332 Advanced Computer Architecture

Unassessed tutorial exercise 6

Avoiding load delays

Load and store instructions are a key factor in program performance. Caching is a well-known approach to improving access latency. However, loads and stalls also interfere with performance by limiting the scope for re-ordering instructions. This exercise explores some approaches to avoiding this effect.

Exercise 6.1: Speculative loads

Once we have the mechanism to support speculative execution, we can speculatively execute a load instruction even if there is an uncompleted (perhaps also speculative) store instruction which might write to the same address.

Presumably we need to maintain a table of addresses affected (or used by?) yet-to-be completed instructions.

1. Explain how this table is maintained and used
2. What happens when a misprediction occurs?
3. What determines the required number of table entries - what if the table is too small?
4. What are the potential disadvantages of speculative pre-fetching - can it slow your program down? Can it lead to wrong results?

Exercise 6.2: Value prediction

Another technique arises from the following observation: when a load instruction executes, the value it gets was (in most cases) stored by some previously-executed store instruction. It turns out that most of the time (40%–75%), the address of that store instruction is easily predicted - it's the same load that produced the value used by this store last time it was executed.

Why might this be so?

Presumably we again need to maintain a table of some kind.

1. Explain how this table is maintained and used
2. What happens when a misprediction occurs?
3. What determines the required number of table entries - what if the table is too small?

Exercise 6.3: Multithreading

Some loads can't easily be predicted by the above techniques, and don't hit in the cache. In this case, it may take hundreds of clock cycles to get the data. A popular idea is to run several concurrent threads on the same CPU, so that when a cache miss stalls one thread, the others can use the resources.

There are three distinct ways to do this:

- Interleave issue of instructions from the N different threads, so that a load and a store in any given thread are sufficiently well-separated that the data will already be available.
- When a cache miss occurs, save the current context and start issuing instructions from a new (unblocked) thread
- Build two or more separate instruction issue units, sharing a common pool of functional units.

Comment on the factors which influence whether multithreading in each of these forms is a good idea,

Paul Kelly, Imperial College, 2001.

332 Advanced Computer Architecture: Notes on the solutions to Exercise 6

Exercise 6.1: Speculative loads

Once we have the mechanism to support speculative execution, we can speculatively execute a load instruction even if there is an uncompleted (perhaps also speculative) store instruction which might write to the same address.

Presumably we need to maintain a table of addresses affected (or used by?) yet-to-be completed instructions.

1. Explain how this table is maintained and used

   A: With out-of-order memory accesses we have to consider RAW, WAR and WAW hazards.

   We will probably already have a write-back buffer (WBB) - a queue of (committed) stores waiting to be serviced by the memory system. When a load is executed, speculatively or otherwise, we have to check whether its address matches any in the WBB. This deals with RAW hazards caused by committed stores.

   What about a store which has yet to complete? We may not even know its target address, but we want a later load to proceed... We could allow this speculatively - but when the store finally commits, we have to check whether the load did in fact get the right data - so we need a table of loads speculatively issued in this way. Such a load can be committed only after all preceding stores have committed - and must be discarded if any if any such store's target address matches.

   See for example the ALAT (advance load address table) in the Intel IA-64 design.

2. What happens when a misprediction occurs?

   A: A misprediction here is when the load was allowed to proceed ahead speculatively, ahead of a store which writes to the same address. When the store's target address is known (and committed), the speculative load is either squashed (if matches), or committed (of course with multiple such stores we have to wait for them all).

3. What determines the required number of table entries - what if the table is too small?

   A: Similarly to the ROB size, the ALAT table size limits the number of loads that can overtake a store. When its limit is reached, execution of a load is delayed until a slot becomes available. This happens when one of the outstanding loads is committed.

4. What are the potential disadvantages of speculative pre-fetching - can it slow your program down? Can it lead to wrong results?

   A: Speculative loads could causes unwanted page faults. The fault should be raised when the instruction is committed. Speculative loads can displace valuable data from the cache. Speculative cache misses can add to memory contention (especially in a multiprocessor).

   Finally, if two loads to different addresses are serviced out-of-order, you could get unexpected results. Suppose another processor is writing to these locations in some order; the first process may observe the updates as if they had happened in a different order.

   To fix this, many processors have a special instruction which allows in-order memory access to be enforced (eg see the MB instruction on the Compaq Alpha).

Exercise 6.2: Value prediction

See Lipasti etal: "Value Locality and Load Value Prediction", ASPLOS VII-1996 (`http://citeseer.nj.nec.com/lipasti96value.html`).

Also "Improving the accuracy and performance of memory communication through renaming" by Gary Tyson and Todd Austin (IEEE Micro-30) (`http://citeseer.nj.nec.com/tyson97improving.html`).

Exercise 6.3: Multithreading

See papers on the Tera MTA `http://www.tera.com/products/systems/craymta/`,

Also Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 392–403, June 22–24, 1995 `http://citeseer.nj.nec.com/32969.html`.

Paul Kelly, Imperial College, 2000.