficient****.*
- *The* ***code optimizer*** *reads IR, emits* ***better*** *IR; almost all optimizations done here are* ***machine-independent****. Machine-dependent optimizations are done in the back-end.*
- *Main techniques used: graph algorithms, control- and data- flow analysis*

Copyright 1994 - 2007 Zhong Shao, Yale University

Code Optimizations (cont'd)

- *A* ***code optimizer*** *is often organized as follows:*



- ***Control-Flow Analysis ---*** *divide the IR into basic blocks, build the control-flow graph (CFG)*
- ***Data-Flow Analysis ---*** *gather data-flow information (e.g., the set of live variables).*
- ***Code Transformations ---*** *the actual optimizations*

Copyright 1994 - 2007 Zhong Shao, Yale University

Code Optimizations (cont'd)

- *Optimizations that are restricted to one basic block are called* ***local-optimizations****; otherwise, they are called* ***global optimizations***
- *Here are a* ***partial*** *list of* ***well-known*** *compiler optimizations:*

  algebraic optimizations (strength reduction, constant folding)

  common-subexpression eliminations

  copy propagations and constant propagations

  dead-code eliminations

  code-motions (i.e., lifting loop-invariants)

  induction variable eliminations; strength reductions for loops

Copyright 1994 - 2007 Zhong Shao, Yale University

Examples: Source Code

- *C code for* ***quicksort*** *(also in ASU page 588) :*

```
void quicksort(m, n);
int m, n;
{
   int i, j, v, x;
   if (n <= m) return;

   i = m-1; j = n; v = a[n];
   while (1) {
      do i = i+1; while ( a[i] < v);
      do j = j-1; while ( a[j] > v);
      if (i >= j) break;
      x = a[i]; a[i] = a[j]; a[j] = x;
   }
   x = a[i]; a[i] = a[n]; a[n] = x;

   quicksort(m,j); quicksort(i+1,n);
}
```

Copyright 1994 - 2007 Zhong Shao, Yale University

Example: Intermediate Code

- *Intermediate code for the shaded fragments of previous example:*

```
(01)   i := m - 1             (16)      t7 := 4 * i
(02)   j := n                 (17)      t8 := 4 * j
(03)  t1 := 4 * n             (18)      t9 := a[t8]
(04)   v := a[t1]             (19)   a[t7] := t9
(05)   i := i + 1             (20)     t10 := 4 * j
(06)  t2 := 4 * i             (21)  a[t10] := x
(07)  t3 := a[t2]             (22)     goto (5)
(08)  if t3 < v goto (5)      (23)     t11 := 4 * i
(09)   j := j - 1             (24)       x := a[t11]
(10)  t4 := 4 * j             (25)     t12 := 4 * i
(11)  t5 := a[t4]             (26)     t13 := 4 * n
(12)  if t5 > v goto (9)      (27)     t14 := a[t13]
(13)  if i >= j goto (23)     (28)  a[t12] := t14
(14)  t6 := 4 * i             (29)     t15 := 4 * n
(15)   x := a[t6]             (30)  a[t15] := x
```

Copyright 1994 - 2007 Zhong Shao, Yale University

Control-Flow Analysis

- *How to build the* ***Control-Flow Graph*** *(CFG) ?*

  each basic block as node, each **jump** statement as edge.
  there is always a **root** --- the "initial" node or the entry point

- *How to identify* ***loops*** *? and how to identify* ***nested loops*** *?*

  1. build the **dominator tree** from the CFG
  2. find all the **back edges**; each back edge defines a **natural loop**
  3. keep finding the **innermost** loop and reduce it to a single node.

- *Given a CFG* ***G*** *with the initial node (root)* ***r****, we say node* ***d dominates*** *node* ***n****, if every path from root* ***r*** *to* ***n*** *goes through* ***d****.*

- ***Dominator tree*** *is used to characterize the "dominate" relation:* ***r*** *as the root, the parent of a node is its* ***immediate dominator.*** *(see ASU page 602--608 for more details)*

Copyright 1994 - 2007 Zhong Shao, Yale University

Data-Flow Analysis

- ***Data-Flow Analysis*** *refers to a process in which the optimizer collects* ***data-flow information*** *at all the program points.*

- ***Examples*** *of interesting* ***data-flow information****:*

  reaching definitions: the set of definitions reaching a **program point**

  available expressions: the set of expressions available at a **point**.

  live variables: the set of variables that are live at a **point**
  ....

- ***Program points****: with each basic block, the point between two adjacent statements, or the point before the first statement and after the last. A* ***path*** *from point* p_1 *to* p_n *is a sequence of points* $p_1, ..., p_n$ *such that* p_i *and* p_{i+1} *are "****adjacent****" for all* $i = 1,...,n-1$*.*

Copyright 1994 - 2007 Zhong Shao, Yale University

Data-Flow Analysis (cont'd)

- *For each statement* ***S****, we associate it with four sets:*

  `in[S]` *: the set of data-flow info. associated with the point before* ***S***
  `out[S]` *: the set of data-flow info. associated with the point after* ***S***
  `gen[S]` *: the set of data-flow info.* ***generated*** *by* ***S***
  `kill[S]` *: the set of data-flow info.* ***destroyed*** *by* ***S***

  Naturally, if S_1 *and* S_2 *are two "****adjacent****" statements within a basic block, say,* S_2 *immediately follows* S_1*, then* $in[S_2] = out[S_1]$

- *We can define these four sets for each basic block* ***B*** *in the same way. The* ***gen*** *and* ***kill*** *sets of a basic block can be calculated from the corresponding values for each statement of that basic block.*

- ***Forward-DataFlowProblem****: the data-flow info. is calculated* ***along*** *the direction of control flow;* ***Backward-DataFlowProblem****: the data-flow info. is calculated* ***opposite*** *to the direction of control flow.*

Copyright 1994 - 2007 Zhong Shao, Yale University

Example: Reaching Definitions

- *A definition **d** reaches a point **p** if there is a path from the point immediately following **d** to **p**, such that **d** is not "**killed**" along that path.*
- *A definition of a variable **v** is "**killed**" between two points if there is a read of **v** or an assignment to **v** in between.*
- ***Goal**: given a program point **p**, find out the set of definitions that might reach point **p**. This is a **forward** data-flow problem:*

```
/* initialize out[B] assuming in[B] = ∅ for all B */
change := true;

while change do begin
  change := false;
  for each block B do begin
    in[B] := union of out[P] for all predecessor P of B;
    oldout := out[B];
    out[B] := gen[B] ∪ (in[B] - kill[B]);
    if out[B] <> oldout then change := true
  end
end
```

Copyright 1994 - 2007 Zhong Shao, Yale University

Other Data-Flow Problems

- ***Use-Definition Chains:** for each use of a variable **v**, find out all the definitions that reach that use. (directly from reaching definitions info.)*
- ***Available Expressions**: an expression **x + y** is **available** at a point **p** if every path from the initial node to **p** evaluates **x + y**, and after the last such evaluation prior to reaching **p**, there are no subsequent assignments to **x** or **y**. (this is a forward data-flow problem)*
- ***Live-Variable Analysis**: a variable **x** is **live** at point **p** if the value of **x** at **p** may be used along some path starting at **p**. (this is a backward data-flow problem)*
- ***Definition-Use Chains**: for each program point **p**, compute the set of uses **s** of a variable **x** such that there is a path from **p** to **s** that does not redefine **x**. (backward data-flow problem)*

Copyright 1994 - 2007 Zhong Shao, Yale University

Using Data-Flow Info.

- ***Common Subexpression Eliminations**: a flow graph with available expression information. (ASU page 634)*

For every statement **s** of the form **x := y + z** such that **y+z** is available at the beginning of **s**'s block, neither **y** nor **z** is defined prior to s in that block.

1. discover all the last evaluations of **y+z** that reach **s**'s block
2. create a new variable u.
3. replace each statement **w := y+z** found in (1) by
   u := y + z
   w := u
4. replace statement **s** by **x := u**

Copyright 1994 - 2007 Zhong Shao, Yale University

Using Data-Flow Info. (cont'd)

- ***Copy Propagations:** a flow graph plus the ud-chains and du-chains information, and also some copy-statement info. (see ASU page 638)*

  for each copy **s : x := y** , determine all the uses of **x** that reached by this definition of **x**, then for each use of x, determine **s** is the only definitions that reachs this use, if so, replace the use of **x** with **y.**

- ***Loop Invariants**: a flow graph plus the ud-chains information*

  a statement is a *loop invariant* if its operands are all constants, or its reaching definitions are loop invariants or from outside the loop.

- *For **more examples**, see the ASU section 10.7.*
- ***Challenges**: what if there are procedure calls, pointer dereferencing ...? also, how to make these algorithms more efficient ?*

Copyright 1994 - 2007 Zhong Shao, Yale University

Static-Single Assignment

- ***Motivation**: how to make data-flow analysis more efficient & powerful ?*
- ***Static-Single Assignment** (SSA) form --- an extension of CFG :*

```
v := 4
z := v + 5
v := 6
y := v + 7

if P
then v := 4
else v := 6

u = v + y
```

SSA transformation →

```
v1 := 4
z := v1 + 5
v2 := 6
y := v2 + 7

if P
then v3 := 4
else v4 := 6

v5 = φ(v3,v4)
u = v5 + y
```

- ***Main idea #1**: each assignment to a variable is given a unique name, and all of the uses reached by that assignment are renamed to match the assignment's new name.*

Copyright 1994 - 2007 Zhong Shao, Yale University

Static-Single Assignment (cont'd)

- ***Main idea #2**: after each branch-join node, a special form of assignment called a* ϕ-function *is inserted.* $\boldsymbol{\phi(v_1, v_2, \ldots, v_n)}$ *means that if the runtime execution comes from the **i**-th predecessor, then the above* ϕ-functio *returns the value of* $\mathbf{v_i}$ *.*

- ***Why SSA is good** ? SSA significantly simplifies the representation of many kinds of dataflow information; data flow algorithms built on def-use chains, etc. gain* **asymptotic** *efficiency.*

  In SSA, each use is reached by a unique def, so the size of def-use chains is linear to the number of edges in the CFG.

  In non-SSA, the def-use chains are much bigger.

Copyright 1994 - 2007 Zhong Shao, Yale University

SSA Construction [Cytron91]

- *Turn every "**preserving**" def into a "**killing**" def, by copying potentially unmodified values (at subscripted defs, call sites, aliased defs, etc.)*
- *Every ordinary definition of **v** defines a new name.*
- *At each node in the flow graph where multiple definitions of **v** <u>meets</u>, a* ϕ-function *is introduced to represent yet another new name of **v**.*
- *Uses are renamed by their dominating definitions (where uses at a* ϕ-function *are regarded as belonging to the appropriate predecessor node of the* ϕ-function*).*
- ***Code Size**: the f-function inserted in SSA can increase the code size, but only **linearly**; in practice, the ratio of **SSA** over **OLD** is 0.6 - 2.4.*

Copyright 1994 - 2007 Zhong Shao, Yale University