Topic 12: Acyclic Instruction Scheduling

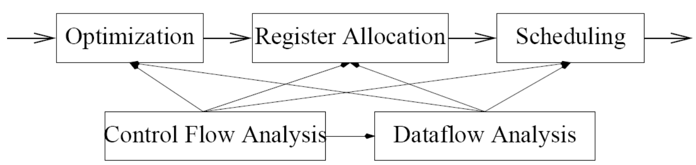
COS 320

Compiling Techniques

Princeton University
Spring 2018

Prof. David August

The Back End



The Back End:

1. Maps infinite number of virtual registers to finite number of real registers → *register allocation*
2. Removes inefficiencies introduced by front-end → *optimizer*
3. Removes inefficiencies introduced by programmer → *optimizer*
4. Adjusts pseudo-assembly composition and order to match target machine → *scheduler*

Scheduling

Multiply instruction takes 2 cycles...

```
1    r1 = r0 + 0
2    r2 = M[FP + A]
3    r3 = r0 + 4
4    r4 = M[FP + X]

  LOOP:
1    r5 = r3 * r1
2
3    r5 = r2 + r5
4    M[r5] = r4
5    r1 = r1 + 1
6    BR r1 <= 10, LOOP
```

```
1    r1 = r0 + 0
2    r2 = M[FP + A]
3    r3 = r0 + 4
4    r4 = M[FP + X]

  LOOP:
1    r5 = r3 * r1
2    r1 = r1 + 1
3    r5 = r2 + r5
4    M[r5] = r4
5    BR r1 <= 10, LOOP
```

Scheduling

Multiply instruction takes 2 cycles...
Machine executes 2 instructions per cycle...

```
1    r1 = r0 + 0
2    r2 = M[FP + A]       1    r1 = r0 + 0    r2 = M[FP + A]
3    r3 = r0 + 4          2    r3 = r0 + 4    r4 = M[FP + X]
4    r4 = M[FP + X]

  LOOP:                     LOOP:
1    r5 = r3 * r1         1    r5 = r3 * r1   r1 = r1 + 1
2    r1 = r1 + 1          2
3    r5 = r2 + r5         3    r5 = r2 + r5
4    M[r5] = r4           4    M[r5] = r4     BR r1 <= 10, LOOP
5    BR r1 <= 10, LOOP
```

Instruction Level Parallelism

- Instruction-Level Parallelism (ILP), the concurrent execution of independent assembly instructions.
- ILP is a cost effective way to extract performance from programs.
- Exploiting ILP requires global optimization and scheduling.
- Processors are becoming increasingly dependent on the ability of compilers to expose ILP.
  - Current state-of-the-art machines can execute 3 to 6 instructions per cycle if available. (i.e. Pentium III, DEC Alpha 21264)
  - Some processors rely on compiler for guidance. (i.e. Itanium)
- Current state-of-the-art compilers cannot expose this level of ILP in integer programs.

Data Dependence

- A *data dependence* is a constraint on scheduling arising from the flow of data between two instructions. Types:
  - RAW: An instruction u is *flow-dependent* on a preceding instruction d if u consumes a value computed by d.
  - WAR: An instruction d is *anti-dependent* on a preceding instruction u if d writes to a location read by u.
  - WAW: An instruction d_2 is *output-dependent* on a preceding instruction d_1 if d_1 writes to a location also written by d_2.
- Types of data:
  - Register dependence
  - Memory dependence

Data Dependence

```
     r1  =  r2 + r3

     Branch r1  <=  10, TRUE

     r4  =  r2 * r5

     r5  =  r4 + 1

TRUE:
     r4  =  r5 - 1
```

False Dependence

Eliminate WAW dependences

```
r1 =

branch

r1 =
   = r1
```

Eliminate WAR dependences

```
   = r1
r1 =
   = r1
```

- Eliminate RAW dependences?
- Register allocation vs. splitting live ranges

Control Dependence

- A *control dependence* is a constraint on scheduling arising from the control flow of the program.

```
     Branch r1  <=  10, TARGET1

     Branch r2  <=  10, TARGET2

     r4  =  r3 + 5

TARGET1:
     r5  =  r4 - 1

TARGET2:  (Assume: r4 not live here)
```

Control Dependences

Sources of Control Dependence

- Liveness
- Side-effects
  - Potentially Excepting Instructions (PEIs)
  - Memory Writes
  - Input/Output

Dependences

Latency

- Amount of time after the execution of an instruction that its result is ready.
- An instruction can have more than one latency!

Data Dependence Graph

- A *data dependence graph* consists of instructions and a set of directed data dependence edges among them in which each edge is labeled with its latency and type of dependence.
- Scheduling (code motion) must respect dependence graph.

Resources

- What does “two instructions per cycle” mean?
- *Resource* - A function of the processor that can be used by only one instruction at a time.
- Examples:
  - Fetch units
  - Decode units
  - Execution units
  - Register ports

Pipelining

Resource Map

Scheduling

- The goal of *scheduling* is to construct a sort of the dependence graph that:
  - Produces the same result - respects dependences
  - Minimizes execution time - makes maximal use of machine resources
- Scheduling is NP-hard even with simple formulation of problem.
- Use Heuristics to approximate solution.
- In practice, is exhaustive search of all schedules practical in most cases?

Heuristic: List Scheduling

- List scheduling, the most common heuristic, is $O(n_2)$.
- Create *ready queue* to hold *ready* instructions.
- An instruction is *ready* when all incoming dependences are satisfied.
- A dependence is satisfied when source of dependence are has been scheduled at least `latency` cycles earlier.

List Scheduling

```
build dependence graph
insert instructions with no incoming dependences into ready queue
WHILE (instruction are not scheduled) DO
    current_cycle_sched = FALSE
    FOREACH instruction i in ready queue DO
        IF (resources exist to schedule i in cycle) THEN
            schedule i, update ready queue
            current_cycle_sched = TRUE
    IF (NOT current_cycle_sched) THEN
        cycle++
        update ready queue
```

List Scheduling

```
  LOOP:
1   r5  =  r3  *  r1

2   r1  =  r1  +  1

3   r5  =  r2  +  r5

4   M[r5]  =  r4

5   BR r1  <=  10, LOOP
```

List Scheduling Priority

Hardware Scheduling

Machines can also do scheduling...

- hardware schedulers process code after it has been fetched
- hardware finds independent instructions
- works with legacy architectures (found in x86 ¿ Pentium)
- program knowledge more precise at run-time - memory dependence

But compiler still important.

- Hardware schedulers have a small window.
- Hardware complexity increases.
- Hardware does not benefit directly from compiler optimization.

Expression Reformulation

Loop Unrolling

```
sum = 0;
for i = 1 to 30:
  sum = sum + A[i];
```

```
0    r1 = 0              r2 = 0

Loop:

0    r3 = M[r1 + A]      r1 = r1 + 1

1

2    r2 = r2 + r3        BR r1 < 30, Loop
```

Renaming

```
0    r1 = 0              r2 = 0

Loop:

0    r3 = M[r1 + A]      r1 = r1 + 1

1

2    r2 = r2 + r3

3    r3 = M[r1 + A]      r1 = r1 + 1

4

5    r2 = r2 + r3        BR r1 < 30, Loop
```

Accumulator Expansion

```
0    r1 = 0              r2 = 0

Loop:

0    r3 = M[r1 + A]      r1 = r1 + 1

1    r4 = M[r1 + A]      r1 = r1 + 1

2    r2 = r2 + r3

3    r2 = r2 + r4        BR r1 < 30, Loop
```

Accumulator Expansion

```
0   r1 = 0              r2 = 0

Loop:

0   r3 = M[r1 + A]      r1 = r1 + 1

1   r4 = M[r1 + A]      r1 = r1 + 1

2   r5 = M[r1 + A]      r1 = r1 + 1      r2 = r2 + r3

3   r2 = r2 + r4

4   r2 = r2 + r5        BR r1 < 30, Loop
```

Induction Variable Elimination

```
0   r1 = 0              r23 = 0
1   r24 = 0             r25 = 0

Loop:

0   r3 = M[r1 + A]      r1 = r1 + 1

1   r4 = M[r1 + A]      r1 = r1 + 1

2   r5 = M[r1 + A]      r1 = r1 + 1     r23 = r23 + r3

3   r24 = r24 + r4

5   r25 = r25 + r5      BR r1 < 30, Loop

0   r2 = r23 + r24
1   r2 = r2 + r25
```

Loop Unrolling and Optimization

```
0   r13 = 0             r14 = 1
1   r15 = 2             r23 = 0
2   r24 = 0             r25 = 0

Loop:

0   r3 = M[r13 + A]     r13 = r13 + 3       r4 = M[r14 + A]
    r14 = r14 + 3       r5 = M[r15 + A]     r15 = r15 + 3

1

2   r23 = r23 + r3      r24 = r24 + r4    r25 = r25 + r5
    BR r13 < 30, Loop

0   r2 = r23 + r24
1   r2 = r2 + r25
```