RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers More Register Allocation

Instruction Scheduling

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

RUTGERS

Announcements

- Reminder: get ilab account
- First project will come out on Wednesday

RUTGERS

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 ...

Our Example: Live Ranges

- Live Ranges

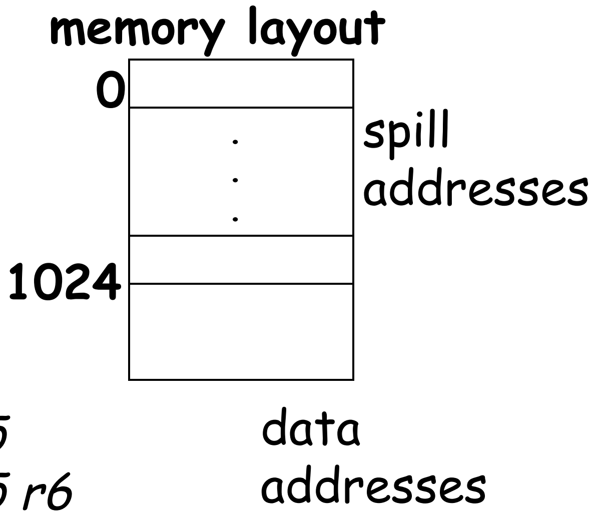
```
1  loadI   1028    ⇒ r1    // r1
2  load    r1      ⇒ r2    // r1 r2
3  mult    r1, r2  ⇒ r3    // r1 r2 r3
4  loadI   5       ⇒ r4    // r1 r2 r3 r4
5  sub     r4, r2  ⇒ r5    // r1    r3    r5
6  loadI   8       ⇒ r6    // r1    r3    r5 r6
7  mult    r5, r6  ⇒ r7    // r1    r3          r7
8 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

NOTE: live sets on exit of each instruction

An Example : Bottom-up

- Bottom up (3 registers to allocate)

```
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   //
```



An Example : Bottom-up

- Bottom up (3 physical registers to allocate: ra, rb, rc)

	source code			life ranges	register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ r1	*// r1*	*r1*		
2	load	r1	⇒ r2	*// r1 r2*	*r1*	*r2*	
3	mult	r1, r2	⇒ r3	*// r1 r2 r3*	*r1*	*r2*	*r3*
4	loadI	5	⇒ r4	*// r1 r2 r3 r4*	*r4*	*r2*	*r3*
5	sub	r4, r2	⇒ r5	*// r1 r5 r3*	*r4*	*r5*	*r3*
6	loadI	8	⇒ r6	*// r1 r5 r3 r6*	*r6*	*r5*	*r3*
7	mult	r5, r6	⇒ r7	*// r1 r7 r3*	*r6*	*r7*	*r3*
8	sub	r7, r3	⇒ r8	*// r1 r8*	*r6*	*r8*	*r3*
9	store	r8	⇒ r1	*//*	r1	r8	r3

Note: This is only one possible allocation and assignment!

RUTGERS

An Example : Bottom-up

- Bottom up (3 physical registers to allocate: ra, rb, rc)

	source code			register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ r1	*r1*		
2	load	r1	⇒ r2	*r1*	*r2*	
3	mult	r1, r2	⇒ r3	*r1*	*r2*	*r3*
4	loadI	5	⇒ r4	*r4*	*r2*	*r3*
5	sub	r4, r2	⇒ r5	*r4*	*r5*	*r3*
6	loadI	8	⇒ r6	*r6*	*r5*	*r3*
7	mult	r5, r6	⇒ r7	*r6*	*r7*	*r3*
8	sub	r7, r3	⇒ r8	*r6*	*r8*	*r3*
9	store	r8	⇒ r1	r1	r8	r3

Let's generate code now!

An Example : Bottom-up

- Bottom up (3 physical registers to allocate: ra, rb, rc)

	source code				register allocation and assignment(on exit)		
				write	ra	rb	rc
1	loadI	1028	⇒ **ra**	←	*r1*		
2	load	r1	⇒ r2		*r1*	*r2*	
3	mult	r1, r2	⇒ r3		*r1*	*r2*	*r3*
4	loadI	5	⇒ r4		*r4*	*r2*	*r3*
5	sub	r4, r2	⇒ r5		*r4*	*r5*	*r3*
6	loadI	8	⇒ r6		*r6*	*r5*	*r3*
7	mult	r5, r6	⇒ r7		*r6*	*r7*	*r3*
8	sub	r7, r3	⇒ r8		*r6*	*r8*	*r3*
9	store	r8	⇒ r1		r1	r8	r3

For written registers, use current register assignment.

An Example : Bottom-up

- Bottom up (3 physical registers to allocate: ra, rb, rc)

	source code			read	register allocation and assignment(on exit)		
					ra	rb	rc
1	loadI	1028	⇒ ra		*r1*		
2	load	**ra**	⇒ r2		*r1*	*r2*	
3	mult	r1, r2	⇒ r3		*r1*	*r2*	*r3*
4	loadI	5	⇒ r4		*r4*	*r2*	*r3*
5	sub	r4, r2	⇒ r5		*r4*	*r5*	*r3*
6	loadI	8	⇒ r6		*r6*	*r5*	*r3*
7	mult	r5, r6	⇒ r7		*r6*	*r7*	*r3*
8	sub	r7, r3	⇒ r8		*r6*	*r8*	*r3*
9	store	r8	⇒ r1		r1	r8	r3

For read registers, use previous register assignment.

RUTGERS

An Example : Bottom-up

- Bottom up (3 physical registers to allocate: ra, rb, rc)

	source code			write	register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ ra		*r1*		
2	load	ra	⇒ **rb**	←	*r1*	*r2*	
3	mult	r1, r2	⇒ r3		*r1*	*r2*	*r3*
4	loadI	5	⇒ r4		*r4*	*r2*	*r3*
5	sub	r4, r2	⇒ r5		*r4*	*r5*	*r3*
6	loadI	8	⇒ r6		*r6*	*r5*	*r3*
7	mult	r5, r6	⇒ r7		*r6*	*r7*	*r3*
8	sub	r7, r3	⇒ r8		*r6*	*r8*	*r3*
9	store	r8	⇒ r1		r1	r8	r3

An Example : Bottom-up

- Bottom up (3 physical registers to allocate: ra, rb, rc)

	source code				register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ ra		*r1*		
2	load	ra	⇒ rb		*r1*	*r2*	
3	mult	ra, rb	⇒ rc		*r1*	*r2*	*r3*
	store*	ra	⇒ 10	spill code	*r1*	*r2*	*r3*
4	loadI	5	⇒ r4	**NOT ILOC*	*r4*	*r2*	*r3*
5	sub	r4, r2	⇒ r5		*r4*	*r5*	*r3*
6	loadI	8	⇒ r6		*r6*	*r5*	*r3*
7	mult	r5, r6	⇒ r7		*r6*	*r7*	*r3*
8	sub	r7, r3	⇒ r8		*r6*	*r8*	*r3*
9	store	r8	⇒ r1		r1	r8	r3

Insert spill code.

An Example : Bottom-up

- Bottom up (3 physical registers to allocate: ra, rb, rc)

	source code				register allocation and assignment(on exit)		
					ra	rb	rc
1	loadI	1028	⇒ ra		*r1*		
2	load	ra	⇒ rb		*r1*	*r2*	
3	mult	ra, rb	⇒ rc		*r1*	*r2*	*r3*
	store	ra	⇒ 10	spill code	*r1*	*r2*	*r3*
4	loadI	5	⇒ ra		*r4*	*r2*	*r3*
5	sub	ra, rb	⇒ rb		*r4*	*r5*	*r3*
6	loadI	8	⇒ ra		*r6*	*r5*	*r3*
7	mult	rb, ra	⇒ rb		*r6*	*r7*	*r3*
8	sub	rb, rc	⇒ rb		*r6*	*r8*	*r3*
9	store	r8	⇒ r1		r1	r8	r3

RUTGERS

An Example : Bottom-up

- Bottom up (3 physical registers to allocate: ra, rb, rc)

	source code				register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ ra		*r1*		
2	load	ra	⇒ rb		*r1*	*r2*	
3	mult	ra, rb	⇒ rc		*r1*	*r2*	*r3*
	store*	ra	⇒ 10	spill code	*r1*	*r2*	*r3*
4	loadI	5	⇒ ra		*r4*	*r2*	*r3*
5	sub	ra, rb	⇒ rb		*r4*	*r5*	*r3*
6	loadI	8	⇒ ra		*r6*	*r5*	*r3*
7	mult	rb, ra	⇒ rb		*r6*	*r7*	*r3*
8	sub	rb, rc	⇒ rb		*r6*	*r8*	*r3*
	load*	10	⇒ ra	spill code	*r1*	*r8*	*r3*
9	store	r8	⇒ r1	**NOT ILOC*	r1	r8	r3

Insert spill code.

An Example : Bottom-up

- Bottom up (3 physical registers to allocate: ra, rb, rc)

	source code				register allocation and assignment(on exit)		
					ra	rb	rc
1	loadI	1028	⇒ ra		*r1*		
2	load	ra	⇒ rb		*r1*	*r2*	
3	mult	ra, rb	⇒ rc		*r1*	*r2*	*r3*
	store*	ra	⇒ 10	spill code	*r1*	*r2*	*r3*
4	loadI	5	⇒ ra		*r4*	*r2*	*r3*
5	sub	ra, rb	⇒ rb		*r4*	*r5*	*r3*
6	loadI	8	⇒ ra		*r6*	*r5*	*r3*
7	mult	rb, ra	⇒ rb		*r6*	*r7*	*r3*
8	sub	rb, rc	⇒ rb		*r6*	*r8*	*r3*
	load*	10	⇒ ra	spill code	*r1*	*r8*	*r3*
9	store	rb	⇒ ra		r1	r8	r3

Done.

RUTGERS

Spilling revisited

- Bottom up (3 physical registers to allocate: ra, rb, rc)

	source code				register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ ra	create value	*r1*		
2	load	ra	⇒ rb		*r1*	*r2*	
3	mult	ra, rb	⇒ rc		*r1*	*r2*	*r3*
				no spill code	*r1*	*r2*	*r3*
4	loadI	5	⇒ ra		*r4*	*r2*	*r3*
5	sub	ra, rb	⇒ rb		*r4*	*r5*	*r3*
6	loadI	8	⇒ ra		*r6*	*r5*	*r3*
7	mult	rb, ra	⇒ rb		*r6*	*r7*	*r3*
8	sub	rb, rc	⇒ rb		*r6*	*r8*	*r3*
	loadI	1028	⇒ ra	spill code	*r1*	*r8*	*r3*
9	store	rb	⇒ ra		r1	r8	r3

Rematerialization: Re-computation is cheaper than store/load to memory

Bottom-up spilling revisited

source code example

```
     . . .

1  add        r1, r2    ⇒ r3
2  add        r4, r5    ⇒ r6
     . . .

x  Need to spill either r3 or r6 ; both used farthest in the future
   . . .

y  add        r3,r6     ⇒ r27
```

Should r3 or r6 be spilled before instruction x
(Assume neither register value can be rematerialized) ?

Bottom-up spilling revisited

source code example

```
  | . . .
  |
1 | add        r1, r2    ⇒ r3
2 | add        r4, r5    ⇒ r6
  | . . .
  |
x | Need to spill either r3 or r6 ; both used farthest in the future
  | . . .
  |
y | add        r3,r6     ⇒ r27
```

Should r3 or r6 be spilled before instruction x
(Assume neither register value can be rematerialized) ?

What if r3 has been spilled before instruction x, but r6 has not?

RUTGERS

Bottom-up spilling revisited

source code example

Spilling clean vs. dirty virtual registers: clean is cheaper!

```
  | ...
1 | add     r1, r2   ⇒ r3
2 | add     r4, r5   ⇒ r6
  | ...
x | Need to spill either r3 or r6 ; both used farthest in the future
  | ...
y | add     r3,r6    ⇒ r27
```

Should r3 or r6 be spilled before instruction x
(Assume neither register value can be rematerialized) ?

What if r3 has been spilled before instruction x, but r6 has not?
Spilling clean register (r3) avoids storing value of dirty register (r6).
Note: there is no reassignment of virtual registers; issue is whether
a virtual registers has been spilled already (clean) or not!

RUTGERS

First Programming Project

Implement and evaluate three (four?) different register allocators (two top-down, and one bottom-up)

Not a group project! Start early! You will run into problems, and you will need time to fix them.

May use any language supported on ilab to implement your register allocator; however, TAs only support a subset of these languages (list will be posted)

Deliverables: (1) working code (on ilab) and (2) evaluation report; there are two separate deadlines for each part; the code is 60%, and the report is 40% of your project grade

Look at our web site for details. **Project will come out soon**. Good luck!

RUTGERS

Instruction Scheduling

EaC Chapter 12

Motivation

- Instruction latency (pipelining)
  several cycles to complete instructions; instructions can be issued every cycle
- Instruction-level parallelism (VLIW, superscalar)
  execute multiple instructions per cycle

Issues

- Reorder instructions to reduce execution time
- Static schedule – insert NOPs to preserve correctness
- Dynamic schedule – hardware pipeline stalls
- Preserve correctness, improve performance
- Interactions with other optimizations (register allocation!)

RUTGERS

Instruction Scheduling

EaC Chapter 12

Motivation

- Instruction latency (pipelining)
  several cycles to complete instructions; instructions can be issued every cycle
- Instruction-level parallelism (VLIW, superscalar)
  execute multiple instructions per cycle

Issues

- Reorder instructions to reduce execution time
- Static schedule – insert NOPs to preserve correctness
- Dynamic schedule – hardware pipeline stalls
- Preserve correctness, improve performance
- Interactions with other optimizations (register allocation!)
- Note: After register allocation, code shape contains real, not virtual registers ==> register may be redefined

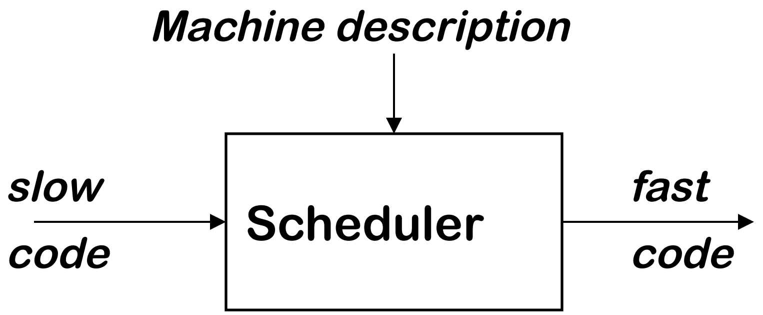
RUTGERS

Instruction Scheduling (Engineer's View)

The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time

The Concept



The task

- **Produce correct code**
- **Minimize wasted (idle) cycles**
- **Scheduler operates efficiently**

Data Dependences (stmt./instr. level)

Dependences $\Rightarrow$ defined on memory locations / registers

Statement/instruction b depends on statement/instruction a if there exists:

- true of flow dependence
  a writes a location/register that b later reads (RAW conflict)
- anti dependence
  a reads a location/register that b later writes (WAR conflict)
- output dependence
  a writes a location/register that b later writes (WAW conflict)

Dependences define ORDER CONSTRAINTS that need to be respected in order to generate correct code.

```
true          anti          output
a =             = a         a =
  = a         a =           a =
```

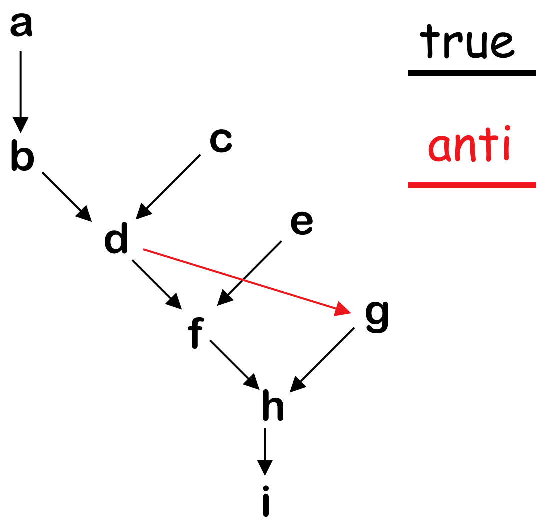
Instruction Scheduling (The Abstract View)

To capture properties of the code, build a precedence graph *G*

- Nodes *n* $\in$ *G* are operations with *type(n)* and *delay(n)*
- An edge $e = (n_1,n_2) \in G$ if n_2 depends on n_1

```
a:   loadAI    r0,@w   ⇒ r1
b:   add       r1,r1   ⇒ r1
c:   loadAI    r0,@x   ⇒ r2
d:   mult      r1,r2   ⇒ r1
e:   loadAI    r0,@y   ⇒ r3
f:   mult      r1,r3   ⇒ r1
g:   loadAI    r0,@z   ⇒ r2
h:   mult      r1,r2   ⇒ r1
i:   storeAI   r1      ⇒ r0,@w
```

The Code



The Precedence Graph

All other dependences (output & anti) are covered, i.e., are satisfied through the dependencies shown

RUTGERS

Next class

More on Instruction Scheduling

Lexical Analysis

Read EaC: Chapters 2.1 – 2.5;