Topic 6a
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
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 ($\delta$)
  
  \[
  \begin{align*}
  x &:= 4; \\
  \quad \vdots & \downarrow \\
  y &:= x + 1; \quad \delta \\
  \end{align*}
  \]

- **Output Dependencies** - write/write ($\delta^0$)
  
  \[
  \begin{align*}
  x &:= 4; \\
  \quad \vdots & \downarrow \delta^0 \\
  x &:= y + 1; \\
  \end{align*}
  \]

- **Anti-dependencies** - read/write ($\delta^{-1}$)
  
  \[
  \begin{align*}
  y &:= x + 1; \\
  \quad \vdots & \downarrow \delta^{-1} \\
  x &:= 4; \quad \delta^{-1} \\
  \end{align*}
  \]
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)
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
- ……
Problem: How to extend the concept to loops?

(s1) do i = 1,5
(s2) \( x := a + 1; \) \( s2 \ \delta^{-1} \ s3 \), \( s2 \ \delta \ s3 \)
(s3) \( a := x - 2; \)
(s4) end do \( s3 \ \delta \ s2 \) (next iteration)
Reordering Transformation

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.

- ...
Reordering Transformation 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.
Instruction Scheduling

Motivation

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) 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
- multiprocessing

Of these, the first two forms are commonly exploited by instruction scheduling.
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 through the same pipeline stages. 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
Pipeline Illustration

The standard Von Neumann 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{center}
\begin{tikzpicture}
\draw[very thick, red] (0,1) -- (1,1) node [midway, above] {2 Idle Cycles};
\end{tikzpicture}
\end{center}

Schedule S2 (completion time = 5 cycles):

\begin{center}
\begin{tikzpicture}
\draw[very thick, red] (0,1) -- (1,1) node [midway, above] {1 Idle Cycle};
\end{tikzpicture}
\end{center}

Assume:

- Register instruction 1 cycle
- Memory instruction 3 cycle
Superscalar Illustration

Multiple instructions in the same pipeline stage at the same time

IF RF EX ME WB

FU1

FU2

FU1

FU2

FU1

FU2

FU1

FU2

2008/4/15
A Quiz

Give the following instructions:

- **i1**: move r1 ← r0
- **i2**: mul r4 ← r2, r3
- **i3**: mul r5 ← r4, r1
- **i4**: add r6 ← r4, r2

Assume `mul` takes 2 cycles, other instructions take 1 cycle.

Schedule the instructions in a “clean” pipeline.

Q1. For above sequence, can the pipeline issue an instruction in each cycle? Why? No. think about i2 and i3.

Q2. Is there a possible instruction scheduling such that the pipeline can issue an instruction in each cycle?

Yes!. There is a schedule:

![Instruction Schedule Diagram]

```plaintext
i2  i1  i3  i4
```

2008/4/15
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

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

Answer: when there is a *dependence* from \( i \) to \( j \).
General Approaches of Instruction Scheduling

- Trace scheduling
- Software pipelining
- List scheduling
- ...
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:

- Maintain a list of instructions that are ready to execute
  - data dependence constraints would be preserved
  - machine resources are available
- Moving cycle-by-cycle through the schedule template:
  - choose instructions from the list & schedule them
  - update the list for the next cycle
- 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.
Construct DDG with Weights

Construct a DDG by assigning weights to nodes and edges in the DDG to model the pipeline/function unit 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 $k$ must be issued no earlier than $d_e$ cycles after the source node $j$ 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 \( f(j) (j = 1, \ldots, m) \) identifies the new position of instruction \( j \) in the basic block. For each DDG edge form \( j \) to \( k \), the schedule must satisfy \( f(j) \leq f(k) \).
Legal Schedules 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 \( k \),
    
    start-time \( (k) \geq \) completion-time \( (j) \)
  
  where
    
    completion-time \( (j) = \) start-time \( (j) \) + \( \) (weight between \( j \) and \( k \))
Legal Schedules Pipeline (Con’t)

- **Schedule length**

  The length of a schedule $S$ is defined as:

  $$L(S) = \text{completion time of schedule } S$$

  $$= \max \{ \text{completion-time (j)} \} \quad \overline{1} \leq j \leq m$$

  The schedule $S$ must have at least one operation $n$ with start-time$(n) = 1$

- **Time-optimal schedule**

  A schedule $S_i$ is time-optimal if $L(S_i) \leq L(S_j)$ for all other schedule $S_j$ that contain the same set of operations.
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!
Assume a machine M with $n$ functional units or a “clean” pipeline with $n$ stages.

What is the complexity of an 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
A Heuristic Rank (priority) Function Based on Critical paths

**Critical Path:** The longest path through the DDG. It determines overall execution time of the instruction sequence represented by this DDG.

1. Attach a dummy node START as the virtual beginning node of the block, and a dummy node END as the virtual terminating node.
2. Compute EST (Earliest Starting Times) for each node in the augmented DDG as follows (this is a forward pass):
   
   \[
   \begin{align*}
   \text{EST}[\text{START}] &= 0 \\
   \text{EST}[y] &= \text{MAX} \left( \{ \text{EST}[x] + \text{edge}_\text{weight}(x, y) \mid \text{there exists an edge from } x \text{ to } y \} \right)
   \end{align*}
   \]

3. Set CPL := EST[END], the critical path length of the augmented DDG.
4. Compute LST (Latest Starting Time) of all nodes (this is a backward pass):
   
   \[
   \begin{align*}
   \text{LST}[\text{END}] &= \text{EST}(\text{END}) \\
   \text{LST}[y] &= \text{MIN} \left( \{ \text{LST}[x] - \text{edge}_\text{weight}(y, x) \mid \text{there exists an edge from } y \text{ to } x \} \right)
   \end{align*}
   \]

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

   (all instructions on a critical path will have zero rank)  **Why?**

Build a priority list \(L\) of the instructions in non-decreasing order of ranks.

**NOTE:** there are other heuristics
Example of Rank Computation

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

Register instruction 1 cycle
Memory instruction 3 cycle

Node x EST[X] LST[x] rank (x)
Start 0 0 0
i1 0 0 0
i2 1 3 2
i3 1 1 0
i4 4 4 0
END 5 5 0

\[ \Rightarrow \text{Priority list} = (i1, i3, i4, i2) \]
Other Heuristics for Ranking

- Node’s rank is the number of immediate successors?
- Node’s rank is the total number of descendants?
- Node’s rank is determined by long latency?
- Node’s rank is determined by the last use of a value?
- Critical Resources?
- Source Ordering?
- Others?

Note: these heuristics help break ties, but none dominates the others.
Heuristic Solution: Greedy List Scheduling Algorithm

for each instruction $j$ do
  pred-count[$j$] := #predecessors of $j$ in DDG
  ready-instructions := \{ $j$ | pred-count[$j$] = 0 \}
while (ready-instructions is non-empty) do
  $j$ := first ready instruction according to output
  output $j$ as the next instruction in the schedule
  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-instructions + \{ $k$ \};
end if
end for
end while

Remove the node $j$ from the processors node set of each successor node of $j$. If the set is empty, means one of the successor of $j$ can be issued – no predecessor!

Issue the instruction. Note: no timing information is considered here!

Consider resource constraints beyond a single clean pipeline
Instruction Scheduling for a Basic Block

Goal: find a legal schedule with minimum completion time

1. Rename to avoid output/anti-depedences (optional).
2. Build the data dependence graph (DDG) for the basic block
   • Node = target instruction
   • Edge = data dependence (flow/anti/output)
3. Assign weights to nodes and edges in the DDG so as to model target processor.
   • For all nodes, attach a resource reservation table
   • Edge weight = latency
4. Create priority list
5. Iteratively select an operation and schedule
Quiz

Question 1
The list scheduling algorithm does not consider the timing constraints (delay time for each instruction). How to change the algorithm so that it works with the timing information?

Question 2
The list scheduling algorithm does not consider the resource constraints. How to change the algorithm so that it works with the resource constraints?
The list scheduling produces a schedule that is within a factor of 2 of optimal for a machine with one or more identical pipelines and a factor of $p+1$ for a machine that has $p$ pipelines with different functions. [Lawler et al. Pipeline Scheduling: A Survey, 1987]
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 $O(n^2)$

Note: we are considering basic block scheduling here
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.
Summary

1. Data Dependence and DDG
2. Reordering Transformations
3. Hardware Parallelism
4. Parallelism Constraints
5. Scheduling Complications
6. Legal Schedules for Pipeline
7. List Scheduling
   • Weighted DDG
   • Rank Function Based on Critical paths
   • Greedy List Scheduling Algorithm
Case Study

Instruction Scheduling in Open64
Phase Ordering

Amenable for SWP?

- Multiple block
- Large loop body
- And else

Yes

No

Acyclic Global sched

Global Register Alloc

Local Register Alloc

Acyclic Global sched

SWP

SWP-RA

Code emit
Global Acyclic Instruction Scheduling

- Perform scheduling within loop or an area not enclosed by any loop.
- It is not capable of moving instructions across iterations or out (or into a loop).
- Instructions are moved across basic block boundary.
- Primary priority function: the dep-height weighted by edge-frequency.
- Prepass schedule: invoked before register allocator.
Scheduling Region Hierarchy

Global CFG

Region hierarchy

Nested regions are visited (by scheduler) prior to their enclosing outer region

Irreducible loop, not for global sched

SWP candidate
Global Scheduling Example

Before schedule

```
mov r1=<a const> (sched)
```

```
addl r1=r2,r3
```

```
ld8 r4=[r1] (sched)
```

After schedule

```
mov r1=<a const> (sched)
```

```
addl r1=r2,r3
```

```
ld8 r4=[r1] (sched)
```

Constraining dependence

1d8 is p-ready candidate

Cntl flow
dependence
Motion path
Local Instruction Scheduling

- Postpass: after register allocation
- On demand: only schedule those block whose instruction are changed after global scheduling.
- Forward List scheduling
- Priority function: dependence height + others used to break tie. (compare dep-height and slack)