

CS322 W'05 Lecture Notes
Lecture 10.1

Minimizing Registers Needed to Evaluate Expression Trees

Key idea (Sethi & Ullman): At each node, **first** evaluate subtree requiring largest number of registers to evaluate. Can then save result of this evaluation in a register while doing other subtree.

1. Label each node with minimum number of registers needed to evaluate subtree.

```
risc_label(t)
  if isLeaf(t) then
    t->label = 1 (depends on instruction set)
  else
    label(t->left)
    label(t->right)
    if (t->left->label == t->right->label)
      t->label = t->left->label + 1
    else
      t->label = max(t->left->label,
                    t->right->label)
```

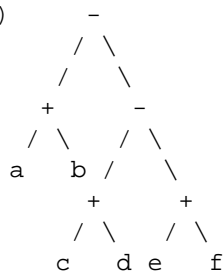
2. Use labels to guide order of code emission; emit code for higher-numbered subtree **first**.

3. If we run out of registers (when does this happen?), spill to temporary memory locations.

Register Allocation for Expressions

Choice of evaluation order can affect number of registers needed, e.g.:

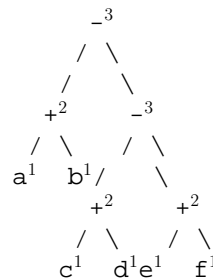
$$(a+b) - ((c+d) - (e+f))$$



If we compute left child first, need 4 regs, but doing right child first needs only 3.

```
load a,r1      load c,r1
load b,r2      load d,r2
add r1,r2,r1   add r1,r2,r1
load c,r2      load e,r2
load d,r3      load f,r3
add r2,r3,r2   add r2,r3,r2
load e,r3      sub r1,r2,r1
load f,r4      load a,r2
add r3,r4,r3   load b,r3
sub r2,r3,r2   add r2,r3,r2
sub r1,r2,r1   sub r2,r1,r1
```

Sethi-Ullman Numbering Example



Other Issues in Tree Evaluation Order

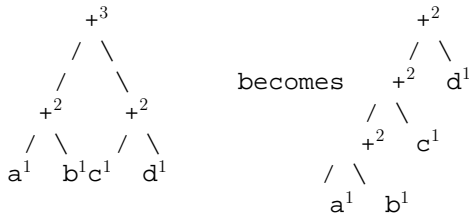
Some machines (e.g., X86) allow one operand to be a complex expression, while the other must be a register, which also holds the result (“accumulator” style):

```
add 37, r0    ; r0 <- r0 + 37
add b, r1     ; r1 <- r1 + b
sub r0, r1    ; r1 <- r1 - r0
```

- These machines have different Sethi-Ullman numbering, e.g., right **leaves** may require no registers at all.

- If we must spill registers, it is better to evaluate left child of non-commutative operators (like $-$, $/$) **last**, because they will require their left operand in a register anyhow.

Can use associativity to make trees “less bushy,” e.g.



Basic Block Example

```
prod := 0;
i := 1;
while i <= 20 do
  prod := prod + a[i] * b[i];
  i := i + 1
end
```

```
→ 1. prod := 0
   2. i := 1
→ 3. if i > 20 goto 14
→ 4. t1 := i * 4
   5. t2 := addr a
   6. t3 := *(t2+t1)
   7. t4 := i * 4
   8. t5 := addr b
   9. t6 := *(t5+t4)
  10. t7 := t3 * t6
  11. prod := prod + t7
  12. i := i + 1
  13. goto 3
→ 14. ---
```

Basic Blocks

- Extend analysis of register use to program units larger than expressions but still completely analyzable at compile time.

- **Basic Block** = sequence of instructions with single entry & exit.

- If first instruction of BB is executed, so is remainder of block (in order).

- To calculate basic blocks:

(1) Determine **BB leaders** (\rightarrow):

(a) First statement in routine

(b) Target of any jump (conditional or unconditional).

(c) Statement following any jump.

(What about subroutine calls?)

(2) Basic block extends from leader to (but not including) next leader (or end of routine).

BB Code Generation using Liveness

Can combine code generation with “greedy” register allocation: bring each variable into a register when first needed, and leave it there as long as it’s needed (if possible).

Maintain **register descriptors** saying which variable is in each register, and **address descriptors** saying where (in memory and/or a register) each variable is.

For each IR instruction $x := y \text{ op } z$:

1. If y isn’t in a register, load it into a free one, updating descriptors.

2. Similarly for z .

3. If y and/or z are no longer live following this instruction, mark their registers as free.

4. Choose a free register for x , updating descriptors.

5. Generate instruction $\text{op } r_y, r_z, r_x$.

For the special case $x := y$, load y into a register, if necessary, and then mark that register as holding x too.

- Must now be careful not to free a register unless **none** of its associated variables is live.

- Registers behave like a **cache** for memory locations.

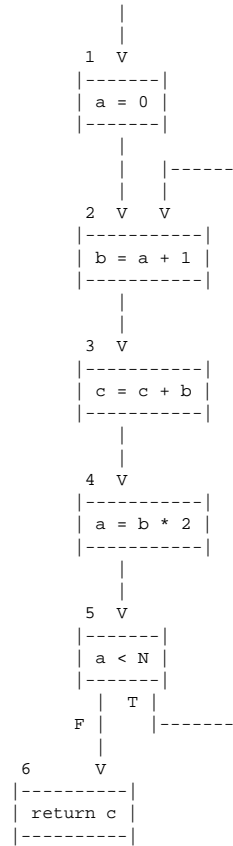
- Nasty problems for source-level debuggers and dump utilities – where **is** that variable?!?

Example

Source code: `d := (a-b) + (a-c) + (a-c)`

		Live after inst:	
		a b c	
IR: <code>t := a - b</code>		a c t	
<code>u := a - c</code>		t u	
<code>v := t + u</code>		u v	
<code>d := v + u</code>			d
		Descriptors after Instructions	
IR Statement	Code	Regs	Addr
		-	a,b,c:mem
<code>t := a - b</code>	<code>ld a,r0</code> <code>ld b,r1</code> <code>sub r0,r1,r1</code>	r0:a r1:t	a:r0,mem b,c:mem t:r1
<code>u := a - c</code>	<code>ld c,r2</code> <code>sub r0,r2,r0</code>	r0:u r1:t	a,b,c:mem u:r0 t:r1
<code>v := t + u</code>	<code>add r1,r0,r1</code>	r0:u r1:v	a,b,c:mem u:r0 v:r1
<code>d := v + u</code>	<code>add r0,r1,r0</code> <code>st r0,d</code>	r0:d r1:v	a,b,c:mem d:r0,mem

Control-flow graph



Control-flow Graphs

To assign registers on a per-procedure basis, need to perform liveness analysis on entire procedure, not just basic blocks.

To analyze the properties of entire procedures with multiple basic blocks, we use a **control-flow** graph.

In simplest form, control flow graph has one node per statement, and an edge from n_1 to n_2 if control can ever flow directly from statement 1 to statement 2.

We write $pred[n]$ for the set of predecessors of node n , and $succ[n]$ for the set of successors.

(In practice, usually build control-flow graphs where each node is a basic block, rather than a single statement.)

Example routine:

```

a = 0
L: b = a + 1
  c = c + b
  a = b * 2
  if a < N goto L
  return c
    
```

Liveness Analysis using Dataflow Analysis

Working from the future to the past, we can determine the **edges** over which each variable is live.

In the example:

b is live on $2 \rightarrow 3$ and on $3 \rightarrow 4$.

a is live from on $1 \rightarrow 2$, on $4 \rightarrow 5$, and on $5 \rightarrow 2$ (but not on $2 \rightarrow 3 \rightarrow 4$).

c is live throughout (including on entry $\rightarrow 1$).

Can see that two registers suffice to hold a,b,c.

Dataflow Equations

We can do liveness analysis (and many other analyses) via **dataflow** analysis.

A node **defines** a variable if its corresponding statement assigns to it.

A node **uses** a variable if its corresponding statement mentions that variable in an expression (e.g., on the rhs of assignment).

For any variable v , define

- $def[v]$ = set of graph nodes that define v
- $use[v]$ = set of graph nodes that use v

Similarly, for any node n , define

- $def[n]$ = set of variables defined by node n ;
- $use[v]$ = set of variables used by node n .

Solution

For correctness, order in which we take nodes doesn't matter, but it turns out to be fastest to take them in roughly reverse order:

node	use def		1st		2nd		3rd	
	use	def	out	in	out	in	out	in
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

Implementation issues:

- Algorithm always terminates, because each iteration must enlarge at least one set, but sets are limited in size (by total number of variables).
- Time complexity is $O(N^4)$ worst-case, but between $O(N)$ and $O(N^2)$ in practice.
- Typically do analysis using entire basic blocks as nodes.
- Can compute liveness for all variables in parallel (as here) or independently for each variable, on demand.
- Sets can be represented as bit vectors or linked lists; best choice depends on set density.

Setting up Equations

A variable is **live** on an edge if there is a directed path from that edge to a **use** of the variable that does not go through any **def**.

A variable is **live-in** at a node if it is live on any in-edge of that node; it is **live-out** if it is live on any out-edge.

Then the following equations hold:

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

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

We want the **least fixed point** of these equations: the smallest in and out sets such that the equations hold.

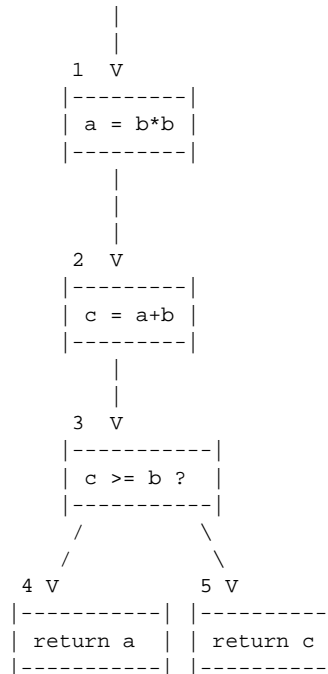
We can find this solution by **iteration**:

- Start with empty sets
- Use equations to add variables to sets, one node at a time.
- Repeat until sets don't change any more.

Adding additional variables to the sets is safe, as long as the sets still obey the equations, but inaccurately suggests that more live variables exist than actually do.

Static vs. Dynamic Liveness

Consider the following graph:



Static Liveness (continued)

Is a live-out at node 2? It depends on whether control flow ever reaches node 4.

A smart compiler could answer no.

A smarter compiler could answer similar questions about more complicated programs.

But **no** compiler can ever **always** answer such questions correctly. This is a consequence of the **uncomputability** of the **Halting Problem**.

So we must be content with **static** liveness, which talks about paths of control-flow edges, and is just a **conservative** approximation of **dynamic liveness**, which talks about actual execution paths.

Halting Problem

Theorem There is no program H that takes an input any program P and input X , and (without infinite-looping) returns true if $P(X)$ halts and false if $P(X)$ infinite-loops.

Proof Suppose there were such an H . From it, construct the function

```
F(Y) = if H(Y,Y) then (while true do ()) else 1
```

Now consider $F(F)$.

- If $F(F)$ halts, then, by the definition of H , $H(F, F)$ is true, so the then clause executes, so $F(F)$ does not halt.

- But, if $F(F)$ loops forever, then $H(F, F)$ is false, so the else clause is taken, so $F(F)$ halts.

Hence $F(F)$ halts if and only if it doesn't halt.

Since we've reached a contradiction, the initial assumption is wrong: there can be no such H .

Corollary No program $H'(P, X, L)$ can tell, for any program P , input X , and label L within P , whether L is ever reached on an execution of P on X .

Proof If we had H' , we could construct H . Consider a program transformation T that, from any program P constructs a new program by putting a label L at the end of the program, and changing every halt to goto L . Then $H(P, X) = H'(T(P), X, L)$.

Register Interference Graphs

Mixing instruction selection and register allocation gets confusing; need a more systematic way to look at the problem.

- Initially generate code assuming an infinite number of "logical" registers; calculate live ranges

Previous Example:

```

                                Live after instr.
ld a,t0           ; a:t0    t0
ld b,t1           ; b:t1    t0 t1
sub t0,t1,t2      ; t:t2    t0   t2
ld c,t3           ; c:t3    t0   t2 t3
sub t0,t3,t4      ; u:t4          t2   t4
add t2,t4,t5      ; v:t5          t4 t5
add t5,t4,t6      ; d:t6                    t6
st t6,d
    
```

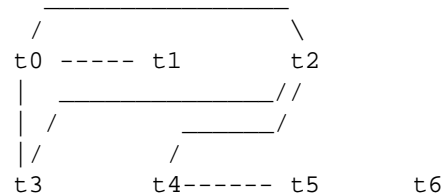
- Build a **register interference graph**, which has

- a node for each logical register.

- an edge between two nodes if the corresponding registers are simultaneously live.

Coloring Interference Graphs

Interference Graph Example:



A **coloring** of a graph is an assignment of colors to nodes such that no two connected nodes have the same color. (Like coloring a map, where nodes=countries and edges connect countries with common border.)

Suppose we have k physical registers available. Then aim is to color interference graph with k or fewer colors. This implies we can allocate logical registers to physical registers without spilling.

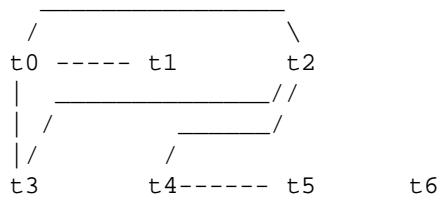
In general case, determining whether a graph can be k -colored is hard (N.P. Complete, and hence probably exponential).

But a simple heuristic will **usually** find a k -coloring if there is one.

Graph Coloring Heuristic

1. Choose a node with fewer than k neighbors.
2. Remove that node. Note that if we can color the resulting graph with k colors, we can also color the original graph, by giving the deleted node a color different from all its neighbors.
3. Repeat until **either**
 - there are no nodes with fewer than k neighbors, in which case we must spill; **or**
 - the graph is gone, in which case we can color the original graph by adding the deleted nodes back in one at a time and coloring them.

Example:



Finds a 3-coloring. There cannot be a 2-coloring (why not?).

Each “color” corresponds to a physical register, so 3 registers will do for this example.