Instruction scheduling

Michel Schinz

Instruction ordering

When a compiler emits the instructions corresponding to a program, it imposes a total order on them.

However, that order is usually not the only valid one, in the sense that it can be changed without modifying the program's behaviour.

For example, if two instructions i_1 and i_2 appear sequentially in that order and are independent, then it is possible to swap them.

Instruction scheduling

Among all the valid permutations of the instructions composing a program – i.e. those which preserve the program's behaviour – some can be more desirable than others. For example, one order might lead to a faster program on some machine, because of architectural constraints.

The aim of **instruction scheduling** is to find a valid order that optimises some metric, like execution speed.

Pipeline stalls

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

However, an instruction can be executed only if the data it needs is ready. Otherwise, the pipeline **stalls** for one or several cycles.

Stalls can appear because some instructions (*e.g.* division) require several cycles to complete, or because data has to be fetched from memory.

Scheduling example

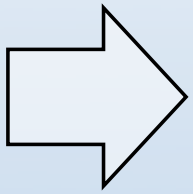
The following example will illustrate how proper scheduling can reduce the time required to execute a piece of code.

We assume the following delays for instructions:

Instruction(s)	Delay
`LOAD, STOR`	3
`MUL`	2
`ADD`	1

Scheduling example

Cycle	Instruction
1	LOAD R1 R30 0
4	ADD R1 R1 R1
5	LOAD R2 R30 4
8	MUL R1 R1 R2
9	LOAD R2 R30 8
12	MUL R1 R1 R2
13	LOAD R2 R30 12
16	MUL R1 R1 R2
18	STOR R1 R30 16



Cycle	Instruction
1	LOAD R1 R30 0
2	LOAD R2 R30 4
3	LOAD R3 R30 8
4	ADD R1 R1 R1
5	MUL R1 R1 R2
6	LOAD R2 R30 12
7	MUL R1 R1 R3
9	MUL R1 R1 R2
11	STOR R1 R30 16

After scheduling (including renaming), the last instruction is issued at cycle 11 instead of 18!

Instruction dependencies

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 behaviour of the program.

The most common reason for dependency is data-dependency: i_2 uses a value that is computed by i_1.

However, as we will see, there are other kinds of dependencies.

Data dependencies

We distinguish three kinds of dependencies between two instructions i_1 and i_2:

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

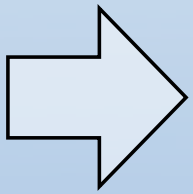
Anti-dependencies

Anti-dependencies are not real dependencies, in the sense that they do not arise from the flow of data. They are due to a single location – *e.g.* a register – being used to store different values.

Most of the time, anti-dependencies can be removed by renaming locations – *e.g.* registers.

For example, the program on the left contains a WAW anti-dependency between the two `LOAD` instructions, that can be removed by renaming the second use of `R1`.

```
LOAD R1 R30 0
PINT R1
LOAD R1 R30 4
PINT R1
```



```
LOAD R1 R30 0
PINT R1
LOAD R2 R30 4
PINT R2
```

Computing dependencies

Identifying dependencies among instructions that only access registers is easy.

Instructions that access memory are harder to handle. In general, it is not possible to know whether two such instructions refer to the same memory location. Conservative approximations therefore have to be used.

Dependency graph

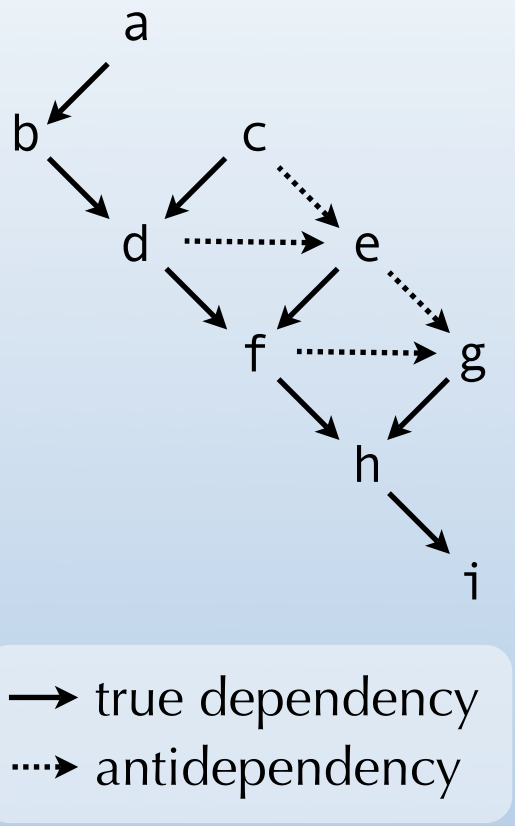
The **dependency graph** is a directed graph representing dependencies among instructions.

Its nodes are the instructions to schedule, and there is an edge from node n_1 to node n_2 iff the instruction of n_2 depends on n_1.

By topologically sorting the nodes of this graph, it is possible to compute all possible schedules of a set of instructions.

Dependency graph example

Name	Instruction
a	`LOAD R1 R30 0`
b	`ADD R1 R1 R1`
c	`LOAD R2 R30 4`
d	`MUL R1 R1 R2`
e	`LOAD R2 R30 8`
f	`MUL R1 R1 R2`
g	`LOAD R2 R30 12`
h	`MUL R1 R1 R2`
i	`STOR R1 R30 16`



Difficulty of scheduling

Optimal instruction scheduling is NP-complete.

As always, this implies that we will use techniques based on heuristics to find a good – but sometimes not optimal – solution to that problem.

List scheduling is a technique to schedule the instructions of a *single basic block*.

Its basic idea is to simulate the execution of the instructions, and to try to schedule instructions only when all their operands can be used without stalling the pipeline.

List scheduling algorithm

The list scheduling algorithm maintains two lists:

- `ready` is the list of instructions that could be scheduled without stall, ordered by priority,
- `active` is the list of instructions that are being executed.

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

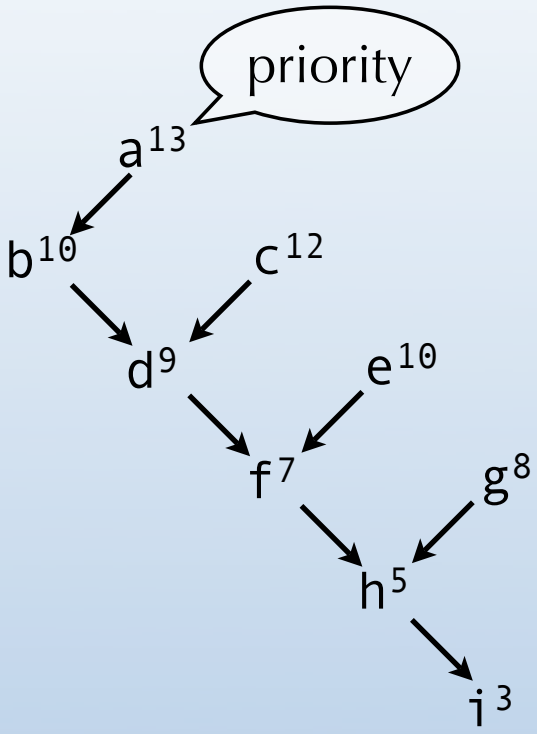
Prioritising instructions

Instructions are sorted by priority in the ready list. How are those priorities computed?

The most common scheme is to use the length of the longest latency-weighted path from the node to a root of the dependency graph as the priority.

Other schemes exits, though. For example, a node's priority can be the number of its immediate successors.

List scheduling example



Cycle	ready	active
1	[a13,c12,e10,g8]	[a]
2	[c12,e10,g8]	[a,c]
3	[e10,g8]	[a,c,e]
4	[b10,g8]	[b,c,e]
5	[d9,g8]	[d,e]
6	[g8]	[d,g]
7	[f7]	[f,g]
8	[]	[f,g]
9	[h5]	[h]
10	[]	[h]
11	[i3]	[i]
12	[]	[i]
13	[]	[i]
14	[]	[]

Scheduling conflicts

It is hard to decide whether scheduling should be done before or after register allocation.

If register allocation is done first, it can introduce anti-dependencies when reusing registers.

If scheduling is done first, register allocation can introduce spilling code, destroying the schedule.

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

Summary

Instruction scheduling tries to find an order in which instructions should be issued to improve some metric – typically execution time.

List scheduling is an instruction scheduling technique. It works by always scheduling the next instruction that is ready, *i.e.* whose operands are available. When several candidate instructions exist, a heuristic is used to decide which one to schedule next.