Linear-Scan Register Allocation

CS 352
Lecture 12
11/28/07
Introduction

- A simple local allocation algorithm
- Assume code is already scheduled
- Build a linear ordering of live ranges (also called live intervals or lifetimes)
- Scan the live range and assign registers until you run out of them - then spill
Live Interval

- Assume some numbering of instructions in the IR

- \([i,j]\) is said to be a live interval for variable \(v\) if
  - there is no instruction with number \(j' > j\) such that \(v\) is live at \(j'\)
  - No instruction with number \(i' < i\) such that \(v\) is live at \(i'\)

- Conservative approximation of live ranges:
  - There may be subranges of \([i,j]\) in which \(v\) is not live.
Basic Idea

A and D may use the same physical register; B and E may use the same register.

Assume 2 registers:

At step 3, variable C is spilled.
Summary

- Linear scan register allocation is composed of 4 simple steps:
  - Order the instructions in linear fashion
    - Many have proposed heuristics for finding the best linear order
  - Calculate the set of live intervals
    - Each temporary is given a live interval
  - Allocate a register to each interval
    - If a register is available, then allocation is possible
    - Otherwise, an already allocated register is chosen (register spill occurs)
  - Rewrite the code according to the allocation
    - Actual registers replace temporary or virtual registers
    - Spill code is generated
The Algorithm

LinearScanRegisterAllocation
  active ← {} 
  foreach live interval i in order of increasing start point 
    ExpireOldIntervals(i) 
  if length(active) = R then 
    SpillAtInterval(i) 
  else 
    register[i] ← a free register 
  add i to active, sorted by increasing end point
The Algorithm

`ExpireOldIntervals(i)`

- foreach interval j in active, in order of increasing end point:
  - if `endpoint[j] >= startpoint[i]`
  - then return
  - remove j from active
  - add register[j] to pool of free registers
The Algorithm

SpillAtInterval(i)

spill ← last interval in active
if endpoint[spill] >= endpoint[i]
    then register[i] ← register[spill]
        location[spill] ← new stack loc.
        remove spill from active
        add i to active, sorted by increasing end point
else location[i] ← new stack location
Example

- Given the following CFG:

```
L1: if t1>t2 then L3 else L2

L2:
  t3 = t1 + 1
  t0 = t3
  return t0

L3:
  t1 = t1 - 1
  goto L1
```
Example

- Find a linear ordering of instructions:
  - The a priori choice of ordering affects performance
  - Exhaustive search not feasible
  - One possible ordering: (based on IR)
    0: L1: if $t_1 > t_2$ then L3 else L2;
    1: L3: $t_1 = t_1 - 1$
    2: goto L1;
    3: L2: $t_3 = t_1 + 1$;
    4: $t_0 = t_3$;
    5: return $t_0$
Computing Live Intervals

0: L1: if \( t_1 > t_2 \) then L3 else L2;
1: L3: \( t_1 = t_1 - 1 \)
2: goto L1;
3: L2: \( t_3 = t_1 + 1 \);
4: \( t_0 = t_3 \);
5: return \( t_0 \)

Live Intervals:
\( t_0[4,5] \quad t_2[0,2] \quad \text{only overlap is } t_2 \text{ and } t_1 \)
\( t_1[0,3] \quad t_3[3,4] \)
Allocate registers to intervals

- Maintain three lists:
  - Free: set of available registers
  - Alloc: set of allocated registers
  - Active: list of active intervals ordered by increasing end points

- Assign registers:
  - Order the intervals in order of increasing end points
  - Scan the list of intervals; select the next ti
  - Free all registers assigned to intervals in Active whose interval is less than or equal to start of ti
  - If a free register exists in Free, allocate it
  - If Free is empty, then spill:
    - If last interval on the Active list end beyond the interval for ti, then ti is given that register, and ti's interval is added to active
    - If ti's interval ends at the same point or beyond the last interval in Active, then ti is given a stack location
Allocate registers to intervals

Free = \{r1,r2\}
Active = \{\}
Alloc = \{\}

-- Looking at t2, Allocate r1
Free = \{r2\}
Active = \{t2:[0,2]\}
Alloc = \{r1:t2\}

-- Looking at t1, Allocate r2
Active = \{t1:[0,3], t2:[0,2]\}
Free = \{\}
Alloc = \{r1:t2,r2:t1\}
Allocate registers to intervals

\[ t_2[0,2] \]
\[ t_1[0,3] \]
\[ t_3[3,4] \]
\[ t_0[4,5] \]

More intervals to process

start of interval 3

Free = \{r1\}
Alloc = \{r2:t1\}

-- Looking at \( t_3 \), allocate \( r_1 \)
Active = \{r1:t3,r2:t1\}

start of interval 4

Free = \{r2\}
Alloc = \{r1:t3\}

-- Looking at \( t_0 \), allocate \( r_2 \)
Active = \{r1:t3,r2:t0\}
Consider a different ordering

\[ t_1[0,5] \]
\[ t_2[0,5] \]
\[ t_3[1,2] \]
\[ t_0[2,3] \]

Free = \{r_1, r_2\}
Active = {}  
Alloc = {}  

-- Looking at \( t_1 \), Allocate \( r_1 \)
Free = \{r_2\}
Active = \{t_1:[0,5]\}
Alloc = \{r_1:t_1\}

-- Looking at \( t_2 \), Allocate \( r_2 \)
Active = \{t_1:[0,5], t_2:[0,5]\}
Free = {}
More intervals to process
Active = {t2:[0,5], t1:[0,5]}

start of interval 1
Alloc = {r2:t2, r1:t1}

-- Looking at t3
-- Free is empty
-- End of t1 > t3
-- t3 allocated to r1, t1 on stack

Free = {}
Active = {t3:[1,2], t2:[0,5]}
Alloc = {r1:t3, r2:t2}
Alloc = \{r1:t3, r2:t2\}

-- start of interval 2

-- Looking at t0

-- Free is empty (spill needed)

-- end of t2 > end of t0

-- t0 is allocated to r2, t2 on stack
Rewriting the code

- Code is rewritten with assigned registers and spill code inserted
- Spill code is additional code that may increase cycle time:

0: L1: $t1(r1) > t2(r2)$ then L3 else L2
1: spill($t1(r1)$) and free r1
2: L2: $t3(r1) = t1(stk) + 1$
3: spill($t0(r2)$) and free r2
4: $t0(r2) = t3(r1)$
5: return $t0(r2)$
6: L3: $t1(stk) = t1(stk) - 1$
7: goto L1