Instruction scheduling

Advanced Compiler Construction
Michel Schinz – 2020-04-09

Instruction ordering

When emitting the instructions of a program, a compiler imposes a total order on them, but:

- there is (usually) more than one valid order,
- some orders might be better than others.

Example:

Two independent instructions appearing in sequence can be swapped.

Instruction scheduling

Goal of **instruction scheduling**:

Find, among all valid permutations of the instructions of a program, one that is better than the others.

(Usually, better = executes faster.)

Pipeline stalls

Modern, pipelined architectures can usually issue at least one instruction per clock cycle.

However, an instruction can execute only if its arguments are ready, otherwise the pipeline **stalls** until it is the case.

Causes for stalls:

- the instruction producing the argument has not finished executing yet,
- the argument must be fetched from memory,
- etc.

Scheduling example

The following example will illustrate how proper scheduling can reduce the time required to execute a piece of RTL code.
We assume the following delays for instructions:

Instruction kind	RTL notation	Delay
Memory load or store	$R_a \leftarrow \mathtt{Mem}[R_b + c]$ $\mathtt{Mem}[R_b + c] \leftarrow R_a$	3
Multiplication	$R_a \leftarrow R_b * R_c$	2
Addition	$R_a \leftarrow R_b + R_c$	1

Scheduling example

Cycle	Instruction
1	$R_1 \leftarrow \mathtt{Mem}[R_{SP}]$
4	$R_1 \leftarrow R_1 + R_1$
5	$R_2 \leftarrow \mathtt{Mem}[R_{SP}+1]$
8	$R_1 \leftarrow R_1 * R_2$
9	$R_2 \leftarrow \mathtt{Mem}[R_{SP}+2]$
12	$R_1 \leftarrow R_1 * R_2$
13	$R_2 \leftarrow \mathtt{Mem}[R_{SP}+3]$
16	$R_1 \leftarrow R_1 * R_2$
18	$\mathtt{Mem}[R_{SP}+4] \leftarrow R_1$

Cycle	Instruction
1	$R_1 \leftarrow \mathtt{Mem}[R_{SP}]$
2	$R_2 \leftarrow \mathtt{Mem}[R_{SP}+1]$
3	$R_3 \leftarrow \mathtt{Mem}[R_{SP}+2]$
4	$R_1 \leftarrow R_1 + R_1$
5	$R_1 \leftarrow R_1 * R_2$
6	$R_2 \leftarrow \mathtt{Mem}[R_{SP}+3]$
7	$R_1 \leftarrow R_1 * R_3$
9	$R_1 \leftarrow R_1 * R_2$
11	$\mathtt{Mem}[R_{SP}+4] \leftarrow R_1$

Instruction dependences

An instruction i_2 **depends** on an instruction i_1 when it is not possible to execute i_2 before i_1 without changing the behavior of the program.
We distinguish three kinds of dependencies:

1. true dependency – i_2 reads a value written by i_1 (**read after write** or **RAW**),
2. anti-dependency – i_2 writes a value read by i_1 (**write after read** or **WAR**),
3. anti-dependency – i_2 writes a value written by i_1 (**write after write** or **WAW**).

Anti-dependencies

Anti-dependencies do not arise from the flow of data.
They are due to a single location being reused.
Often, they can be removed by "renaming" locations, e.g. using different registers.
In the example below, the program (left) contains a WAW anti-dependency that can be removed by "renaming" the second use of R_1.

$R_1 \leftarrow \mathtt{Mem}[R_{SP}]$
$R_4 \leftarrow R_4 + R_1$
$R_1 \leftarrow \mathtt{Mem}[R_{SP}+1]$
$R_4 \leftarrow R_4 + R_1$

$R_1 \leftarrow \mathtt{Mem}[R_{SP}]$
$R_4 \leftarrow R_4 + R_1$
$R_2 \leftarrow \mathtt{Mem}[R_{SP}+1]$
$R_4 \leftarrow R_4 + R_2$

Computing dependencies

Identifying dependencies is:

- easy if they only access registers,
- impossible (in general) if they access memory.

For memory, conservative approximations have to be used. We won't cover them here.

Dependence graph

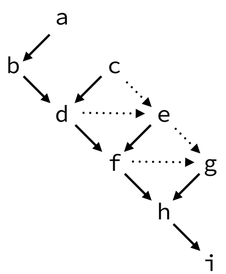
The **dependence graph** is a directed graph representing dependencies among instructions:

- the nodes are the instructions to schedule,
- there is an edge from n_1 to n_2 iff n_2 depends on n_1.

Any topological sort of the nodes of this graph is a valid schedule of the instructions.

Dependence graph example

Name	Instruction
a	$R_1 \leftarrow \texttt{Mem}[R_{SP}]$
b	$R_1 \leftarrow R_1 + R_1$
c	$R_2 \leftarrow \texttt{Mem}[R_{SP}+1]$
d	$R_1 \leftarrow R_1 * R_2$
e	$R_2 \leftarrow \texttt{Mem}[R_{SP}+2]$
f	$R_1 \leftarrow R_1 * R_2$
g	$R_2 \leftarrow \texttt{Mem}[R_{SP}+3]$
h	$R_1 \leftarrow R_1 * R_2$
i	$\texttt{Mem}[R_{SP}+4] \leftarrow R_1$



→ true dependence
⇢ antidependence

List scheduling

Optimal instruction scheduling is NP-complete.

List scheduling is:

- a heuristic scheduling technique,
- that works on a single basic block.

Basic idea:

- simulate the execution of the instructions, and
- schedule them only when their operands are ready.

List scheduling algorithm

The list scheduling algorithm maintains two lists:
- `ready`: the instructions that could be scheduled without stall, ordered by priority,
- `active`: the instructions that are being executed.

At each step:
- the highest-priority instruction from `ready` is scheduled,
- it gets moved to `active`,
- it stays there for a time equal to its delay.

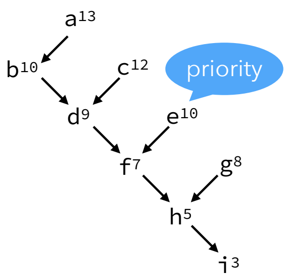
Before scheduling is performed, renaming is done to remove all anti-dependencies that can be removed.

Instruction priority

Nodes (i.e. instructions) are sorted by priority in the `ready` list. The priority of a node can be defined as:
- the length of the longest latency-weighted path from it to a root of the dependence graph,
- the number of its immediate successors,
- the number of its descendants,
- its latency,
- etc.

Unfortunately, none of these is better for all cases.

List scheduling example



A node's priority is the length of the longest latency-weighted path from it to a root of the dependence graph

Cycle	ready	active
1	$[a^{13},c^{12},e^{10},g^{8}]$	[a]
2	$[c^{12},e^{10},g^{8}]$	[a,c]
3	$[e^{10},g^{8}]$	[a,c,e]
4	$[b^{10},g^{8}]$	[b,c,e]
5	$[d^{9},g^{8}]$	[d,e]
6	$[g^{8}]$	[d,g]
7	$[f^{7}]$	[f,g]
8	[]	[f,g]
9	$[h^{5}]$	[h]
10	[]	[h]
11	$[i^{3}]$	[i]
12	[]	[i]
13	[]	[i]
14	[]	[]

Scheduling conflicts

Should scheduling be done before or after register allocation?
- If it is done first, register allocation can introduce spilling code that destroys the schedule.
- If it is done second, register allocation can introduce anti-dependencies when reusing registers.

Solution:
- schedule first,
- allocate registers
- schedule once more if spilling was necessary.