Lecture 7
Instruction Scheduling

I. Basic Block Scheduling
II. Global Scheduling
   (for Non-Numeric Code)

Reading: Chapter 10.3 – 10.4
Scheduling Constraints

• **Data dependences**
  – The operations must generate the *same results* as the corresponding ones in the original program.

• **Control dependences**
  – All the operations executed in the original program *must be executed* in the optimized program.

• **Resource constraints**
  – No over-subscription of resources.

Same constraints for instruction level / processor level parallelism
Data Dependence

• **Must maintain order of accesses to potentially same locations**
  - **True dependence**: write -> read (RAW hazard)
    \[
    a = \ldots = a
    \]
  - **Output dependence**: write -> write (WAW hazard)
    \[
    a = \ldots
    a = \ldots
    \]
  - **Anti-dependence**: read -> write (WAR hazard)
    \[
    = a
    a = \ldots
    \]

Quiz: What is missing?

• **Data Dependence Graph**
  - **Nodes**: operations
  - **Edges**: \( n_1 \) -> \( n_2 \) if \( n_2 \) is data dependent on \( n_1 \)
    - labeled by the execution length of \( n_1 \)
Analysis on Memory Variables

• **Undecidable in general**
  
  \[
  \text{read } x \\
  \text{read } y \\
  A[x] = \ldots \\
  \ldots = A[y]
  \]

• Two memory accesses can potentially be the same unless proven otherwise

• Classes of analysis:
  
  – **simple**: \(\text{base} + \text{offset1} = \text{base} + \text{offset2}\) ?
  
  – “data dependence analysis”:
    
    Array accesses whose indices are affine expressions of loop indices \(A[2i] = A[2i+1]\) ?
  
  – **interprocedural analysis**: \(\text{global} = \text{parameter}\) ?
  
  – **pointer analysis**: \(\text{pointer1} = \text{pointer2}\) ?

• Data dependence analysis is useful for many other purposes
Resource Constraints

- Each instruction type has a **resource reservation table**

<table>
<thead>
<tr>
<th>Functional units</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

- Pipelined functional units: occupy only one slot
- Non-pipelined functional units: multiple time slots
- Instructions may use more than one resource
- A resource type may have multiple units
- Limited instruction issue slots
  - may also be managed like a resource
Example of a Machine Model

- Each machine cycle can execute 2 operations

  - 1 ALU operation or branch operation
    
    \[ \text{Op} \text{ dst, src1, src2} \] executes in 1 clock

  - 1 load or store operation
    
    \[ \text{LD} \text{ dst, addr} \] result is available in 2 clocks
    pipelined: can issue LD next clock

    \[ \text{ST} \text{ src, addr} \] executes in 1 clock cycle
    - can issue LD/ST to any location in the next clock
    - a write buffer is often used to hold the result
Basic Block Scheduling Example

LD R2 <- 0(R1)
ST 4(R1) <- R2
LD R3 <- 8(R1)
ADD R3 <- R3,R4
ADD R3 <- R3,R2
ST 12(R1) <- R3
ST 0(R7) <- R7

Quiz: Shown are the register data dependence constraints. What are the memory data dependence constraints?
With Resource Constraints

- NP-complete in general \(\rightarrow\) Heuristics time!

- **List Scheduling:**
  
  READY = nodes with 0 predecessors

  Loop until READY is empty {
    
    Let \(n\) be the node in READY with highest priority

    Schedule \(n\) in the earliest slot
    that satisfies precedence + resource constraints

    Update predecessor count of \(n\)'s successor nodes
    Update READY

  }

  M. Lam
List Scheduling for Basic Blocks

• **Scope:** DAGs
  - Schedules operations in topological order
  - Never backtracks

• **Variations:**
  - Priority function for node $n$
    - critical path: max clocks from $n$ to any node
    - resource requirements
    - source order

Quiz: Why use the simple list scheduling algorithm for basic blocks?

Quiz: How many operations are in a basic block?

Quiz: How much parallelism is in a basic block?
II. Introduction to Global Scheduling

Assume each clock can execute 2 operations of any kind & no aliases

if (a==0) goto L

c = b

L:
e = d + d

LD R6 <- 0(R1)
nop
BEQZ R6, L

LD R8 <- 0(R4)
nop
ADD R8 <- R8,R8
ST 0(R5) <- R8

LD R7 <- 0(R2)
nop
ST 0(R3) <- R7

Quiz: Is there parallelism in this program? How do you schedule it?
Result of Code Scheduling

B₁

i₁
LD R6 ← 0(R1)
nop
BEQZ R6, L

i₂
LD R8 ← 0(R4)
nop

B₂

i₃
ADD R8 ← R8,R8

i₄
LD R7 ← 0(R2)
nop
ST 0(R3) ← R7

B₃

i₅
LD R8 ← 0(R4)
nop
ADD R8 ← R8,R8
ST 0(R5) ← R8

i₆
LD R6 ← 0(R1)
nop
BEQZ R6, L

i₇
LD R7 ← 0(R2)

i₈
ADD R8 ← R8,R8

i₉
ST 0(R5) ← R8

i₁₀
ST 0(R3) ← R7

L:

ST 0(R5) ← R8

B₃'}

L:

ST 0(R5) ← R8 ; ST 0(R3) ← R7
Lessons from the Example

- Basic blocks are small, lots of dependences

- Global scheduling (across basic blocks) is necessary
  - Lots of dependences esp. with memory operations (esp. due to aliases)

- Static schedulers can look ahead & prioritize over dynamic schedulers.
Control Dependence Constraints

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.
- Requirement:
  - Raises no exception,
  - Satisfies data dependences
Scheduling Algorithms

• Many algorithms have been proposed; they look very different

• To analyze them, we separate the discussion:
  – What are the legal code motions?
    • universal: a small possible number of basic moves
  – Which ones should we use?
    • dependent on the designer's assessment of program and machine characteristics
Legal Code Motions

Goal: Shorten execution time *probabilistically* while honoring data dependences.

Moving instructions **up**:
- Move instruction to a cut set (from entry)
- Speculation: move to where it is not anticipated.

Moving instructions **down**:
- Move instruction to a cut set (from exit)
- May execute extra instruction
- Can duplicate code
A Note on Data Dependences

- All code motions must satisfy data dependence constraints

Quiz: Can we move these operations up?

Data dependences must be updated as code is moved.
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
- Only one level of speculation
Program Representation

- 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 **function** is represented as a **hierarchy of regions**
  - The whole control flow graph is a region
  - In a structured program:
    - Each loop in the flow graph is a region
    - 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{$B$} $\cup$
        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
                }
            }
        }
    }
}
Node $d$ dominates node $n$ in a graph ($d$ dom $n$):
    – if every path from the start node to $n$ goes through $d$ (a node dominates itself)
*Priority functions*: non-speculative before speculative
Extensions

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

• List scheduling
  – Greedy algorithm with no backtracking
  – Topological sort with a priority function to choose among candidates

• Global scheduling
  – Limited opportunity of code motion within a single basic block
  – Legal code motions: explain with cut sets
    • Speculative execution is necessary to find enough parallelism
  – Simple heuristics to move code up from select basic blocks

• General lessons
  – Use program characteristics to guide the effort tradeoff