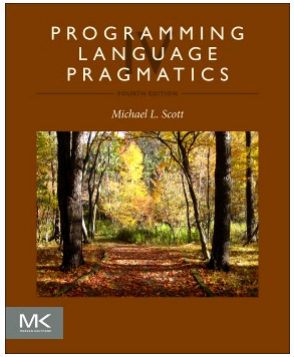
Code Optimization

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 17

Jonathan Aldrich

Machine-Dependent Optimization

- We've covered a few machine-independent optimizations
  - Dead-code elimination, common subexpression elimination, value numbering
- These (almost) always pay off, regardless of the machine
- Machine-Dependent optimizations take into account machine resources and reorganize or reschedule code to make best use of them

Machine-Dependent Optimization

- Machine-specific code optimizations include instruction scheduling and register allocation as well as peephole optimizations
  - Preliminary and final instruction scheduling are essentially identical (Phases 1 & 3)
  - Register allocation (Phase 2) and instruction scheduling tend to interfere with one another
    - the instruction schedules minimize pipeline stalls which tend to increase the demand for architectural registers (*register pressure*)
    - we schedule instructions first, then allocate architectural registers, then schedule instructions again
      - If it turns out that there aren't enough architectural registers, the register allocator will generate additional load and store instructions to *spill* registers temporarily to memory
      - the second round of instruction scheduling attempts to fill any delays induced by the extra loads

Scheduling

- Execution of any instruction takes machine resources (e.g. an arithmetic-logic unit) and time
  - Hardware resources: execution units (ports)
  - Time: pipeline stages such as fetch, decode, execute
- Instruction scheduling makes best use of machine resources to make the *same instructions* run in less time
  - Local instruction scheduling: within a basic block
  - Rearrange instructions to avoid pipeline stalls, within the constraints of data dependencies

Instruction Scheduling Algorithms

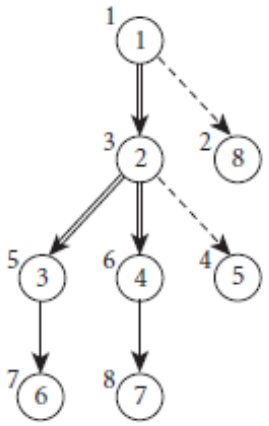
- Considerations:
  - Dependencies between instructions
  - Every instruction takes multiple cycles
  - Resource (machine units) usage of each instruction
- Most algorithms are based on list scheduling
  - Model each cycle of execution, assign instruction(s) to cycles in a forward or backward pass
  - An instruction is ready as soon as its input operands are ready, accounting for the delay of inputs
  - An instruction may not be able to be scheduled at a particular cycle because a unit it needs is in use by another instruction
  - Use heuristic to pick from ready instructions
    - e.g. longest latency-weighted path to return value, number of successors, number of descendants, latency

Instruction Scheduling

- To schedule instructions to make better use of the pipeline, we first arrange them into a directed acyclic graph (DAG), in which each node represents an instruction, and each arc represents a *dependence*
  - Most arcs will represent *flow* dependences, in which one instruction uses a value produced by a previous instruction
  - A few will represent *anti*-dependences, in which a later instruction overwrites a value read by a previous instruction
  - In our example, these will correspond to updates of induction variables

Instruction Scheduling

Block 2:	Scheduled:
1. v19 := v13 × v18	v19 := v13 × v18
—	v18 := v18 – 1
—	—
—	—
—	—
2. v13 := v19 div v17	v13 := v19 div v17
—	v17 := v17 + 1
—	—
—	—
—	—
3. *v26 := v13	*v26 := v13
4. *v34 := v13	*v34 := v13
5. v17 := v17 + 1	v26 := v26 + 4
6. v26 := v26 + 4	v34 := v34 – 4
7. v34 := v34 – 4	
8. v18 := v18 – 1	
–– fall through to Block 3	



Block 3: (same)

v43 := v17 ≤ v42
if v43 goto Block 2
–– else fall through to Block 4

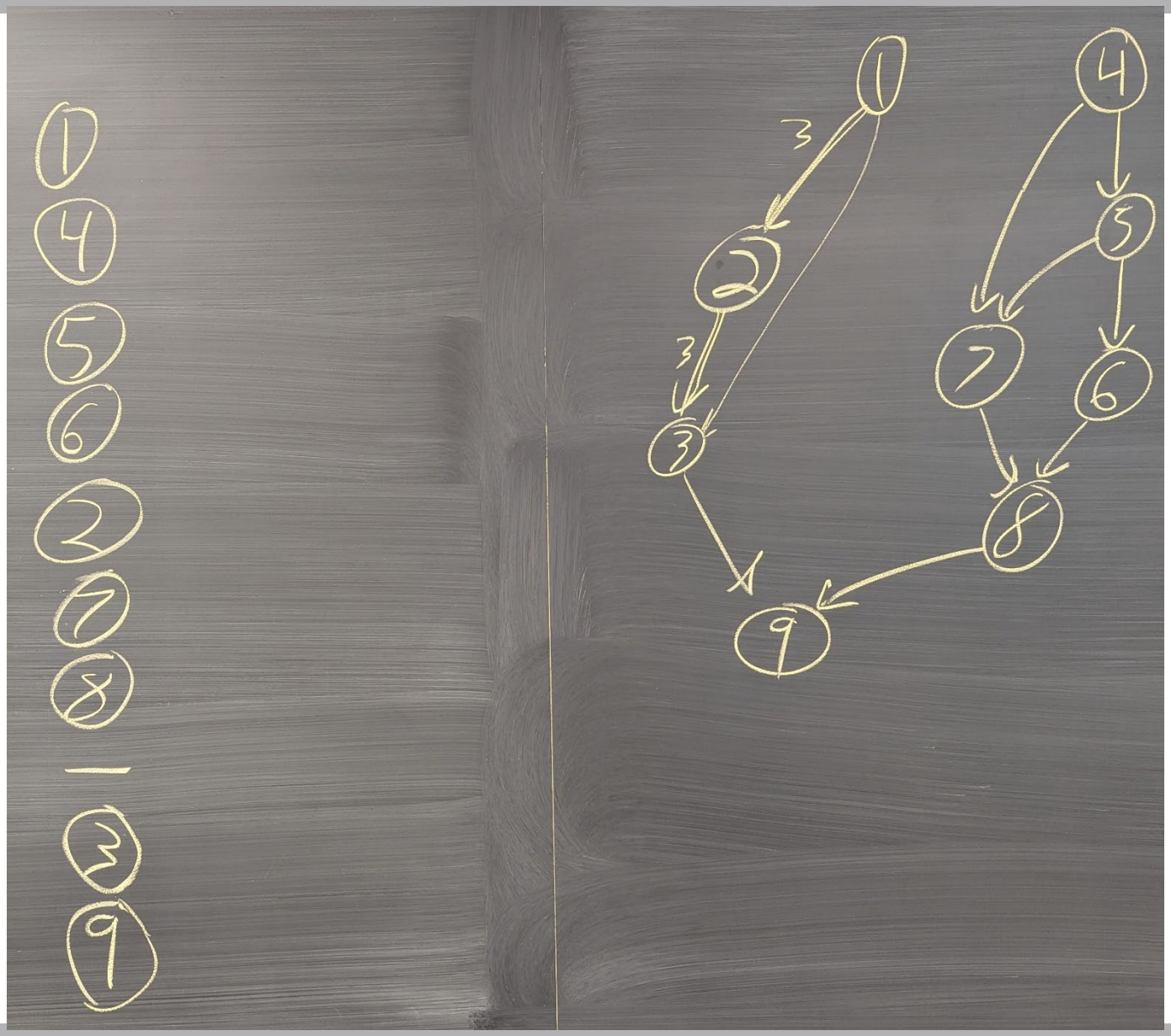
Figure 17.13 **Dependence DAG for Block 2 of Figure C-17.12, together with pseudocode for the entire loop, both before (left) and after (right) instruction scheduling.** Circled numbers in the DAG correspond to instructions in the original version of the loop. Smaller adjacent numbers give the schedule order in the new loop. Solid arcs indicate flow dependences; dashed arcs indicate anti-dependences. Double arcs indicate pairs of instructions that must be separated by four additional instructions in order to avoid pipeline delays on our hypothetical machine. Delays are shown explicitly in Block 2. Unless we modify the array indexing code (Exercise C-17.20), only two instructions can be moved.

Instruction Scheduling Exercise

Assume the following program is going to be run on a platform where multiplication and division take 4 cycles and all other operations take one cycle. How many cycles will this take as written? Perform list scheduling and show the results. How many cycles will it take after optimization?

```
a := x * 37
b := a / 3
c := a + b
i := y >> 2
h := i - 1
g := h << 2
f := h + i
e := f - g
d := c + e
```

Instruction Scheduling Solution



Instruction Scheduling Exercise

Assume the following program is going to be run on a platform where multiplication and division take 4 cycles and all other operations take one cycle. How many cycles will this take as written? **A: 15**

Perform list scheduling and show the results. How many cycles will it take after optimization? **A: 10**

1) a := x * 37
   (3 stalls)
2) b := a / 3
   (3 stalls)
3) c := a + b
4) i := y >> 2
5) h := i - 1
6) g := h << 2
7) f := h + i
8) e := f - g
9) d := c + e

Register Allocation

- In a simple compiler with no global optimizations, register allocation can be performed independently in every basic block
  - simple compilers usually apply a set of heuristics to identify such frequently accessed variables variables and allocate them to registers over the life of a subroutine.
  - candidates for a dedicated register include loop indices, the implicit pointers of with statements in Pascal-family languages, and scalar local variables and parameters
- It has been known since the early 1970s that register allocation is equivalent to the NP-hard problem of graph coloring

Register Allocation

- Program IR has an unlimited number of variable names, often called “virtual registers”
- Real machines have a fixed set, often 8, 16, or 32 of *physical* or *architectural* registers
- May be different sets for float and integer
- Registers are the fastest storage of the entire computer system => important to use effectively
- How do we map the unbounded set of virtual registers onto a finite set?
  - Answer: cannot, in general, must use some stack
  - When to use stack, and which register to use?
    - Program analysis!

Applying Liveness Analysis to Register Allocation

- Observation: liveness analysis computes the set of live variables at the beginning and end of every basic block
- What we need: set of live variables at *every* program point
- Variables that are not live at any of the same program points can use the same register
- Otherwise we say they *interfere*
- Computing the set of interferences allows us to allocate registers to variables efficiently

Approaches to Register Allocation

- Local: only use registers within a block
- Linear scan: simplify control flow graph to a line, obtain a set of intervals that are relatively easy to assign to registers (almost same as local)
- Graph coloring: use *interference graph* directly to compute a more efficient solution
  - Interference graph represents all conflicts between variables that cannot use the same register
  - Graph-coloring is an NP-hard problem
  - Nearly all compilers use a *coloring heuristic* for the graph
- In any case, if allocation fails, resort to *spilling*

Approaches to Register Allocation

- Chaitin's graph coloring algorithm: use *interference graph* directly to compute a more efficient solution
  - Build interference graph from liveness analysis
  - Nodes represent variables
  - (Undirected edges represent interference)
  - Assign one of K colors to each variable
    - ensure that adjacent variables don't have the same color
  - Simplification-based heuristic: remove nodes < K degree onto a stack
  - Pop nodes in reverse order, rebuilding the graph and giving each a color that differs from its neighbors
  - Spill if not enough colors
    - example heuristic: spill max (degree/(defs+uses))

Coloring Algorithm Pseudocode

```
color(g, r)
    let stack = empty
    while true do
        while exists node n in g of degree < r
            remove n from g
            push n on to stack
        if g = empty
            while s is nonempty
                pop n from s
                add n to g
                assign n a color that doesn't conflict with its neighbors
            break
        else
            select a node n in g according to a heuristic: max (degree/(defs+uses))
            remove n from g
```

Register Allocation Exercise

Apply Chaitin's register allocation and spilling algorithm, using graph coloring to allocate 4 registers. Give your answer by assigning a register number 1-4 to each of the variables above, and assign no register if the variable is spilled to memory.

Register Allocation Exercise Solution