Assignment 2

COL718

March 2, 2016

1 Background

Memory ambiguity arises in OoO pipelines when the addresses of memory instructions are not known while scheduling. Specifically, a load instruction cannot be scheduled until the addresses of all previous store instructions are determined. This leads to performance degradation, since all subsequent loads might not be dependent on the store with an unknown address.

Let us take an example to understand this problem.

1: st r2, 10[r5]

2: ld r1, 4[r3]

First, the instructions are sent to an adder to compute the effective address. In this case: 10 + (contents of r5), 4 + (contents of r3). Until the store address of instruction 1 is computed, we do not know whether the subsequent load instruction is dependent on this store or not. In conventional processors, all loads following a store with an unresolved address wait until the address is computed.

To counter this problem, several memory disambiguation techniques have been proposed. One such method is discussed in [1]. This method predicts whether a load instruction is dependent on any preceding store instruction in the instruction window. If it is non-colliding, it can bypass all previous stores in the instruction window. It is based on a Collision History Table (CHT), which is similar to the Branch History Table (BHT).

Another mechanism creates load-store pairs and predicts the exact store (if one exists) that the load is dependent upon ([2]). In a modified technique *Store Sets* are created and loads and stores are dynamically clustered which have aliased the same memory addresses in the past [3]. Loads and stores within the same store sets are issued in order.

Further, in case of a misprediction, two possibilities exist:

1. Stall the pipeline for n cycles to simulate the misprediction penalty and then continue execution. (This is a simplification for simulating the total cycles required, the program execution will be incorrect.)

2. Execute the load and its dependent instructions again. This technique is called as *replay*.

2 Objective

In this assignment, you have to implement the load store dependence prediction scheme for the OoO processor pipeline in the Tejas Simulator.

2.1 Install the Tejas Simulator and understand its operation

1. Tejas is open-source, Java-based multicore architectural simulator developed in-house by the Srishti group. It will be used in the subsequent assignments. The instructions to set up Tejas can be found at http://www.cse.iitd.ac.in/tejas/install.html.
2. Some details about Tejas can be found in [4] and [5].

2.2 Implement the two load-store dependence speculators

You have to implement the following cases:

1. A load store predictor that predicts whether a load is dependent on **any** previous store instruction. Let us call this technique *anyStore* . When a colliding load instruction is mispredicted as non-colliding, freeze the pipeline for 10 cycles and then continue with the execution. (See how it is done for branch misprediction in Tejas.)
2. A load store predictor that predicts the exact store a load is dependent upon, by creating load-store pairs or clusters. Let us call this *pairLS* . Handle mispredictions by freezing the pipeline.

2.3 Implement the replay logic using a replay buffer

To ensure correct operation, the mispredicted load and its dependent instructions must be replayed. Implement the replay technique described in Figure 1 (as discussed in the class; refer to slide#12 of OOO-III). There should be a

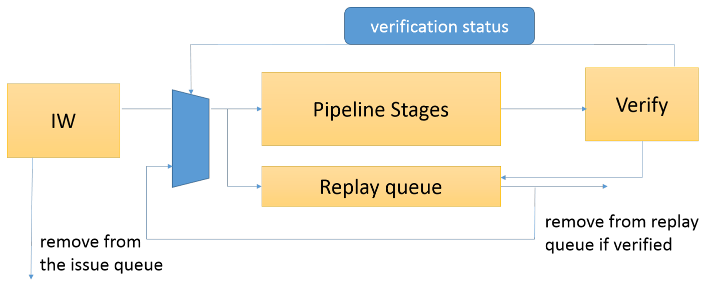
verification status
IW
Pipeline Stages
Verify
Replay queue
remove from replay queue if verified
remove from the issue queue

Figure 1: A method of replaying

Parameters	Traditional	*anyStore* without replay	*anyStore* with replay	*pairLS* without replay	*pairLS* with replay
Size of the Predictor					
Prediction Accuracy					
Percentage of memory instructions					
Percentage of load instructions					
Percentage of loads predicted as colliding					
Percentage of loads actually colliding					
IPC					
Speedup[1]					
L1-cache hit rate					
L2-cache hit rate					

Table 1: Results

means to choose whether to replay or not using any speculator. Make changes wherever necessary to ensure that the algorithm works correctly.

2.4 Report the results obtained

Report the values in Table 1 for the two predictors and the traditional approach implemented in Tejas (in which loads are selected only when all preceding store addresses are resolved) with and without replay. Use benchmarks from the SPEC suite, and simulate using 30 million instructions. We will release these benchmarks in a few days. Until then use benchmarks such as matrix-multiplication.

3 Submission

All your changes should be contained in the "src/simulator/pipeline/outoforder" package. Keep all parameters configurable through the "config.xml" file. There should be an option to include load store predictor or simulate without it. Submit the "outoforder" sub-package along with a design document containing the basic design (storage structures used, their size and organization) as well as benchmarks chosen and results obtained (Table 1) on Moodle. The submission deadline is March 25, 2016, 23:55 hours.

[1] relative to traditional approach

4 Grading

There are 3 parts in this assignment - *anyStore* , *pairLS* and replay. *anyStore* carries 5 marks, *pairLS* carries 6 marks and replay carries 4 marks. The actual marks will be decided on the basis of a viva.

5 References

1. lph.ece.utexas.edu/merez/uploads/Tenure/LoadSched_ISCA26.pdf
2. http://www.cse.iitd.ac.in/~srsarangi/col718_2016/papers/renameissue/dynamic-track.pdf
3. http://www.cse.iitd.ac.in/~srsarangi/col718_2016/papers/renameissue/store-sets.pdf
4. www.cse.iitd.ac.in/~srsarangi/files/papers/patmospaper.pdf: Sarangi, Smruti R., et al. "Tejas: A java based versatile micro-architectural simulator." Power and Timing Modeling, Optimization and Simulation (PATMOS), 2015 25th International Workshop on. IEEE, 2015.
5. http://www.cse.iitd.ac.in/tejas/overview.html