List Scheduling
Compilers

Source code

Lexing (P2)

Tokens

Parsing (P3)

Syntax tree

Analysis (P4)

Checked AST + Symtables

Optimized Linear IR

Optimization Passes

IR Code Gen (P5)

Linear IR

Current focus

Machine code

Machine Code Gen
Instruction Scheduling

- Modern architectures expose many opportunities for optimization
  - Some instructions require fewer cycles
  - **Superscalar** processing (multiple functional units)
  - Instruction pipelining
  - Speculative execution

- Primary obstacle: **data dependencies**
  - A **stall** is a delay caused by having to wait for an operand to load

- **Scheduling**: re-order instructions to improve performance
  - Maximize utilization and prevent stalls
  - Must not modify program semantics
  - Main algorithm: **list scheduling**

[Diagram of instruction scheduling](https://en.wikipedia.org/wiki/Superscalar_processor)
Example

- Which program is preferable?
- Assumptions:
  - Loads and stores have a 3-cycle latency
  - Multiplications have a 2-cycle latency
  - All other instructions have a 1-cycle latency

1. `loadAI [BP-4] => r1`
2. `add r1, r1 => r2`
3. `loadAI [BP-8] => r3`
4. `mult r2, r3 => r4`
5. `loadAI [BP-12] => r5`
6. `mult r4, r5 => r6`
7. `loadAI [BP-16] => r7`
8. `mult r6, r7 => r8`
9. `store AI r8 => [BP-20]`
10. `loadAI [BP-4] => r1`
11. `loadAI [BP-8] => r3`
12. `loadAI [BP-12] => r5`
13. `add r1, r1 => r2`
14. `mult r2, r3 => r4`
15. `loadAI [BP-16] => r7`
16. `mult r4, r5 => r6`
17. `mult r6, r7 => r8`
18. `store AI r8 => [BP-20]`
Schedules

- A **schedule** is a list of instructions in **start/issue** order
  - Sometimes with “idle” cycles (no new instructions) marked with “-”
  - Example: “a, b, -, c, -, -, -” means “*start instruction a on cycle one, b on cycle two, nothing on cycle three, c on cycle four, and then wait two more cycles for everything to finish*”

1. a) loadAI [BP-4] => r1
2.  
4. b) add r1, r1 => r2
5.  
8. c) loadAI [BP-8] => r3
9.  
12. d) mult r2, r3 => r4
13.  
16. e) loadAI [BP-12] => r5
18.  
18. f) mult r4, r5 => r6
18.  
18. g) loadAI [BP-16] => r7
18.  
18. h) mult r6, r7 => r8
18.  
18. i) storeAI r8 => [BP-20]

*a, -, -, b, c, -, -, d, e, -, -, f, g, -, -, h, -, i, -,-*  

1. a) loadAI [BP-4] => r1
2.  
4. c) loadAI [BP-8] => r3
5.  
8. e) loadAI [BP-12] => r5
9.  
12. b) add r1, r1 => r2
13.  
16. d) mult r2, r3 => r4
18.  
18. g) loadAI [BP-16] => r7
18.  
18. f) mult r4, r5 => r6
18.  
18. h) mult r6, r7 => r8
18.  
18. i) storeAI r8 => [BP-20]

*a, c, e, b, d, g, f, -, h, -, i, -,-*
Data Dependence

- **Data dependency** \((x = _; _ = x)\)
  - Read after write
  - Hard constraint
- **Antidependency** \((_ = x; x = _)\)
  - Write after read (not generally present in SSA form)
  - Can rename second “x” to avoid (could require more register spills)
- **Dependency graph**
  - One for each basic block
    - Could have multiple roots; technically a forest of directed acyclic graphs (DAGs)
  - Nodes for each instruction
  - Edges represent data dependencies
    - Edge \((n_1, n_2)\) means that \(n_1\) must be done when \(n_2\) runs
List Scheduling

• Prep work
  − Rename to avoid antidependencies
  − Build data dependence graph
  − Assign priority for each instruction
    • Usually based on node height and instruction latency
      − Priority of a leaf node is its latency
      − Priority of a branch node is its latency plus the maximum priority of any immediate successor
    • Goal: prioritize instructions on the critical path
      − Longest-latency path through the graph
List Scheduling

- Track a set of "ready" instructions
  - No remaining unresolved data dependencies; i.e., can be scheduled
- For each cycle:
  - Check all currently executing instructions for any that have finished
    - Add any new "ready" dependents to set
  - Start executing a new "ready" instruction (if there are any)
    - Greedy algorithm: if multiple instructions are ready, choose the one with the highest priority
    - Helps to note the cycle where the instruction will finish
• Schedule the following code:
  - Loads and stores have a 3-cycle latency
  - Multiplications have a 2-cycle latency
  - All other instructions have a 1-cycle latency

```
a) loadAI [BP-4] => r2
b) storeAI r2 => [BP-8]
c) loadAI [BP-12] => r3
d) add r3, r4 => r3
e) add r3, r2 => r3
f) storeAI r3 => [BP-16]
g) storeAI r7 => [BP-20]
```

Original schedule: 
a, -, -, b, c, -, -, d, e, f, g, -, -
(13 cycles)

New schedule:
c, a, g, d, e, b, f, -, -
(9 cycles)
Example

- a) loadAI [BP-4] => r2
- b) storeAI r2 => [BP-8]
- c) loadAI [BP-12] => r3
- d) add r3, r4 => r3
- e) add r3, r2 => r3
- f) storeAI r3 => [BP-16]
- g) storeAI r7 => [BP-20]

Original schedule:
a, - , - , b, c, - , - , d, e, f, g, - , -
(13 cycles)

New schedule:
c, a, g, d, e, b, f, - , -
(9 cycles)
Instruction Priorities

• Usually based on node height and latency first
  – Minimizes critical path

• Many methods for tie-breaking
  – Node's rank (# of successors; breadth-first search)
  – Node's descendant count
  – Latency (maximize resource efficiency)
  – Resource ordering (maximize resource efficiency)
  – Source code ordering (minimize reordering)
  – No clear winner here!
Tradeoffs

• **Forward vs. backward** list scheduling
  - Backward scheduling: build schedule in reverse
    • Choose last instruction on critical path first
    • Schedule from roots to leaves instead of leaves to roots
    • Similar to backward data flow analysis
  - List scheduling is cheap; just run several variants to see which works better for particular code segments
Tradeoffs

- Instruction scheduling vs. register allocation
  - Fewer registers → more sequential code
  - More registers → more possibilities for parallelism
  - Scheduling can also impact number of spills/loads

```
[54x519]Tradeoffs
[48x435]● Instruction scheduling vs. register allocation
  – Fewer registers → more sequential code
  – More registers → more possibilities for parallelism
  – Scheduling can also impact number of spills/loads

Fewer registers required (2)
More sequential (max latency = 6)

More registers required (3)
Less sequential (max latency = 5)
```
Regional scheduling

- Usually based on local list scheduling
- Extended using various techniques
  - Analyze extended basic blocks (chains of basic blocks)
  - Detect hot traces or paths using profile information
  - Sometimes need to insert compensation code
  - Sometimes need to clone entire blocks
- Particularly important for loops
  - Focus on core kernel of the loop
  - Constrained by loop-carried dependencies
• Schedule this program from earlier

• Assumptions:
  – Loads and stores have a 3-cycle latency
  – Multiplications have a 2-cycle latency
  – All other instructions have a 1-cycle latency

a) loadAI [BP-4] => r1
b) add r1, r1 => r2
c) loadAI [BP-8] => r3
d) mult r2, r3 => r4  
e) loadAI [BP-12] => r5
f) mult r4, r5 => r6
g) loadAI [BP-16] => r7
h) mult r6, r7 => r8
i) storeAI r8 => [BP-20]
Exercise

- Schedule this program from earlier
- Assumptions:
  - Loads and stores have a 3-cycle latency
  - Multiplications have a 2-cycle latency
  - All other instructions have a 1-cycle latency

a) loadAI [BP-4] => r1
b) add r1, r1 => r2
c) loadAI [BP-8] => r3
d) mult r2, r3 => r4
e) loadAI [BP-12] => r5
f) mult r4, r5 => r6
g) loadAI [BP-16] => r7
h) mult r6, r7 => r8
i) storeAI r8 => [BP-20]