Topic 2b
Basic Back-End Optimization

Instruction Selection
Instruction scheduling
Register allocation
ABET Outcome

- Ability to apply knowledge of basic code generation techniques, e.g. Instruction scheduling, register allocation, to solve code generation problems.
- An ability to identify, formulate and solve loops scheduling problems using software pipelining techniques
- Ability to analyze the basic algorithms on the above techniques and conduct experiments to show their effectiveness.
- Ability to use a modern compiler development platform and tools for the practice of above.
- A Knowledge on contemporary issues on this topic.
Reading List

(1) K. D. Cooper & L. Torczon, Engineering a Compiler, Chapter 12
(2) Dragon Book, Chapter 10.1 ~ 10.4
A Short Tour on
Data Dependence
Basic Concept and Motivation

- Data dependence between 2 accesses
  - The same memory location
  - Exist an execution path between them
  - At least one of them is a write

- Three types of data dependencies

- Dependence graphs

- Things are not simple when dealing with loops
Data Dependencies

There is a data dependence between statements $S_i$ and $S_j$ if and only if

- Both statements access the same memory location and at least one of the statements writes into it, and
- There is a feasible run-time execution path from $S_i$ to $S_j$
Types of Data Dependencies

- **Flow (true) Dependencies** - write/read ($\delta$)
  
  ```
  x := 4;
  ...
  y := x + 1;
  ```

- **Output Dependencies** - write/write ($\delta^0$)
  
  ```
  x := 4;
  ...
  x := y + 1;
  ```

- **Anti-dependencies** - read/write ($\delta^{-1}$)
  
  ```
  y := x + 1;
  ...
  x := 4;
  ```
An Example of Data Dependencies

(1) \( x := 4 \)
(2) \( y := 6 \)
(3) \( p := x + 2 \)
(4) \( z := y + p \)
(5) \( x := z \)
(6) \( y := p \)

Flow
Output
Anti
Data Dependence Graph (DDG)

- Forms a data dependence graph between statements
  - nodes = statements
  - edges = dependence relation (type label)
Reordering Transformations using DDG

Given a correct data dependence graph, any order-based optimization that does not change the dependences of a program is guaranteed not to change the results of the program.
A *reordering transformation* is any program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statements.

A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.
Reordering Transformations (Con’t)

- Instruction Scheduling
- Loop restructuring
- Exploiting Parallelism

- Analyze array references to determine whether two iterations access the same memory location. Iterations I1 and I2 can be safely executed in parallel if there is no data dependency between them.

- ...
Example 1:

S1: $A = 0$
S2: $B = A$
S3: $C = A + D$
S4: $D = 2$

$S_x \delta S_y \Rightarrow \text{flow dependence}$
Example 2:

S1: A = 0
S2: B = A
S3: A = B + 1
S4: C = A
Should we consider input dependence?

= X

Is the reading of the same X important?

= X

Well, it may be!
(if we intend to group the 2 reads together for cache optimization!)
Applications of Data Dependence Graph

- register allocation
- instruction scheduling
- loop scheduling
- vectorization
- parallelization
- memory hierarchy optimization
Data Dependence in Loops

Problem: How to extend the concept to loops?

(s1) \[\text{do } i = 1,5\]
(s2) \[x := a + 1;\] \[s2 \quad \delta^{-1} s3, \quad s2 \quad \delta s3\]
(s3) \[a := x - 2;\] \[s3 \quad \delta s2 \quad \text{(next iteration)}\]
(s4) \[\text{end do}\]
Modern processors can overlap the execution of multiple *independent* instructions through pipelining and multiple functional units. Instruction scheduling can improve the performance of a program by placing independent target instructions in parallel or adjacent positions.
Instruction scheduling (con’t)

- Assume all instructions are essential, i.e., we have finished optimizing the IR.
- Instruction scheduling attempts to reorder the codes for maximum instruction-level parallelism (ILP).
- It is one of the instruction-level optimizations.
- Instruction scheduling (IS) in general is NP-complete, so heuristics must be used.
Instruction scheduling: A Simple Example

Since all three instructions are independent, we can execute them in parallel, assuming adequate hardware processing resources.
Hardware Parallelism

Three forms of parallelism are found in modern hardware:

- pipelining
- superscalar processing
- VLIW
- multiprocessing

Of these, the first three forms are commonly exploited by today’s compilers’ instruction scheduling phase.
Pipelining & Superscalar Processing

Pipelining
Decompose an instruction’s execution into a sequence of stages, so that multiple instruction executions can be overlapped. It has the same principle as the assembly line.

Superscalar Processing
Multiple instructions proceed simultaneously assisted by hardware dynamic scheduling mechanism. This is accomplished by adding more hardware, for parallel execution of stages and for dispatching instructions to them.
A Classic Five-Stage Pipeline

- instruction fetch
- decode and register fetch
- execute on ALU
- memory access
- write back to register file

IF  RF  EX  ME  WB

time
The standard non-pipelined model

In a given cycle, each instruction is in a different stage, but every stage is active

The pipeline is “full” here
Parallelism in a pipeline

Example:

\begin{align*}
  i1 &: \text{add } r1, \quad r1, \quad r2 \\
  i2 &: \text{add } r3, \quad r3, \quad r1 \\
  i3 &: \text{lw } r4, \quad 0(r1) \\
  i4 &: \text{add } r5, \quad r3, \quad r4
\end{align*}

Consider two possible instruction schedules (permutations):

Schedule S1 (completion time = 6 cycles):

\begin{align*}
  \begin{array}{cccc}
    i1 & i2 & i3 & i4 \\
    & & & \\
  \end{array}
\end{align*}

2 Idle Cycle

Schedule S2 (completion time = 5 cycles):

\begin{align*}
  \begin{array}{cccc}
    i1 & i3 & i2 & i4 \\
    & & & \\
  \end{array}
\end{align*}

1 Idle Cycle

Assume:

Register instruction 1 cycle
Memory instruction 3 cycle
Superscalar Illustration

Multiple instructions in the same pipeline stage at the same time
Parallelism Constraints

**Data-dependence constraints**
If instruction A computes a value that is read by instruction B, then B can’t execute before A is completed.

**Resource hazards**
Finiteness of hardware function units means limited parallelism.
Scheduling Complications

- Hardware Resources
  - finite set of FUs with instruction type, and width, and latency constraints
  - cache hierarchy also has many constraints

- Data Dependences
  - can’t consume a result before it is produced
  - ambiguous dependences create many challenges

- Control Dependences
  - impractical to schedule for all possible paths
  - choosing an “expected” path may be difficult
  - recovery costs can be non-trivial if you are wrong
Legality Constraint for Instruction Scheduling

**Question:** when must we preserve the order of two instructions, \( i \) and \( j \)?

**Answer:** when there is a dependence from \( i \) to \( j \).
Construct a DDG by assigning weights to nodes and edges in the DDG to model the pipeline as follows:

- Each DDG node is labeled a resource-reservation table whose value is the resource-reservation table associated with the operation type of this node.
- Each edge $e$ from node $j$ to node $k$ is labeled with a weight (latency or delay) $d_e$ indicting that the destination node $j$ must be issued no earlier than $d_e$ cycles after the source node $k$ is issued.
Example of a Weighted Data Dependence Graph

\[i1: \text{add } r1, r1, r2\]
\[i2: \text{add } r3, r3, r1\]
\[i3: \text{lw } r4, (r1)\]
\[i4: \text{add } r5, r3, r4\]

Assume:
Register instruction 1 cycle
Memory instruction 3 cycle
Legal Schedules for Pipeline

Consider a basic block with \( m \) instructions, 
\[ i_1, \ldots, i_m. \]

A legal sequence, \( S \), for the basic block on a pipeline consists of:

A permutation \( f \) on \( 1 \ldots m \) such that \( f(j) \) identifies the new position of instruction \( j \) in the basic block. For each DDG edge from \( j \) to \( k \), the schedule must satisfy \( f(j) \leq f(k) \).
Legal Schedules for Pipeline (Con’t)

Instruction start-time

An instruction start-time satisfies the following conditions:

- **Start-time** \( (j) \) \( \geq 0 \) for each instruction \( j \)
- No two instructions have the same **start-time** value
- For each DDG edge from \( j \) to \( k \), 
  \[ \text{start-time}(k) \geq \text{completion-time}(j) \]

where

\[ \text{completion-time}(j) = \text{start-time}(j) + \text{(weight between } j \text{ and } k) \]
Legal Schedules for Pipeline (Con’t)

We also define:

\[
\text{make_span}(S) = \text{completion time of schedule } S \\
= \text{MAX (} \{ \text{completion-time (}j\text{)} \} \text{)} \\
1 \leq j \leq m
\]
Example of Legal Schedules

Assume:
- Register instruction 1 cycle
- Memory instruction 3 cycle

**Schedule S1** (completion time = 6 cycles):

- \( i1: \text{add } r1, r1, r2 \)
- \( i2: \text{add } r3, r3, r1 \)
- \( i3: \text{lw } r4, (r1) \)
- \( i4: \text{add } r5, r3, r4 \)


\[ \begin{array}{cccc}
0 & 1 & 2 & 5 \\
\hline
i1 & i2 & i3 & i4 \\
\end{array} \]

**Start-time**

- 2 Idle Cycle

**Schedule S2** (completion time = 5 cycles):

- \( i1: \text{add } r1, r1, r2 \)
- \( i2: \text{add } r3, r3, r1 \)
- \( i3: \text{lw } r4, (r1) \)
- \( i4: \text{add } r5, r3, r4 \)

\[ \begin{array}{cccc}
0 & 1 & 2 & 4 \\
\hline
i1 & i2 & i3 & i4 \\
\end{array} \]

**Start-time**

- 1 Idle Cycle
Instruction Scheduling (Simplified)

Problem Statement:

Given an acyclic weighted data dependence graph G with:

- Directed edges: precedence
- Undirected edges: resource constraints

Determine a schedule S such that the length of the schedule is minimized!
Simplify Resource Constraints

Assume a machine M with \( n \) functional units or a “clean” pipeline with \( k \) stages.

What is the complexity of a optimal scheduling algorithm under such constraints?

Scheduling of M is still hard!

- \( n = 2 \) : exists a polynomial time algorithm \([CoffmanGraham]\)
- \( n = 3 \) : remain open, conjecture: NP-hard
General Approaches of Instruction Scheduling

- List Scheduling
- Trace scheduling
- Software pipelining
- ......
Trace Scheduling

A technique for scheduling instructions across basic blocks.

\*The Basic Idea of trace scheduling

- Uses information about actual program behaviors to select regions for scheduling.
Software Pipelining

A technique for scheduling instructions across loop iterations.

- **The Basic Idea of software pipelining**
  - Rewrite the loop as a repeating pattern that overlaps instructions from different iterations.
List Scheduling

A most common technique for scheduling instructions within a basic block.

- **The Basic Idea of list scheduling**
  - All instructions are sorted according to some priority function.
  - Also maintain a list of instructions that are ready to execute (i.e. data dependence constraints are satisfied).
  - Moving cycle-by-cycle through the schedule template:
    - Choose instructions from the list & schedule them (provided that machine resources are available)
    - Update the list for the next cycle
List Scheduling (Con’t)

- Uses a greedy heuristic approach
- Has forward and backward forms
- Is the basis for most algorithms that perform scheduling over regions larger than a single block.
Heuristic Solution: Greedy List Scheduling Algorithm

1. Build a priority list $L$ of the instructions in non-decreasing order of some rank function.
2. For each instruction $j$, initialize
   \[ \text{pred-count}[j] := \text{#predecessors of } j \text{ in DDG} \]
3. \textbf{Ready-instructions} := \{ $j$ | $\text{pred-count}[j] = 0$ \}
4. \textbf{While} (ready-instructions is non-empty) \textbf{do} $j$ := first ready instruction according to the order in priority list, $L$; Output $j$ as the next instruction in the schedule;

Consider resource constraints beyond a single clean pipeline
Heuristics Solution: Greedy List Scheduling Algorithm

Con’d

Ready-instructions := ready-instructions- \{ j \};

for each successor k of j in the DDG do
    pred-count[k] := pred-count[k] - 1;
    if (pred-count[k] = 0) then
        ready-instructions := ready-instruction + \{ k \};
    end if
end for
end while
Special Performance Bounds

For a single two stage pipeline,

\[ m = 1 \text{ and } k = 2 \implies \]

(here \( m \) is the number of pipelines, and \( k \) is the number of pipeline stages per pipeline)

\[
\text{makespan (greedy)}/\text{makespan(OPT)} \leq 1.5
\]
Properties of List Scheduling

- Complexity: $O(n^2)$ --- where $n$ is the number of nodes in the DDG
- In practice, it is dominated by DDG building which itself is also $O(n^2)$
- The result is within a factor of two from the optimal for pipelined machines (Lawler87)

Note: we are considering basic block scheduling here
A Heuristic Rank Function Based on Critical paths

1. Compute EST (Earliest Starting Times) for each node in the augmented DDG as follows:

   \[ \text{EST}[\text{START}] = 0 \]

   \[ \text{EST}[y] = \max \{ \text{EST}[x] + \text{node-weight (x)} + \text{edge-weight (x,y)} \mid \text{there exists an edge from x to y} \} \]

2. Set CPL := EST[END], the critical path length of the augmented DDG.

3. Similarly, compute LST (Latest Starting Time of All nodes);

4. Set rank \((i) = \text{LST}[i] - \text{EST}[i]\), for each instruction \(i\)

   \(\text{(all instructions on a critical path will have zero rank)}\)

NOTE: there are other heuristics
Example of Rank Computation

<table>
<thead>
<tr>
<th>Node, x</th>
<th>EST[X]</th>
<th>LST[x]</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>i3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>i4</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>END</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

=> Priority list = (i1, i3, i4, i2)
Summary

Instruction Scheduling for a Basic Block

1. Build the data dependence graph (DDG) for the basic block
   - Node = target instruction
   - Edge = data dependence (flow/anti/output)

2. Assign weights to nodes and edges in the DDG so as to model target processor e.g., for a two-stage pipeline
   - Node weight = 1, for all nodes
   - Edge weight = 1 for edges with load/store instruction as source node; edge weight = 0 for all other edges
Summary

3. A legal schedule for a weighted DDG must obey all ordering and timing constraints of the weighted DDG

4. **Goal**: find a legal schedule with minimum completion time
Other Heuristics for Ranking

- Number of successors?
- Number of total descendents?
- Latency?
- Last use of a variable?
- Others?

Note: these heuristics help break ties, but none dominates the others.
Hazards Preventing Pipelining

- Structural hazards
- Data dependent hazard
- Control hazard
Local vs. Global Scheduling

1. **Straight-line code** (basic block) – Local scheduling

2. **Acyclic control flow** – Global scheduling
   - Trace scheduling
   - Hyperblock/superblock scheduling
   - IGLS (integrated Global and Local Scheduling)

3. **Loops** - a solution for this case is loop unrolling+scheduling, another solution is software pipelining or modulo scheduling i.e., to rewrite the loop as a repeating pattern that overlaps instructions from different iterations.
Smooth Info flow into Backend in Open64
Flowchart of Code Generator

WHIRL

WHIRL-to-TOP Lowering

CGIR: Quad Op List

Control Flow Opt I
EBO

Hyperblock Formation
Critical-Path Reduction

Process Inner Loops: unrolling,
EBO
Loop prep, software pipelining

Control Flow Opt II
EBO

IGLS: pre-pass
GRA, LRA, EBO
IGLS: post-pass
Control Flow Opt

Code Emission

EBO: Extended basic block optimization peephole, etc.
PQS: Predicate Query System

EBO: Extended basic block optimization peephole, etc.
Software Pipelining vs Normal Scheduling

- **Yes**: Inner loop processing software pipelining
  - Failure/not profitable
  - Success: Code Emission

- **No**: a SWP-amenable loop candidate?
  - IGLS
  - GRA/LRA
  - IGLS

2012/3/12