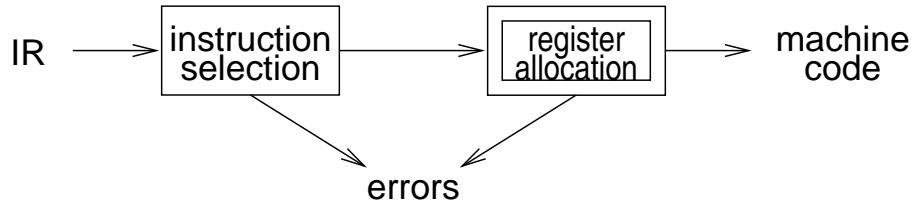


Register allocation



Register allocation:

- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult
 - ⇒ NP-complete for $k \geq 1$ registers

Copyright © 2007 by Antony L. Hosking. *Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from hosking@cs.purdue.edu.*

Liveness analysis

Problem:

- IR contains an unbounded number of temporaries
- machine has bounded number of registers

Approach:

- temporaries with disjoint *live* ranges can map to same register
- if not enough registers then *spill* some temporaries (i.e., keep them in memory)

The compiler must perform *liveness analysis* for each temporary:

It is *live* if it holds a value that may be needed in future

Control flow analysis

Before performing liveness analysis, need to understand the control flow by building a *control flow graph* (CFG):

- nodes may be individual program statements or basic blocks
- edges represent potential flow of control

Out-edges from node n lead to *successor* nodes, $succ[n]$

In-edges to node n come from *predecessor* nodes, $pred[n]$

Example

$a \leftarrow 0$
 $L_1 : b \leftarrow a + 1$
 $c \leftarrow c + b$
 $a \leftarrow b \times 2$
if $a < N$ goto L_1
return c

Liveness analysis

Gathering liveness information is a form of *data flow analysis* operating over the CFG:

- liveness of variables “flows” around the edges of the graph
- assignments *define* a variable, v :
 - $def(v)$ = set of graph nodes that define v
 - $def[n]$ = set of variables defined by n
- occurrences of v in expressions *use* it:
 - $use(v)$ = set of nodes that use v
 - $use[n]$ = set of variables used in n

Definitions

- v is *live* on edge e if there is a directed path from e to a *use* of v that does not pass through any $def(v)$
- v is *live-in* at node n if live on any of n 's in-edges
- v is *live-out* at n if live on any of n 's out-edges
- $v \in use[n] \Rightarrow v$ live-in at n
- v live-in at $n \Rightarrow v$ live-out at all $m \in pred[n]$
- v live-out at $n, v \notin def[n] \Rightarrow v$ live-in at n

Liveness analysis

Define:

$$\begin{aligned}in[n] &= \text{variables live-in at } n \\out[n] &= \text{variables live-out at } n\end{aligned}$$

Then:

$$\begin{aligned}out[n] &= \bigcup_{s \in \text{SUCC}(n)} in[s] \\succ[n] = \phi &\Rightarrow out[n] = \phi\end{aligned}$$

Note:

$$\begin{aligned}in[n] &\supseteq use[n] \\in[n] &\supseteq out[n] - def[n]\end{aligned}$$

$use[n]$ and $def[n]$ are constant (independent of control flow)

Now, $v \in in[n]$ iff. $v \in use[n]$ or $v \in out[n] - def[n]$

Thus, $in[n] = use[n] \cup (out[n] - def[n])$

Iterative solution for liveness

foreach n

$in[n] \leftarrow \phi$

$out[n] \leftarrow \phi$

repeat

foreach n

$in'[n] \leftarrow in[n];$

$out'[n] \leftarrow out[n];$

$in[n] \leftarrow use[n] \cup (out[n] - def[n])$

$out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$

until $in'[n] = in[n] \wedge out'[n] = out[n], \forall n$

Notes

- should order computation of inner loop to follow the “flow”
- liveness flows *backward* along control-flow arcs, from *out* to *in*
- nodes can just as easily be basic blocks to reduce CFG size
- could do one variable at a time, from *uses* back to *defs*, noting liveness along the way

Iterative solution for liveness

Complexity: for input program of size N

- $\leq N$ nodes in CFG
 - $\Rightarrow \leq N$ variables
 - $\Rightarrow N$ elements per *in/out*
 - $\Rightarrow O(N)$ time per set-union
 - **for** loop performs constant number of set operations per node
 - $\Rightarrow O(N^2)$ time for **for** loop
 - each iteration of **repeat** loop can only add to each set
 - sets can contain at most every variable
 - \Rightarrow sizes of all in and out sets sum to $2N^2$,
 - bounding the number of iterations of the **repeat** loop
- \Rightarrow worst-case complexity of $O(N^4)$
- ordering can cut **repeat** loop down to 2-3 iterations
 - $\Rightarrow O(N)$ or $O(N^2)$ in practice

Least fixed points

There is often more than one solution for a given dataflow problem (see example).

Any solution to dataflow equations is a *conservative approximation*:

- v has some later use downstream from n
 $\Rightarrow v \in \text{out}(n)$
- but not the converse

Conservatively assuming a variable is live does not break the program; just means more registers may be needed.

Assuming a variable is dead when really live *will* break things.

Many possible solutions but we want the “smallest”: the least fixpoint. The iterative algorithm computes this least fixpoint.