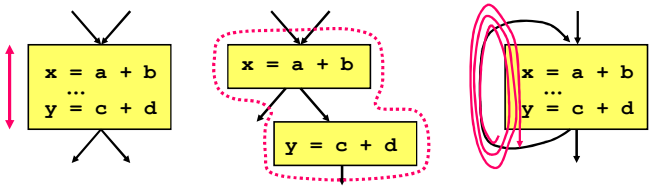
Lecture 20

Global Scheduling & Software Pipelining

[ALSU 10.4-10.5]

Scheduling Roadmap



List Scheduling:
- *within* a basic block
  (lecture 15)

Global Scheduling:
- *across* basic blocks

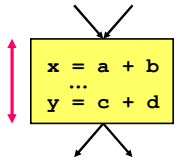
Software Pipelining:
- *across* loop iterations

Review: List Scheduling

- The most common technique for scheduling instructions within a basic block

We don't need to worry about:
- control flow

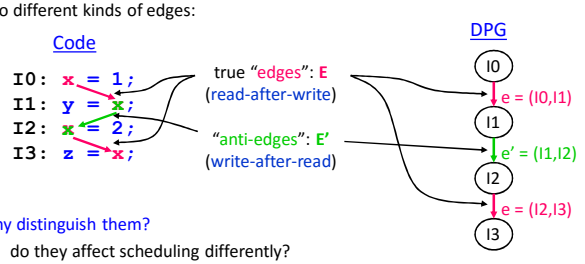
We do need to worry about:
- data dependences
- hardware resources



- Even without control flow, the problem is still NP-hard

Review: The Data Precedence Graph (DPG)

- Two different kinds of edges:

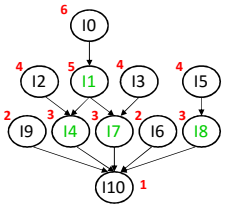


- Why distinguish them?
  - do they affect scheduling differently?
- What about output dependences?

List Scheduling with Priorities

$$priority(x) = max(\forall_{l \in leaves(DPG)} \forall_{p \in paths(x,\ldots,l)} \sum_{p_i=x}^{l} latency(p_i))$$

```
I0: a = 1
I1: f = a + x
I2: b = 7
I3: c = 9
I4: g = f + b
I5: d = 13
I6: e = 19;
I7: h = f + c
I8: j = d + y
I9: z = -1
I10: JMP L1
```

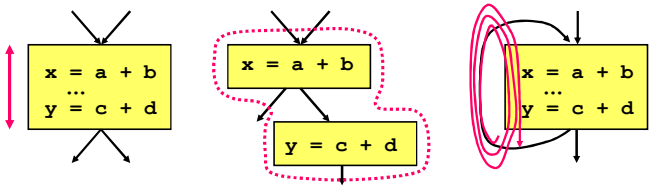


Cycle

		0
		1
		2
		3
		4
		5
		6

- 2 identical fully-pipelined FUs
- adds take 2 cycles; all other insts take 1 cycle

Scheduling Roadmap



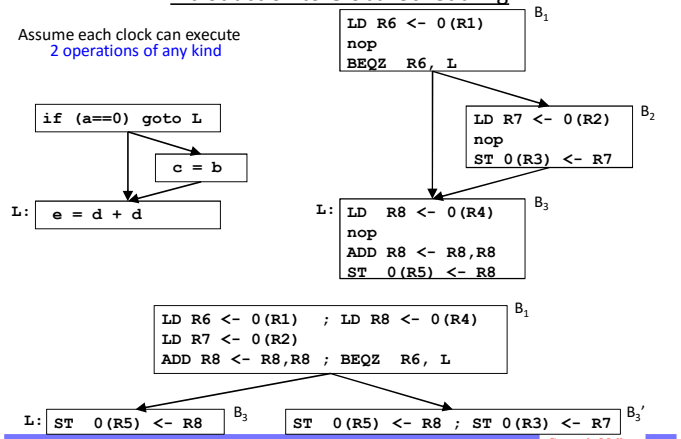
List Scheduling:
- *within* a basic block

Global Scheduling:
- *across* basic blocks

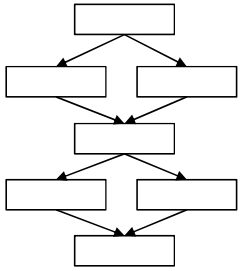
Software Pipelining:
- *across* loop iterations

Introduction to Global Scheduling

Assume each clock can execute
2 operations of any kind



Terminology

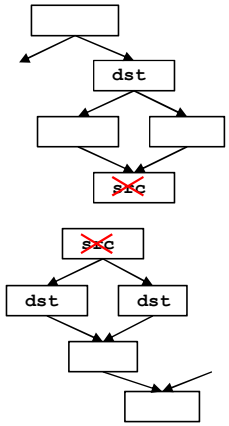


Control equivalence:
- Two operations o_1 and o_2 are *control equivalent* if o_1 is executed if and only if o_2 is executed.

Control dependence:
- An op o_2 is *control dependent* on op o_1 if the execution of o_2 depends on the outcome of o_1.

Speculation:
- An operation o is *speculatively* executed if it is executed before all the operations it depends on (control-wise) have been executed.
- Requirements:
  - does not raise an exception
  - satisfies data dependences

Code Motions

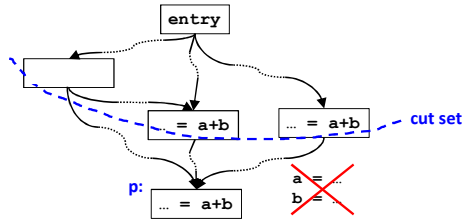


Goal: Shorten execution time probabilistically

Moving instructions up:
- Move instruction to a cut set (from entry)
- Speculation: even when not anticipated.

Moving instructions down:
- Move instruction to a cut set (from exit)
- May execute extra instruction
- Can duplicate code

Review: Code Motion for Partial Redundancy Elimination



- **Partial redundancy at p: redundant on some but not all paths**
  - Add operations to create a cut set containing a+b
  - Note: Moving operations up can eliminate redundancy
- **Constraint on placement: no wasted operation**
  - a+b is "anticipated" at B if its value computed at B will be used along ALL subsequent paths
  - a, b not redefined, no branches that lead to exit without use
- **Range where a+b is anticipated → Choice**

General-Purpose Applications

- **Lots of data dependences**
- **Key performance factor: memory latencies**
- **Move memory fetches up**
  - Speculative memory fetches can be expensive
- **Control-intensive: get execution profile**
  - Static estimation
    - Innermost loops are frequently executed
      - back edges are likely to be taken
    - Edges that branch to exit and exception routines are not likely to be taken
  - Dynamic profiling
    - Instrument code and measure using representative data

A Basic Global Scheduling Algorithm

- **Schedule innermost loops first**
- **Only upward code motion**
- **No creation of copies**
- **Move operations speculatively up only one branch**

Program Representation

- **Recall: A region in a control flow graph is:**
  - a set of basic blocks and all the edges connecting these blocks,
  - such that control from outside the region must enter through a single entry block
- **A procedure is represented as a hierarchy of regions**
  - The whole control flow graph is a region
  - Each natural loop (single entry with back edge to it) in the flow graph is a region
  - Natural loops are hierarchically nested
- **Schedule regions from inner to outer**
  - treat inner loop as a black box unit
    - can schedule around it but not into it
  - ignore all the loop back edges → get an acyclic graph

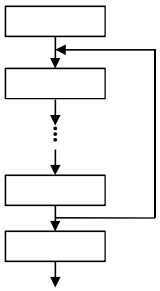
Algorithm

```
Compute data dependences;
For each region from inner to outer {
  For each basic block B in prioritized topological order {
    CandBlocks = ControlEquiv{B} ∪
                 Dominated-Successors{ControlEquiv{B}};
    CandInsts = ready operations in CandBlocks;
    For (t = 0, 1, ... until all operations from B are scheduled) {
      For (n in CandInst in priority order) {
        if (n has no resource conflicts at time t) {
          S(n) = < B, t >
          Update resource commitments
          Update data dependences
        }
      }
      Update CandInsts;
    }}}
```

Priority functions: non-speculative before speculative

Extensions

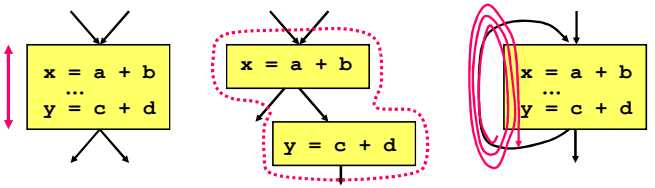
- **Prepass before scheduling: loop unrolling**
- **Especially important to move operation up loop back edges**



Global Scheduling Summary

- **Global scheduling**
  - Legal code motions
  - Heuristics

Scheduling Roadmap



List Scheduling:
- *within* a basic block

Global Scheduling:
- *across* basic blocks

Software Pipelining:
- *across* loop iterations

Example of DoAll Loops

- **Machine:**
  - Per clock: 1 read, 1 write, 1 (2-stage) arithmetic op, with hardware loop op and auto-incrementing addressing mode.
- **Source code:**

```
For i = 1 to n
    D[i] = A[i]*B[i] + c
```

- **Code for one iteration:**

```
1. LD  R5,0(R1++)
2. LD  R6,0(R2++)
3. MUL R7,R5,R6
4.
5. ADD R8,R7,R4
6.
7. ST 0(R3++),R8
```

- **Little or no parallelism within basic block**

Loop Unrolling

Schedule after unrolling by a factor of 4

```
1.L: LD
2.   LD
3.          LD
4.   MUL    LD
5.          MUL    LD
6.   ADD            LD
7.          ADD            LD
8.   ST             MUL    LD
9.                         MUL
10.         ST     ADD
11.                        ADD
12.                ST
13.                       ST      BL (L)
```

- Let **u** be the degree of unrolling:
  - Length of **u** iterations = 7+2(**u**-1)
  - Execution time per source iteration = (7+2(**u**-1)) / **u** = 2 + 5/**u**

Software Pipelined Code

```
1.   LD
2.   LD
3.   MUL    LD
4.          LD
5.          MUL    LD
6.   ADD            LD
7. L:               MUL    LD
8.   ST     ADD             LD     BL (L)
9.                          MUL
10.         ST     ADD
11.
12.                ST     ADD
13.
14.                       ST
```

- Unlike unrolling, software pipelining can give optimal result
- Locally compacted code may not be globally optimal
- DOALL: Can fill arbitrarily long pipelines with infinitely many iterations

Example of DoAcross Loop

Loop:

```
Sum = Sum + A[i];
B[i] = A[i] * c;
```

→

```
1. LD  // A[i]
2. MUL // A[i]*c
3. ADD // Sum += A[i]
4. ST  // B[i]
```

Software Pipelined Code

```
1. LD
2. MUL
3. ADD   LD
4. ST    MUL
5.       ADD
6.       ST
```

Doacross loops

- Recurrences can be parallelized
- Harder to fully utilize hardware with large degrees of parallelism

Problem Formulation

Goals:

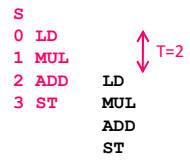
- maximize throughput
- small code size

Find:

- an identical relative schedule S(n) for every iteration
- a constant initiation interval (T)

such that

- the initiation interval is minimized



Complexity:

- NP-complete in general

Impact of Resources on Bound on Initiation Interval

- Example: Resource usage of 1 iteration
  - (assume machine can execute 1 LD, 1 ST, 2 ALU per clock)

LD, LD, MUL, ADD, ST

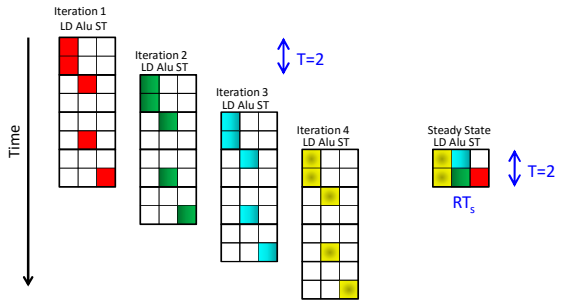
- **Lower bound on initiation interval?**

for all resource i,
number of units required by one iteration: n_i
number of units in system: R_i

Lower bound due to resource constraints: $\max_i \lceil \frac{n_i}{R_i} \rceil$

Scheduling Constraints: Resources

```
1. LD
2. LD
3. MUL
4.
5.
6. ADD
7.
8. ST
```

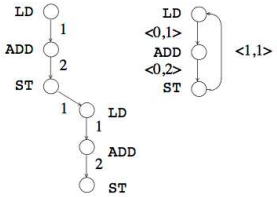


- RT: resource reservation table for single iteration
- RT_s: modulo resource reservation table

$RT_s[i] = \Sigma_{t|(t \bmod T = i)} RT[t]$

Scheduling Constraints: Precedence

```
for (i = 0; i < n; i++) {
    *(p++) = *(q++) + c
}
```



- Minimum initiation interval T? 1+2+1 = 4
- S(n): schedule for n with respect to the beginning of the schedule
- Label edges with < δ, d >
  - δ = iteration difference, d = delay

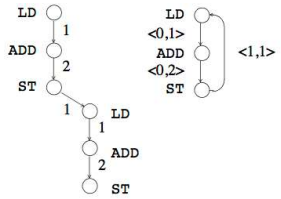
$$\delta \times T + S(n_2) - S(n_1) \geq d$$

Minimum Initiation Interval

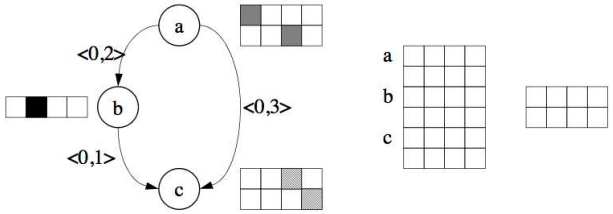
For all cycles c,

max $_c$ CycleLength(c) / IterationDifference (c)

T = 4/1 = 4



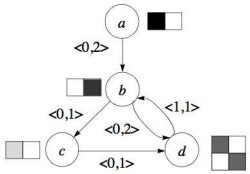
Example: An Acyclic Graph inside a loop



Algorithm: Software Pipelining Acyclic Dependence Graphs

- **Find lower bound of initiation interval: T_0**
  - based on resource constraints
- For **$T = T_0, T_0+1, \ldots$ until all nodes are scheduled**
  - For each node n in topological order
    - s_0 = earliest n can be scheduled
    - for each s = s_0 , s_0 +1, ..., s_0 +T-1
    - if NodeScheduled(n, s) break;
    - if n cannot be scheduled break;
- NodeScheduled(n, s)
  - Check resources of **n** at **s** in modulo resource reservation table
- Can always meet the lower bound if:
  - every operation uses only 1 resource, and
  - no cyclic dependences in the loop

Cyclic Graphs



- No such thing as "topological order"
- b → c; c → b

$$S(c) - S(b) \geq 1$$
$$T + S(b) - S(c) \geq 2$$

- Scheduling b constrains c, and vice versa

$$S(b) + 1 \leq S(c) \leq S(b) - 2 + T$$
$$S(c) - T + 2 \leq S(b) \leq S(c) - 1$$

See [ALSU 10.5.8] for Software Pipelining scheduling algorithm for cyclic dependence graphs

A Closer Look at Register Allocation for Software Pipelining

Software-pipelined code:

```
 1. LD
 2. LD
 3. MUL    LD
 4.        LD
 5.        MUL     LD
 6. ADD            LD
L:7.                MUL     LD
 8. ST     ADD               LD      BL L
 9.                          MUL     LD
10.        ST      ADD               LD
11.                         ST      MUL
12.                 ST      ADD
13.
14.                         ST      ADD
```

```
1. LD  R5,0(R1++)
2. LD  R6,0(R2++)
3. MUL R7,R5,R6
4.
5.
6. ADD R8,R7,R4
7.
8. ST 0(R3++),R8
```

What is the problem w.r.t. R7?

Solution: Modulo Variable Expansion

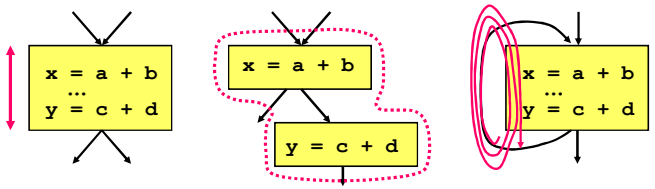
```
 1.   LD R5,0(R1++)
 2.   LD R6,0(R2++)
 3.   LD R5,0(R1++)  MUL R7,R5,R6
 4.   LD R6,0(R2++)
 5.   LD R5,0(R1++)  MUL R9,R5,R6
 6.   LD R6,0(R2++)  ADD R8,R7,R4
L 7.  LD R5,0(R1++)  MUL R7,R5,R6
 8.   LD R6,0(R2++)  ADD R8,R9,R4    ST 0(R3++),R8
 9.   LD R5,0(R1++)  MUL R9,R5,R6
10.   LD R6,0(R2++)  ADD R8,R7,R4    ST 0(R3++),R8   BL L
11.                  MUL R7,R5,R6
12.                  ADD R8,R9,R4    ST 0(R3++),R8
13.
14.                  ADD R8,R7,R4    ST 0(R3++),R8
15.
16.                                  ST 0(R3++),R8
```


Algorithm: Software Pipelining with Modulo Variable Expansion

- **Normally, every iteration uses the same set of registers**
  - introduces artificial anti-dependences for software pipelining
- **Modulo variable expansion algorithm**
  - schedule each iteration ignoring artificial constraints on registers
  - calculate life times of registers
  - degree of unrolling = $\max_r$ (lifetime$_r$ /T)
  - unroll the steady state of software pipelined loop to use different registers
- **Code generation**
  - generate one pipelined loop with only one exit (at beginning of steady state)
  - generate one unpipelined loop to handle the rest
  - code generation is the messiest part of the algorithm!

Conclusions

- **Numerical Code**
  - Software pipelining is useful for machines with a lot of pipelining and instruction level parallelism
  - Compact code
  - Limits to parallelism: dependences, critical resource



List Scheduling:
• *within* a basic block

Global Scheduling:
• *across* basic blocks

Software Pipelining:
• *across* loop iterations

Next Week: Prefetching

- Monday: Prefetching Arrays
- Wednesday: Prefetching Pointer-based Structures