RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers

More Register Allocation

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Announcements

- Reminder: get ilab account
- First homework will be handed out by tomorrow
  Deadline: Friday, February 2
- First project will come out next week

Live Ranges

A **value** is *live* between its *definition* and its *uses*

- Find definitions (x ← …) and uses (y ← … x …)
- From definition to last use is its *live range*
  - → How does a second definition affect this?
- Can represent live range as an interval [*i*,*j*] (in block)
  - → ***live on exit***

Let *MAXLIVE* be the maximum, over each instruction *i* in the block, of the number of values (pseudo-registers) live at *i*.

- If MAXLIVE ≤ *k*, allocation should be easy
  - → no need to reserve *F* registers for spilling
- If MAXLIVE > *k*, some values must be spilled to memory

Finding live ranges is harder in the global case

An Example : Top-Down

- Live Ranges

1	loadI	1028	⇒ r1
2	load	r1	⇒ r2
3	mult	r1, r2	⇒ r3
4	loadI	5	⇒ r4
5	sub	r4, r2	⇒ r5
6	loadI	8	⇒ r6
7	mult	r5, r6	⇒ r7
8	sub	r7, r3	⇒ r8
9	store	r8	⇒ r1

NOTE: live sets on exit of each instruction

An Example : Top-Down

- Live Ranges

1	loadI	1028	⇒ r1	// *r1*					
2	load	r1	⇒ r2	// *r1 r2*					
3	mult	r1, r2	⇒ r3	// *r1 r2 r3*					
4	loadI	5	⇒ r4	// *r1 r2 r3 r4*					
5	sub	r4, r2	⇒ r5	// *r1*	*r3*	*r5*			
6	loadI	8	⇒ r6	// *r1*	*r3*	*r5 r6*			
7	mult	r5, r6	⇒ r7	// *r1*	*r3*		*r7*		
8	sub	r7, r3	⇒ r8	// *r1*				*r8*	
9	store	r8	⇒ r1	//					

NOTE: live sets on exit of each instruction

An Example : Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - $k = 4$, $F = 1$, $(k-F) = 3$

```
1  loadI   1028     ⇒ r1   // r1
2  load    r1       ⇒ r2   // r1 r2
3  mult    r1, r2   ⇒ r3   // r1 r2 r3
4  loadI   5        ⇒ r4   // r1 r2 r3 r4
5  sub     r4, r2   ⇒ r5   // r1    r3    r5
6  loadI   8        ⇒ r6   // r1    r3    r5 r6
7  mult    r5, r6   ⇒ r7   // r1    r3          r7
8  sub     r7, r3   ⇒ r8   // r1                      r8
9  store   r8       ⇒ r1   //
```

- Consider statements with MAXLIVE > $(k-F)$ *basic algorithm*

  Spill heuristic: - number of occurrences of virtual register
  - length of live range (tie breaker)

An Example : Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - *k* = 4, *F* = 1, (*k*-*F*) = 3

```
1  loadI   1028     ⇒ r1   // r1
2  load    r1       ⇒ r2   // r1 r2
3  mult    r1, r2   ⇒ r3   // r1 r2 r3
4  loadI   5        ⇒ r4   // r1 r2 r3 r4          -- MAXLIVE = 4
5  sub     r4, r2   ⇒ r5   // r1    r3    r5
6  loadI   8        ⇒ r6   // r1    r3    r5 r6    -- MAXLIVE = 4
7  mult    r5, r6   ⇒ r7   // r1    r3          r7
8  sub     r7, r3   ⇒ r8   // r1                   r8
9  store   r8       ⇒ r1   //
```

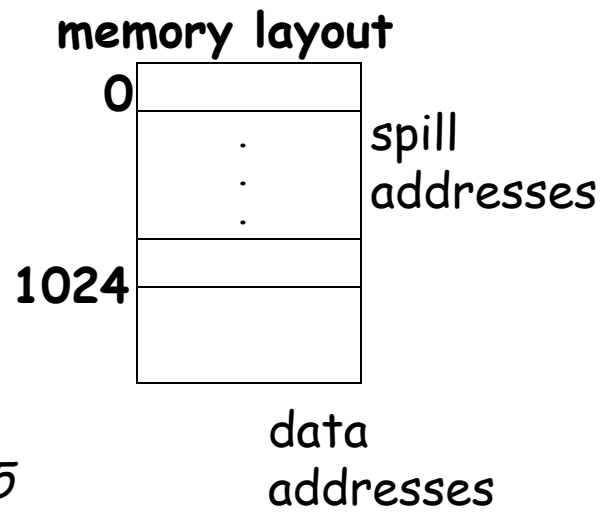
- Consider statements with MAXLIVE > (*k*-*F*) *basic algorithm*

  Spill heuristic: - number of occurrences of virtual register
  - length of live range (tie breaker)

An Example : Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - $k = 4, F = 1, (k-F) = 3$

```
1  loadI   1028     ⇒ r1   // r1
2  load    r1       ⇒ r2   // r1 r2
3  mult    r1, r2   ⇒ r3   // r1 r2 r3
4  loadI   5        ⇒ r4   // r1 r2 r3 r4
5  sub     r4, r2   ⇒ r5   // r1    r3    r5
6  loadI   8        ⇒ r6   // r1    r3    r5 r6
7  mult    r5, r6   ⇒ r7   // r1    r3          r7
8  sub     r7, r3   ⇒ r8   // r1                r8
9  store   r8       ⇒ r1   //
```



- Consider statements with MAXLIVE > (k-F) *basic algorithm*

  Spill heuristic: - number of occurrences of virtual register
  - length of live range (tie breaker)

Note: EAC Top down algorithm does not look at live ranges and MAXLIVE, but counts overall occurrences across entire basic block

An Example : Top-Down

- 3 physical registers for allocation: ra, rb, rc
- 1 physical register designated to be in the feasible set *F*

```
1  loadI     1028     ⇒ ra     // r1
2  load      ra       ⇒ rb     // r1 r2
3  mult      ra, rb   ⇒ f1     // r1 r2 r3
   store*    f1       ⇒ 10     // spill code  *NOT ILOC
4  loadI     5        ⇒ rc     // r1 r2 r3 r4
5  sub       rc, rb   ⇒ rb     // r1    r3    r5
6  loadI     8        ⇒ rc     // r1    r3    r5 r6
7  mult      rb, rc   ⇒ rb     // r1    r3          r7
   load*     10       ⇒ f1     // spill code  *NOT ILOC
8  sub       rb, f1   ⇒ rb     // r1                r8
9  store     rb       ⇒ ra     //
```

- Insert spill code for every occurrence of spilled virtual register in basic block using feasible register **f1**

Spill code

- A virtual register is spilled by using only registers from the feasible set (F), not the allocated set (k-F) = {ra, rb, … }
- How to insert spill code, with F = {f1, f2, … }?
  - → For the definition of the spilled value (assignment of the value to the virtual register), use a feasible register as the target register and then use an additional register to load its address in memory, and perform the store:

```
add ra, rb  ⇒ f1   // target of operation is spilled
loadI @f    ⇒ f2   // value lives at memory location @f
store f1    ⇒ f2
```

  - → For the use of the spilled value, load value from memory into a feasible register:

```
loadI @f    ⇒ f1   // value lives at memory location @f
load f1     ⇒ f1
add f1, rb  ⇒ ra   // operand is spilled
```

- How many feasible registers do we need for an *add* instruction?

Spill code

- A virtual register is spilled by using only registers from the feasible set (F), not the allocated set (k-F) = {ra, rb, … }
- How to insert spill code, with F = {f1, f2, … }?
  - → For the definition of the spilled value (assignment of the value to the virtual register), use a feasible register as the target register and then use an additional register to load its address in memory, and perform the store:

```
add ra, rb ⇒ f1   // target of operation is spilled

storeAI f1 ⇒ r0, @f
```

  - → For the use of the spilled value, load value from memory into a feasible register:

```
loadAI r0, @f ⇒ f1   // value lives at memory location @f

add f1, rb ⇒ ra   // operand is spilled
```

- How many feasible registers do we need for an *add* instruction?

Bottom-up Allocator

The idea:

- Focus on replacement rather than allocation
- Keep values "used soon" in registers
- Only parts of a live range may be assigned to a physical register (≠ top-down allocation's "all-or-nothing" approach)

Algorithm:

- Start with empty register set
- Load on demand
- When no register is available, free one

Replacement (heuristic):

- Spill the value whose next use is farthest in the future
- Sound familiar? Think page replacement ...

Next class

Bottom-up register allocation

Instruction scheduling