Computer Science Part II 2002
Paper 9 Question 7
AM – Optimising Compilers

New

Solution Notes y2002p9.tex

(a) bookwork: Observe that two instructions may be permuted if neither writes to a register read or written by the other. We define a graph (actually a DAG), whose nodes are instructions within a basic block. Place an edge from instruction a to instruction b if a occurs before b in the original instruction sequence and if a and b cannot be permuted. Now observe that the any of the minimal elements of this DAG (normally drawn at the top in diagrammatic form) can be validly scheduled to execute first and after removing such a scheduled instruction from the graph any of the new minimal elements can be scheduled second and so on. In general any topological sort of this DAG gives a valid scheduling sequence. Some are better than others and to achieve non-NP-complete complexity we cannot in general search freely, so the current $O(n^2)$ algorithm makes the choice of the next-to-schedule instruction *locally*, by choosing among the minimal elements with the *static scheduling heuristics*

- choose an instruction which does not conflict with the previous emitted instruction
- choose an instruction which is most likely to conflict if first of a pair (e.g. `ld.w` over `add`)
- choose an instruction which is as far as possible (over the longest path) from a graph-maximal instruction—the ones which can be validly be scheduled as the last of the basic block.

On the MIPS or SPARC the first heuristic can never harm. The second tries to get instructions which can provoke stalls out of the way in the hope that another instruction can be scheduled between a pair which cause a stall when juxtaposed. The third has similar aims—given two independent streams of instructions we should save some of each stream for inserting between stall-pairs of the other.

So, given a basic block

- construct the scheduling DAG as above; doing this by scanning backwards through the block and adding edges when dependencies arise works in $O(n^2)$
- initialise the *candidate list* to the minimal elements of the DAG
- while the candidate list is non-empty
  - emit an instruction satisfying the static scheduling heuristics (for the first iteration the 'previous instruction' with which we must avoid dependencies is any of the final instructions of predecessor basic blocks which have been generated so far.

- if no instruction satisfies the heuristics then either emit NOP (MIPS) or emit an instruction satisfying merely the final two static scheduling heuristics (SPARC).
- remove the instruction from the DAG and insert the newly minimal elements into the candidate list.

On completion the basic block has been scheduled.

One little point which must be taken into account on non-interlocked hardware (e.g. MIPS) is that if any of the successor blocks of the just-scheduled block has already been generated then the first instruction of one of them might fail to satisfy timing constraints with respect to the final instruction of the newly generated block. In this case a NOP must be appended.

(b) assuming the above algorithm, construct the dependency DAG. This is

```
ld r3                ld r4
 |                    |      \
st r3                st r4        add r5,r4,#4
```

The algorithm as given will choose randomly between the two candidate loads (this shows a deficiency in it really). Suppose for pesimism it chooses `ld r3`. Then the candidates are now `st r3` and `ld r4`. Only the latter does not cause a stall, and so is emitted. Now all remaining instructions are candidates, but the `st r3` does not cause a stall, but should be emitted because it is more likely to cause a stall than the `add`. Now the `add` is emitted to avoid a `st-st` stall, and finally the `st r4`. Thus we have:

```
1   ld  r3,0(r1)
3   ld  r4,4(r1)
2   st  r3,0(r2)
5   add r5,r4,#4
4   st  r4,4(r2)
```

The only real alternative is

```
3   ld  r4,4(r1)
1   ld  r3,0(r1)
4   st  r4,4(r2)   (because st can be first part of a stall, not add)
5   add r5,r4,#4
2   st  r3,0(r2)
```

Infelicity in question: it is not stated whether or not r1 and r2 may be near enough that reads and writes may overlap. The suggested answer above reflects the intention when the question was set, but note that perfect marks could therefore be obtained also by making the 2nd load have a dependency on the 1st store (in lectures it was suggested that treating memory as a 1-bit-like register 'MEM', written by stores and read by all reads unless demonstrably unaliased would have this effect simply and systematically).

(c) Dependency dag is:

```
add  or
|   /
sub
|
xor
```

(The add-xor dependency is not shown transitivity of dependency means this has no effect on the schedulings of the DAG). Hence there are just two schedulings, which of 'add' and 'or' goes first, then the rest is fixed. (Remember the antidependency between the sub and xor!).