Lecture 9

Instruction Scheduling

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

Reading: Chapter 10.3 – 10.4
Who Schedules

- Compiler
- Assembler
- Hardware
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.
Data Dependence

- **Must maintain order of accesses to potentially same locations**
  - **True dependence:** write -> read (RAW hazard)
    
    ```
    a = ...
    = a
    ```
  
  - **Output dependence:** write -> write (WAW hazard)
    
    ```
    a = ...
    a = ...
    ```
  
  - **Anti-dependence:** read -> write (WAR hazard)
    
    ```
    = a
    a = ...
    ```

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

- **Undecidable in general**
  
  read x; read y;
  
  A[x] = ...
  
  ... = A[y]

- **Two memory accesses can potentially be the same unless proven otherwise**

- **Classes of analysis:**
  
  - simple: base+offset1 = base+offset2 ?
  
  - “data dependence analysis”:
    
    - Array accesses whose indices are affine expressions of loop indices
      

  - interprocedural analysis: global = parameter?

  - pointer analysis: pointer1 = pointer2?

  - language rules:
    
    - int *a; float *b; *a=...; *b=...
    
    - int *restrict p;

- Data dependence analysis is useful for many other purposes
Aside

• Can these be reordered

   for i

    LD R2 ← 0(R1)
    ADDI R2 ← R2, 1
    ST (R1) ← R2
    ADDI R1 ← R1, 4

    LD R3 ← 0(R1)
    ADDI R3 ← R3, 1
    ST (R1) ← R3
    ADDI R1 ← R1, 4
Resource Constraints

- Each instruction type has a resource reservation table

Functional units

<table>
<thead>
<tr>
<th></th>
<th>ld</th>
<th>st</th>
<th>alu</th>
<th>fmpry</th>
<th>fadd</th>
<th>br</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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
- Multiple units of same resource
- 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

  \[ \text{Op } \text{dst,src1,src2} \text{ executes in 1 clock} \]

• 1 load or store operation

  \[ \text{LD dst, addr } \text{ result is available in } 2 \text{ clocks} \]

  pipelined: can issue LD next clock

  \[ \text{ST src, addr } \text{ executes in 1 clock cycle} \]
Basic Block Scheduling

LD R2 <- 0(R1)

i1

ST 4(R1) <- R2

i2

LD R3 <- 8(R1)

i3

ADD R3 <- R3,R4

i4

ADD R3 <- R3,R2

i5

ST 12(R1) <- R3

i6

ST 0(R7) <- R7

i7

alu mem
With Resource Constraints

• NP-complete in general \( \Rightarrow \) Heuristics time!
• List Scheduling:

\[
\text{READY} = \text{nodes with 0 predecessors}
\]

Loop until READY is empty {

Let \( n \) be the node in READY with \textbf{highest priority}

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

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

}
List Scheduling

• **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
II. Introduction to Global Scheduling

Assume each clock can execute 2 operations of any kind.

```c
if (a==0) goto L

c = b

L: e = d + d

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

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

LD R7 <- 0(R2)
stall
ST 0(R3) <- R7
```
Result of Code Scheduling

LD R6 <- 0(R1) ; LD R8 <- 0(R4)
LD R7 <- 0(R2)
ADD R8 <- R8,R8 ; BEQZ R6, L

L:

B₁
ST 0(R5) <- R8

B₃
ST 0(R5) <- R8 ; ST 0(R3) <- R7

B₃′
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.
- Requirement: Raises no 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
A Note on Updating Data Dependences

```
a = 0
```

```
a = 1
```
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 either:
  - a reducible loop,
  - the entire function

- A **function** is represented as a **hierarchy of regions**
  - The whole control flow graph is a region
  - Each natural loop 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;
        }
    }
    Update CandInsts;
}}

Priority functions: non-speculative before speculative
Alternative: Hyperblock Scheduling

- Hyperblock: A set of basic blocks with a single entry
Hyperblock Scheduling

• Use a heuristic to select blocks
• Tail duplication to ensure a single entry
• Node Splitting

• If conversion
• Promotion (speculation)
• Instruction Merging (PRE)
• Scheduling
Basic Algorithm Versus Hyperblock

• Basic algorithm designed for a machine with very limited parallelism
  – Hyperblock designed for a machine that has lots of parallelism
• Hyperblock assumes predicated instruction set
• Neither designed for numerical/media code nor for dynamically scheduled machines