

# CS577 Modern Language Processors

## Spring 2008

### Lecture 6

## REGISTER ALLOCATION

Task: Manage scarce resources (registers) in environment with imperfect information (static program text) about dynamic program behavior.

General aim is to keep frequently-used values in registers as much as possible, to lower memory traffic. Can have a **large** effect on program performance.

Variety of approaches are possible, differing in sophistication and in scope of analysis used.

Allocator may be unable to keep every “live” variable in registers; must then “spill” variables to memory. Spilling adds new instructions, which often affects the allocation analysis, requiring a new iteration.

# LIVENESS

To determine how long to keep a given variable (or temporary) in a register, need to know the range of instructions for which the variable is **live**.

A variable or temporary is **live** immediately following an instruction if its current value will be needed in the future (i.e., it will be used again, and it won't be changed before that use).

```

                                ! live after instruction:
mov 3, %t2                      ! %t2
mov %t2, %t3                    ! %t2 %t3
add %t3, 4, %t4                 | %t2      %t4
add %t2, %t4, %t4               |          %t4
st %t4, [a]                     | (nothing)
```

It's easy to calculate liveness for a consecutive series of instructions without branches, just by working backwards.

## LIVENESS (2)

But if a value can stay in a register over a jump, things get harder.

```

                                ! live after instruction:
1      mov 0, %t1                ! %t1      %t3
2 L1:  add %t1, 1, %t2           !      %t2 %t3
3      add %t3, %t2, %t3        !      %t2 %t3
4      mul %t2, 2, %t1          ! %t1      %t3
5      cmp %t1, 1000           ! %t1      %t3
6      bl  L1                    ! %t1      %t3
7      return %t3               ! (nothing)
```

To calculate liveness in this case requires **iterative flow analysis** and the result is only **conservative approximation** to true liveness (more later).

The **live range** of a variable is the set of instructions which leave it live. E.g. in 2nd example, live range of %t1 is {1, 4, 5, 6}.

Basic idea: If two variables have disjoint live ranges, they can occupy the same physical register.

So in both examples, 2 physical registers suffice to allocate all vars.

## LINEAR SCAN ALLOCATION

Using live ranges turns out to be computationally expensive (more later).

A simple alternative is to approximate each live range by a **live interval**.

This is the consecutive interval of instructions between the first and last use of each variable.

```

                                ! live after instruction:
1      mov 0, %t1                ! %t1      %t3
2 L1:  add %t1, 1, %t2           !      %t2 %t3
3      add %t3, %t2, %t3        !      %t2 %t3
4      mul %t2, 2, %t1          ! %t1      %t3
5      cmp %t1, 1000            ! %t1      %t3
6      bl L1                     ! %t1      %t3
7      return %t3               ! (nothing)
```

Live ranges: %t1: 1,4,5,6 %t2:2,3 %t3:1,2,3,4,5,6

Live intervals: %t1: [1,6] %t2: [2,3] %t3: [1,6]

(Revised) Basic idea: if two temporaries have non-overlapping live intervals, they can occupy the same physical register.

## LINEAR SCAN ALLOCATION ALGORITHM DETAILS

1. Compute  $startpoint[i]$  and  $endpoint[i]$  of live interval  $i$  for each variable. Store the intervals in a list in order of increasing start point.
2. Initialize set  $active := \emptyset$  and pool of free registers = all usable registers.
3. For each live interval  $i$  in order of increasing start point:
  - (a) For each interval  $j$  in  $active$ , in order of increasing end point
    - If  $endpoint[j] \geq startpoint[i]$  break to step (b).
    - Remove  $j$  from  $active$ .
    - Add  $register[j]$  to pool of free registers.
  - (b) Set  $register[i] :=$  next register from pool of free registers, and remove it from pool. (If pool is already empty, need to spill.)
  - (c) Add  $i$  to  $active$ , sorted by increasing end point.

A more complicated form of linear scan does take advantage of knowing precise live ranges (expressed as "holes" in live intervals).

## COMPUTING LIVENESS

Liveness is another property that is computed using dataflow analysis. Classic algorithm is defined on standard CFG (not necessarily in SSA).

$\text{gen}[n]$  = set of variables used by node  $n$

$\text{kill}[n]$  = set of variables defined by node  $n$

$s$	$\text{gen}[s]$	$\text{kill}[s]$
$t \leftarrow b \text{ bop } c$	$\{b, c\}$	$\{t\}$
$t \leftarrow M[b]$	$\{b\}$	$\{t\}$
$M[a] \leftarrow b$	$\{a, b\}$	$\{\}$
if $a \text{ relop } b$ then $L$	$\{a, b\}$	$\{\}$
$t \leftarrow f(a_1, \dots, a_n)$	$\{a_1, \dots, a_n\}$	$\{t\}$

Note that flow equations for this problem are **backwards**, i.e., data flows in reverse direction from control flow.

$$\text{in}[n] = \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n])$$
$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

In this case we want the fixed point solution having the **smallest** sets.

## LIVENESS EXAMPLE

```
1      a ← 0
2  L:   b ← a + 1
3      c ← c + b
4      a ← b * 2
5      if a < N goto L
6      f(c)
```

Assume that  $a, b, c$  are local variables not used after the termination of this code fragment.

We can extract the problem parameters:

$n$	$\text{succ}[n]$	$\text{gen}[n]$	$\text{kill}[n]$
6	-	$c$	-
5	2,6	$a$	-
4	5	$b$	$a$
3	4	$b, c$	$c$
2	3	$a$	$b$
1	2	-	$a$

## SOLVING EXAMPLE

Here's a solution. The control flow equations are solved as usual, but it is more efficient (takes fewer iterations) to fill them in (roughly) reverse execution order, computing `out[]` before `in[]`. That's because liveness is a "backwards" flow problem.

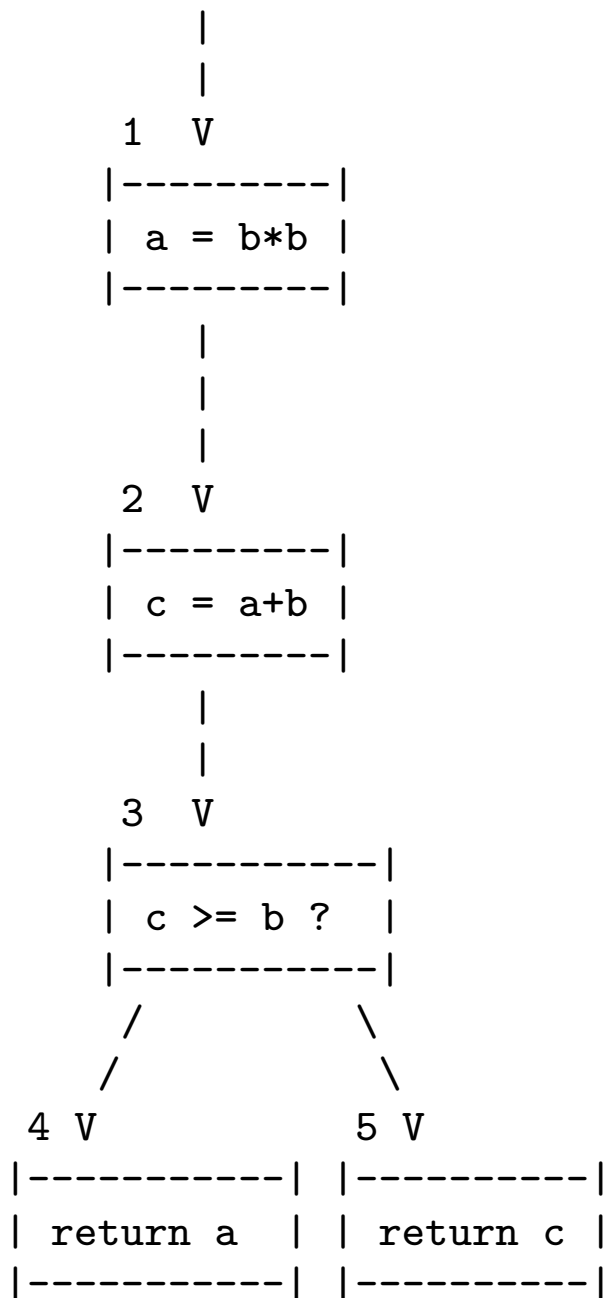
	iteration 0		iteration 1		iteration 2		iteration 3	
n	out[n]	in[n]	out[n]	in[n]	out[n]	in[n]	out[n]	in[n]
6	-	-	-	c	-	c	-	c
5	-	-	c	a,c	a,c	a,c	a,c	a,c
4	-	-	a,c	b,c	a,c	b,c	a,c	b,c
3	-	-	b,c	b,c	b,c	b,c	b,c	b,c
2	-	-	b,c	a,c	b,c	a,c	b,c	a,c
1	-	-	a,c	c	a,c	c	a,c	c

Note that in this example, no more than two of  $\{a, b, c\}$  are ever simultaneously live, so two registers will suffice to hold these variables at all times.

## VISUALIZING RESULTS

		Live ranges	Live Intervals
1	a ← 0	a            c	a            c
2	L: b ← a + 1	b    c	a    b    c
3	c ← c + b	b    c	a    b    c
4	a ← b * 2	a            c	a            c
5	if a < N goto L	a            c	a            c
6	f(c)		

## STATIC VS. DYNAMIC LIVENESS EXAMPLE



## STATIC LIVENESS (2)

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.

# Coloring Register Interference Graphs

More general mechanism for doing register allocation is by translation to **graph coloring**.

- 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.
- Attempt to **color** the nodes of the graph so that no two nodes connected by an edge have the same color.

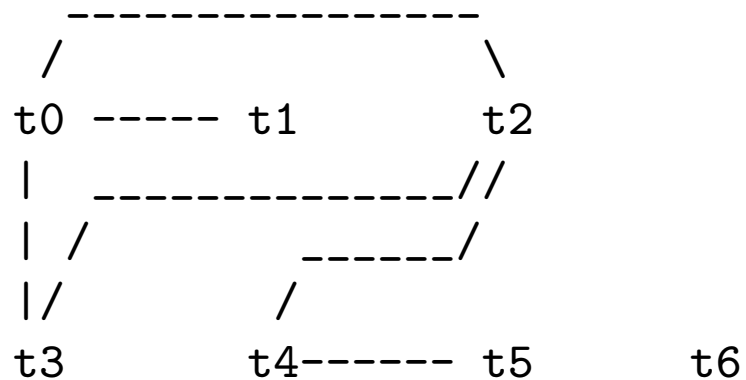
(Like coloring a map, where nodes=countries and edges connect countries with a common border.)

- If we have  $k$  physical registers, we try to color with  $k$  colors.
- If this fails, we must spill and try again. (Nasty in practice.)

## EXAMPLE

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						

Live after instr.



# COLORING HEURISTICS

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.

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.

For our example, this heuristic finds a 3-coloring, which is the best we can do.

## LIVENESS AND COLORING FOR SSA GRAPH

Computing liveness is easier for SSA graphs:

- Don't need kill sets!
- Must be careful about  $\phi$ -nodes:
  - Assignments must be done in parallel.
  - Arguments are not really live simultaneously.

Recent work indicates that interference graphs derived from SSA code can be colored in polynomial time

- Should allow considerably simpler coloring-based allocation algorithms.