EECS 583 – Class 11
ILP Optimization and Intro. to Code Generation

University of Michigan

October 9, 2019

Announcements & Reading Material

- Reminder: HW 2
  - » Due next Wednes, You should have started by now
  - » Talk to Sung/Armand if you are stuck
- Class project ideas
  - » Meeting signup sheet next Wednes in class (Fall Break Monday)
  - » Think about partners/topic!
- Today's class
  - » "Compiler Code Transformations for Superscalar-Based High-Performance Systems," S. Mahlke, W. Chen, J. Gyllenhaal, W. Hwu, P, Chang, and T. Kiyohara, *Proceedings of Supercomputing '92*, Nov. 1992, pp. 808-817
  - » "Machine Description Driven Compilers for EPIC Processors", B. Rau, V. Kathail, and S. Aditya, HP Technical Report, HPL-98-40, 1998. (long paper but informative)
- Next class
  - » "The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors," P. Chang et al., IEEE Transactions on Computers, 1995, pp. 353-370.

ILP Optimization

- Traditional optimizations
  - Redundancy elimination
  - Reducing operation count
- ILP (instruction-level parallelism) optimizations
  - Increase the amount of parallelism and the ability to overlap operations
  - Operation count is secondary, often trade parallelism for extra instructions (avoid code explosion)
- ILP increased by breaking dependences
  - True or flow = read after write dependence
  - False or (anti/output) = write after read, write after write

Back Substitution

- Generation of expressions by compiler frontends is very sequential
  - Account for operator precedence
  - Apply left-to-right within same precedence
- Back substitution
  - Create larger expressions
    - Iteratively substitute RHS expression for LHS variable
  - Note – may correspond to multiple source statements
  - Enable subsequent optis
- Optimization
  - Re-compute expression in a more favorable manner

y = a + b + c – d + e – f;

1. r9 = r1 + r2
2. r10 = r9 + r3
3. r11 = r10 - r4
4. r12 = r11 + r5
5. r13 = r12 – r6

Subs r12:
r13 = r11 + r5 – r6
Subs r11:
r13 = r10 – r4 + r5 – r6
Subs r10
r13 = r9 + r3 – r4 + r5 – r6
Subs r9
r13 = r1 + r2 + r3 – r4 + r5 – r6

Tree Height Reduction

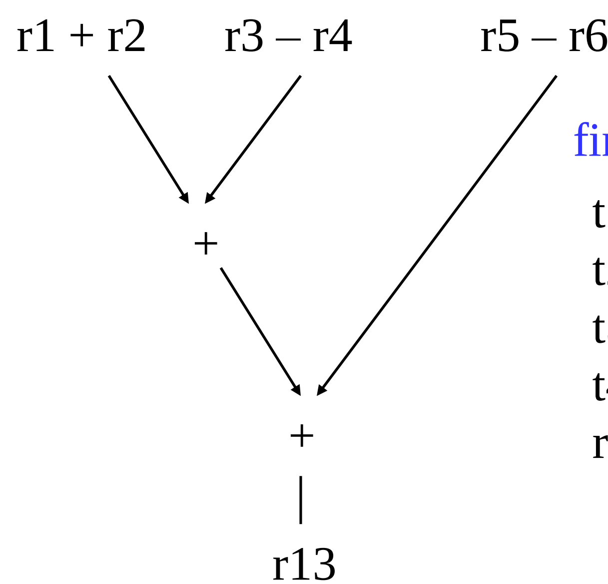
- Re-compute expression as a balanced binary tree
  - » Obey precedence rules
  - » Essentially re-parenthesize
  - » Combine literals if possible
- Effects
  - » Height reduced (n terms)
    - n-1 (assuming unit latency)
    - ceil(log2(n))
  - » Number of operations remains constant
  - » Cost
    - Temporary registers "live" longer
  - » Watch out for
    - Always ok for integer arithmetic
    - Floating-point – may not be!!

original:

r9 = r1 + r2
r10 = r9 + r3
r11 = r10 - r4
r12 = r11 + r5
r13 = r12 – r6

after back subs:

r13 = r1 + r2 + r3 – r4 + r5 – r6



final code:

t1 = r1 + r2
t2 = r3 – r4
t3 = r5 – r6
t4 = t1 + t2
r13 = t4 + t3

Class Problem

Assume: + = 1, * = 3

operand	0	0	0	1	2	0
arrival times	r1	r2	r3	r4	r5	r6

1. r10 = r1 * r2
2. r11 = r10 + r3
3. r12 = r11 + r4
4. r13 = r12 – r5
5. r14 = r13 + r6

Back susbstitute

Re-express in tree-height reduced form

Account for latency and arrival times

Class Problem - Solution

Assume: + = 1, * = 3

operand	0	0	0	1	2	0
arrival times	r1	r2	r3	r4	r5	r6

1. r10 = r1 * r2
2. r11 = r10 + r3
3. r12 = r11 + r4
4. r13 = r12 – r5
5. r14 = r13 + r6

Back susbstitute
Re-express in tree-height reduced form
Account for latency and arrival times

Expression after back substitution
r14 = r1 * r2 + r3 + r4 - r5 + r6

Want to perform operations on r1,r2,r3,r6 first due to operand arrival times

t1 = r1 * r2
t2 = r3 + r6

The multiply will take 3 cycles, so combine t2 with r4 and then r5, and then finally t1

t3 = t2 + r4
t4 = t3 – r5
r14 = t1 + t4

Equivalently, the fully parenthesized expression
r14 = ((r1 * r2) + (((r3 + r6) + r4) - r5))

Optimizing Unrolled Loops

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
       r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

unroll 3 times

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       -------------------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter2  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       -------------------
       r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter3  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

Unroll = replicate loop body n-1 times.

Hope to enable overlap of operation execution from different iterations

Not possible!

Register Renaming on Unrolled Loop

loop:	r1 = load(r2)	loop:	r1 = load(r2)
	r3 = load(r4)		r3 = load(r4)
	r5 = r1 * r3		r5 = r1 * r3
iter1	r6 = r6 + r5	iter1	r6 = r6 + r5
	r2 = r2 + 4		r2 = r2 + 4
	r4 = r4 + 4		r4 = r4 + 4
	r1 = load(r2)		r11 = load(r2)
	r3 = load(r4)		r13 = load(r4)
	r5 = r1 * r3		r15 = r11 * r13
iter2	r6 = r6 + r5	iter2	r6 = r6 + r15
	r2 = r2 + 4		r2 = r2 + 4
	r4 = r4 + 4		r4 = r4 + 4
	r1 = load(r2)		r21 = load(r2)
	r3 = load(r4)		r23 = load(r4)
	r5 = r1 * r3		r25 = r21 * r23
iter3	r6 = r6 + r5	iter3	r6 = r6 + r25
	r2 = r2 + 4		r2 = r2 + 4
	r4 = r4 + 4		r4 = r4 + 4
	if (r4 < 400) goto loop		if (r4 < 400) goto loop

Register Renaming is Not Enough!

```
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 4
       r4 = r4 + 4
      --------------------
       r11 = load(r2)
       r13 = load(r4)
       r15 = r11 * r13
iter2  r6 = r6 + r15
       r2 = r2 + 4
       r4 = r4 + 4
      --------------------
       r21 = load(r2)
       r23 = load(r4)
       r25 = r21 * r23
iter3  r6 = r6 + r25
       r2 = r2 + 4
       r4 = r4 + 4
       if (r4 < 400) goto loop
```

- Still not much overlap possible
- Problems
  - r2, r4, r6 sequentialize the iterations
  - Need to rename these
- 2 specialized renaming optis
  - Accumulator variable expansion (r6)
  - Induction variable expansion (r2, r4)

Accumulator Variable Expansion

```
        r16 = r26 = 0
loop:   r1 = load(r2)
        r3 = load(r4)
        r5 = r1 * r3
iter1   r6 = r6 + r5
        r2 = r2 + 4
        r4 = r4 + 4
      --------------------
        r11 = load(r2)
        r13 = load(r4)
        r15 = r11 * r13
iter2   r16 = r16 + r15
        r2 = r2 + 4
        r4 = r4 + 4
      --------------------
        r21 = load(r2)
        r23 = load(r4)
        r25 = r21 * r23
iter3   r26 = r26 + r25
        r2 = r2 + 4
        r4 = r4 + 4
        if (r4 < 400) goto loop
        r6 = r6 + r16 + r26
```

- Accumulator variable
  - » x = x + y or x = x – y
  - » where y is loop variant!!
- Create n-1 temporary accumulators
- Each iteration targets a different accumulator
- Sum up the accumulator variables at the end
- May not be safe for floating-point values

Induction Variable Expansion

```
       r12 = r2 + 4, r22 = r2 + 8
       r14 = r4 + 4, r24 = r4 + 8
       r16 = r26 = 0
loop:  r1 = load(r2)
       r3 = load(r4)
       r5 = r1 * r3
iter1  r6 = r6 + r5
       r2 = r2 + 12
       r4 = r4 + 12
      --------------------
       r11 = load(r12)
       r13 = load(r14)
       r15 = r11 * r13
iter2  r16 = r16 + r15
       r12 = r12 + 12
       r14 = r14 + 12
      --------------------
       r21 = load(r22)
       r23 = load(r24)
       r25 = r21 * r23
iter3  r26 = r26 + r25
       r22 = r22 + 12
       r24 = r24 + 12
       if (r4 < 400) goto loop
       r6 = r6 + r16 + r26
```

- Induction variable
  - x = x + y or x = x – y
  - where y is loop <u>invariant</u>!!
- Create n-1 additional induction variables
- Each iteration uses and modifies a different induction variable
- Initialize induction variables to init, init+step, init+2*step, etc.
- Step increased to n*original step
- Now iterations are completely independent !!

Better Induction Variable Expansion

```
         r16 = r26 = 0
loop:    r1 = load(r2)
         r3 = load(r4)
         r5 = r1 * r3
iter1    r6 = r6 + r5

--------------------
         r11 = load(r2+4)
         r13 = load(r4+4)
         r15 = r11 * r13
iter2    r16 = r16 + r15

--------------------
         r21 = load(r2+8)
         r23 = load(r4+8)
         r25 = r21 * r23
iter3    r26 = r26 + r25
         r2 = r2 + 12
         r4 = r4 + 12
         if (r4 < 400) goto loop
         r6 = r6 + r16 + r26
```

- With base+displacement addressing, often don't need additional induction variables
  - » Just change offsets in each iterations to reflect step
  - » Change final increments to n * original step

Homework Problem

loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop

Optimize the unrolled loop

Renaming
Tree height reduction
Ind/Acc expansion

loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4

r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4

r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop

Homework Problem - Answer

```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop
```

Optimize the unrolled loop

Renaming
Tree height reduction
Ind/Acc expansion

```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400)
    goto loop
```

```
loop:
  r1 = load(r2)
  r5 = r1 + 3
  r6 = r6 + r5
  r2 = r2 + 4
  r11 = load(r2)
  r15 = r11 + 3
  r6 = r6 + r15
  r2 = r2 + 4
  r21 = load(r2)
  r25 = r21 + 3
  r6 = r6 + r25
  r2 = r2 + 4
  if (r2 < 400)
     goto loop
```

after renaming and tree height reduction

```
r16 = r26 = 0
loop:
 r1 = load(r2)
 r5 = r1 + 3
 r6 = r6 + r5
 r11 = load(r2+4)
 r15 = r11 + 3
 r16 = r16 + r15
 r21 = load(r2+8)
 r25 = r21 + 3
 r26 = r26 + r25
 r2 = r2 + 12
 if (r2 < 400)
    goto loop
 r6 = r6 + r16
 r6 = r6 + r26
```

after acc and ind expansion

Code Generation

- Map optimized "machine-independent" assembly to final assembly code
- Input code
  - Classical optimizations
  - ILP optimizations
  - Formed regions (sbs, hbs), applied if-conversion (if appropriate)
- Virtual → physical binding
  - 2 big steps
  - 1. Scheduling
    - Determine when every operation executions
    - Create MultiOps
  - 2. Register allocation
    - Map virtual → physical registers
    - Spill to memory if necessary

Scheduling Operations

- Need information about the processor
  - Number of resources, latencies, encoding limitations
  - For example:
    - 2 issue slots, 1 memory port, 1 adder/multiplier
    - load = 2 cycles, add = 1 cycle, mpy = 3 cycles; all fully pipelined
    - Each operand can be register or 6 bit signed literal
- Need ordering constraints amongst operations
  - What order defines correct program execution?
- Given multiple operations that can be scheduled, how do you pick the best one?
  - Is there a best one? Does it matter?
  - Are decisions final?, or is this an iterative process?
- How do we keep track of resources that are busy/free
  - Reservation table: Resources x time

More Stuff to Worry About

- Model more resources
  - Register ports, output busses
  - Non-pipelined resources
- Dependent memory operations
- Multiple clusters
  - Cluster = group of FUs connected to a set of register files such that an FU in a cluster has immediate access to any value produced within the cluster
  - Multicluster = Processor with 2 or more clusters, clusters often interconnected by several low-bandwidth busses
    - Bottom line = Non-uniform access latency to operands
- Scheduler has to be fast
  - NP complete problem
  - So, need a heuristic strategy
- What is better to do first, scheduling or register allocation?

Schedule Before or After Register Allocation?

virtual registers	physical registers
r1 = load(r10)	**R1 = load(R1)**
r2 = load(r11)	**R2 = load(R2)**
r3 = r1 + 4	**R5 = R1 + 4**
r4 = r1 – r12	**R1 = R1 – R3**
r5 = r2 + r4	**R2 = R2 + R1**
r6 = r5 + r3	**R2 = R2 + R5**
r7 = load(r13)	**R5 = load(R4)**
r8 = r7 * 23	**R5 = R5 * 23**
store (r8, r6)	**store (R5, R2)**

Too many artificial ordering constraints if schedule after allocation!!!!

But, need to schedule after allocation to bind spill code

Solution, do both! Prepass schedule, register allocation, postpass schedule

Data Dependences

- Data dependences
  - If 2 operations access the same register, they are dependent
  - However, only keep dependences to most recent producer/consumer as other edges are redundant
  - Types of data dependences

Flow	**Output**	**Anti**
$\boxed{r1} = r2 + r3$	$\boxed{r1} = r2 + r3$	$r1 = \boxed{r2} + r3$
$r4 = \boxed{r1} * 6$	$\boxed{r1} = r4 * 6$	$\boxed{r2} = r5 * 6$

More Dependences

- Memory dependences
  - Similar as register, but through memory
  - Memory dependences may be certain or maybe
- Control dependences
  - We discussed this earlier
  - Branch determines whether an operation is executed or not
  - Operation must execute after/before a branch
  - Note, control flow (C0) is not a dependence

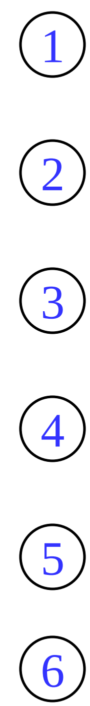
Mem-flow	**Mem-output**	**Mem-anti**	**Control (C1)**
store (r1, r2)	**store (r1, r2)**	**r2 = load(r1)**	**if (r1 != 0)**
↓	↓	↓	↓
r3 = load(r1)	**store (r1, r3)**	**store (r1, r3)**	**r2 = load(r1)**

Dependence Graph

- Represent dependences between operations in a block via a DAG
  - Nodes = operations
  - Edges = dependences
- Single-pass traversal required to insert dependences
- Example

1: r1 = load(r2)
2: r2 = r1 + r4
3: store (r4, r2)
4: p1 = cmpp (r2 < 0)
5: branch if p1 to BB3
6: store (r1, r2)

BB3:



Dependence Edge Latencies

- Edge latency = minimum number of cycles necessary between initiation of the predecessor and successor in order to satisfy the dependence
- Register flow dependence, a → b
  - Latest_write(a) – Earliest_read(b) (earliest_read typically 0)
- Register anti dependence, a → b
  - Latest_read(a) – Earliest_write(b) + 1 (latest_read typically equal to earliest_write, so anti deps are 1 cycle)
- Register output dependence, a → b
  - Latest_write(a) – Earliest_write(b) + 1 (earliest_write typically equal to latest_write, so output deps are 1 cycle)
- Negative latency
  - Possible, means successor can start before predecessor
  - We will only deal with latency >= 0, so MAX any latency with 0

Dependence Edge Latencies (2)

- Memory dependences, a → b (all types, flow, anti, output)
  - latency = latest_serialization_latency(a) – earliest_serialization_latency(b) + 1 (generally this is 1)
- Control dependences
  - branch → b
    - Op b cannot issue until prior branch completed
    - latency = branch_latency
  - a → branch
    - Op a must be issued before the branch completes
    - latency = 1 – branch_latency (can be negative)
    - conservative, latency = MAX(0, 1-branch_latency)

Class Problem

machine model

latencies

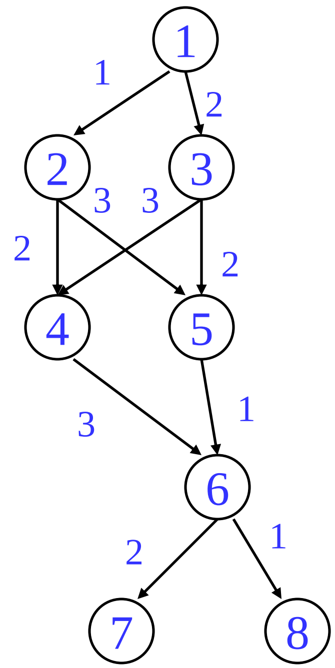
add: 1
mpy: 3
load: 2
sync 1
store: 1
sync 1

1. Draw dependence graph
2. Label edges with type and latencies

1. r1 = load(r2)
2. r2 = r2 + 1
3. store (r8, r2)
4. r3 = load(r2)
5. r4 = r1 * r3
6. r5 = r5 + r4
7. r2 = r6 + 4
8. store (r2, r5)

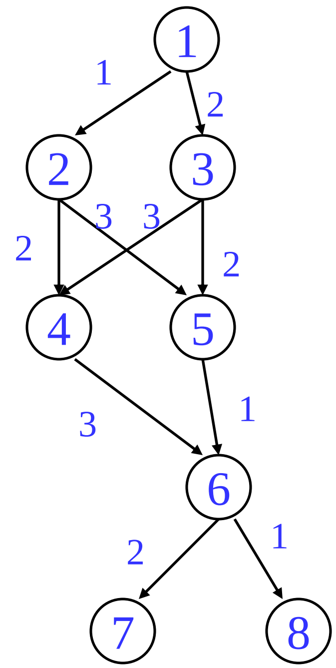
Dependence Graph Properties - Estart

- Estart = earliest start time, (as soon as possible - ASAP)
  - Schedule length with infinite resources (dependence height)
  - Estart = 0 if node has no predecessors
  - Estart = MAX(Estart(pred) + latency) for each predecessor node
  - Example



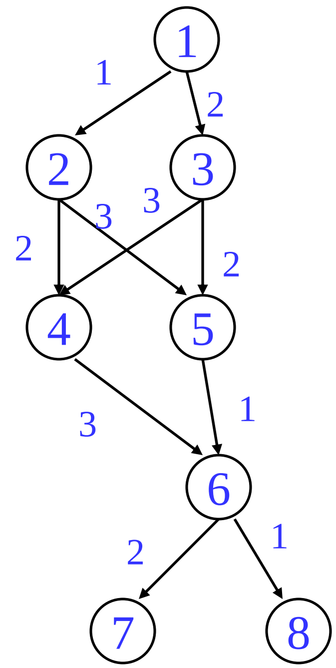
Lstart

- Lstart = latest start time, ALAP
  - » Latest time a node can be scheduled s.t. sched length not increased beyond infinite resource schedule length
  - » Lstart = Estart if node has no successors
  - » Lstart = MIN(Lstart(succ) - latency) for each successor node
  - » Example



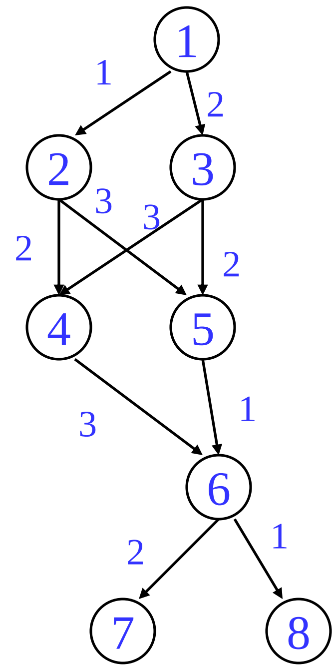
Slack

- Slack = measure of the scheduling freedom
  - Slack = Lstart – Estart for each node
  - Larger slack means more mobility
  - Example

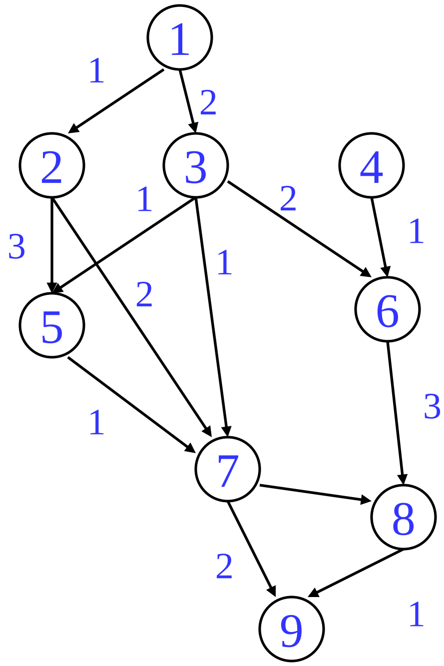


Critical Path

- Critical operations = Operations with slack = 0
  - No mobility, cannot be delayed without extending the schedule length of the block
  - Critical path = sequence of critical operations from node with no predecessors to exit node, can be multiple crit paths



Class Problem



Node	Estart	Lstart	Slack
1			
2			
3			
4			
5			
6			
7			
8			
9			

Critical path(s) =

To Be Continued…