Memory Dependence Prediction
(Sticky Bit Predictor, Store Sets)

ECE 721: Advanced Microarchitecture
Prof. Eric Rotenberg
Rhushikesh Prabhune, Ajinkya Tundurwar

1. Project Description

The project aims to implement Store Sets & Sticky-Bit memory dependence predictor for improving the performance by reducing the number of memory-order violations.

Key Outcomes:

- ➔ Implementation of Sticky-Bit predictor with cyclic clear mechanism.
- ➔ Finding variations of Sticky-Bit performance over variations in the cyclic clear interval.
- ➔ Implementation of Store Sets which uses the (Active List Index + Active List Size as its Store Inum) of stores to create load-store dependencies for load and store instructions.
- ➔ Comparison of Store Sets performances with no speculation, blind speculation, oracle memory disambiguation and Sticky-Bit predictor.

2. Simulator Accomplishments

The Sticky-Bit and Store Sets memory dependence prediction have been implemented and tested successfully on 721sim.

Major microarchitecture feature and/or major simulator config	Perfect Branch Prediction			Real Branch Prediction		
	astar (#instr)	hmmer (#instr)	bzip (#instr)	astar (#instr)	hmmer (#instr)	bzip (#instr)
Sticky Bit Predictor	100M	100M	100M	100M	100M	100M
Store Sets	100M	100M	100M	100M	100M	100M

3. Microarchitecture Design

Unknown stores open up many opportunities to manage loads in order to boost performance. Blind speculation never stalls any loads since it assumes no load-store dependency. This can create memory-order violations which take many cycles to resolve. On the other hand, non-speculative execution assumes all the loads are dependent on unknown stores which leads to unnecessary stalling of loads even when they are not dependent on the corresponding stores. To solve these problems,

different memory dependence predictors are described in literature. This project implements two such predictors, namely, sticky bit and store sets.

3.1 Sticky Bit Memory Dependence Predictor

This predictor consists of a 1-bit predictor table indexed by the load PC's. The bit is set when a load violates with a store. When the same load is encountered again, the load is checked for store dependency in the dispatch stage and the load is stalled if its sticky-bit is previously set. The stalled load is periodically checked for execution via an autonomous *load_replay* buffer engine in the execution stage. The sticky bit algorithm is shown below:-

```
if(sticky bit == 1)
   stall until all unknown store address generated;
else
   execute load;
   if(load_instr == active_list.head)
        if(load_viol == 1)
           sticky bit = 1;
           complete squash;
        else
           retire;
```

The sticky-bit memory prediction can result in false dependencies for load many times which will result in unnecessary stalling of load & thus, degrading IPC performance. The solution to this decrease in IPC performance can be deployment of a cyclic clear mechanism for sticky-bit table which periodically flash clears all the sticky bits after a fixed number of instructions have been retired. The cyclic clear algorithm has reduced the false dependencies to some extent & thus, improving the IPC performance. The optimal cyclic clear interval mainly depends on the application in use. The sticky bit table must have enough read ports so as to process enough load instructions at the dispatch stage. A write port is required at retirement stage to set the sticky bit for the corresponding load. The sticky bit implementation can be implemented with registers or SRAM's.

3.2 Store Set Predictor

Store set memory dependence prediction is a more efficient way of predicting load-store dependencies when compared to Sticky-Bit memory dependence prediction where a single bit is set once a violation is detected. The Store Set Predictor is implemented using two data structures, namely, Store Set Identification Table (SSIT) and Last Fetched Store Table (LFST).

The store sets memory dependence predictor was implemented by augmenting the load-store queue with SSIT and LFST tables. The stalling of load or store instructions was decided at the dispatch stage depending on load or store entry in SSIT & its corresponding entry in LFST, while the creation of load and store entries in SSIT and LFST tables was done at retirement stage on detection of load violation. The store after its execution, invalidates its corresponding inum entry in the LFST table for future load/store instructions. The scheduling of stalled loads for execution was done by load_replay buffer engine. The store set predictor was able to complete its full execution of 100M instructions for astar,

hmmer & bzip benchmarks successfully (these are the benchmarks which have been tested successfully).
We also tried to implement a store set memory dependence predictor in a different way when compared to the above one by augmenting the issue queue with SSIT & LFST tables. This implementation involves checking for load-store dependencies using SSIT & LFST at the dispatch stage for the IQ. For this implementation, both load & store memory operations were given an extra operand which gives information about the stalling of the instruction. This memory operand was filled with store inum from the corresponding load or store instruction entry in the LFST table. If the load or store is found to be dependent on a store instruction, then the ready bit corresponding to the above operand is reset to 0. The corresponding operand ready bit is again set to 1 when the store instruction on which the load/store is dependent broadcasts its corresponding inum to issue queue after its execution. When the above operand ready bit is set to 1 along with other operands ready bit set to 1, then the corresponding load/store instruction can proceed to execution. The performance obtained from this predictor was comparable to the sticky-bit predictor and was better when compared to no speculation. On further thought, we expect the issue queue implementation to give a better performance than the LSQ performance since the issue queue approach stalls the loads only till the dependent store executes. Whereas, the LSQ approach stalls the loads and executes the stalled load through the load_replay buffer engine instead of just checking for one particular store (which the load is dependent on). We were not able to delve into the development of this method of store set implementation much because of time constraints. The simulator was able to run for 10M instructions successfully.

4. Simulator Configurations

The `--disambig` command line argument was altered to include store set implementation and cyclic clear interval for both the sticky bit and store set memory dependence predictor. Following are the options available to run the 721sim.
`--disambig=<oracle>,<spec>,<mdp>,<cylic_clear>`
<oracle>: 0 - No Oracle Speculation, 1 - Oracle Speculation
<spec>: 0 - No Speculation, 1 - Blind Speculation
<mdp>: 0 - No memory dependence predictor, 1 - Sticky-Bit predictor, 2 - Store-Set predictor
<cyclic_clear>: Number of instructions after which predictor table will be cleared , 0 - Predictor table will not be cleared
The *473.astar_test.76.0.22.gz* and *456.hmmer_test.74.0.22.gz* benchmarks were used to analyze the results and plot the graphs.
Input parameters to the simulator:

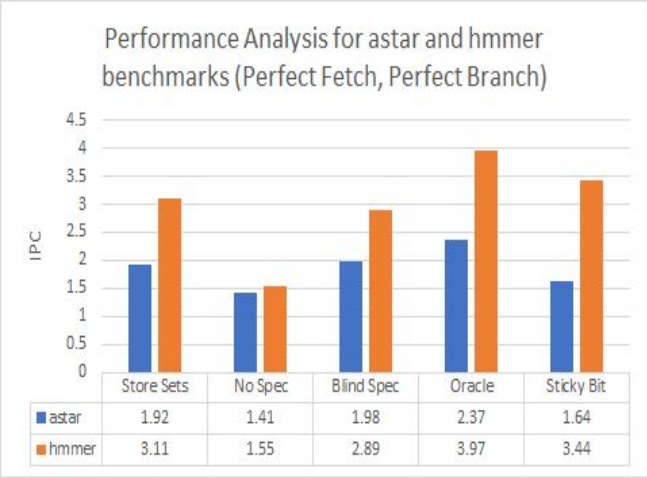
Data Structure	**Size**	**Data Structure**	**Size**
Fetch Queue	64	Issue Queue Partitions	4
Branch Checkpoints	32	Fetch, Decode, Retire Width	4

Active List	256	Issue Width	8
Load-Store Queue	128	Memory	2048 MB
Issue Queue	64		

Predictor Type	**Cyclic clear interval**
Sticky Bit	1M, 10M instructions, no cyclic clear
Store Sets (LSQ)	

Graph Number	**Variable Parameters**	**Values taken for analysis**
#1	Disambiguation	No spec, blind spec, oracle, sticky bit, store sets, no cyclic clear
#2	Cyclic clear interval, real/perfect fetch, real/perfect branch prediction	1M instr, 10M instr, no cyclic clear
#3	Cyclic clear interval, real/perfect fetch, real/perfect branch prediction	1M instr, 10M instr, no cyclic clear

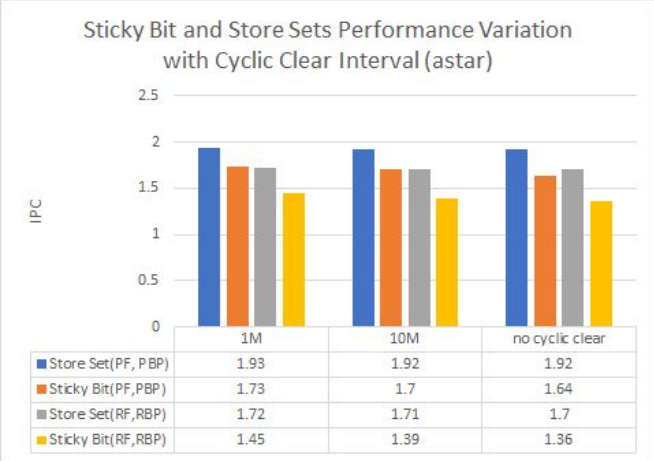
5. Results and Analysis



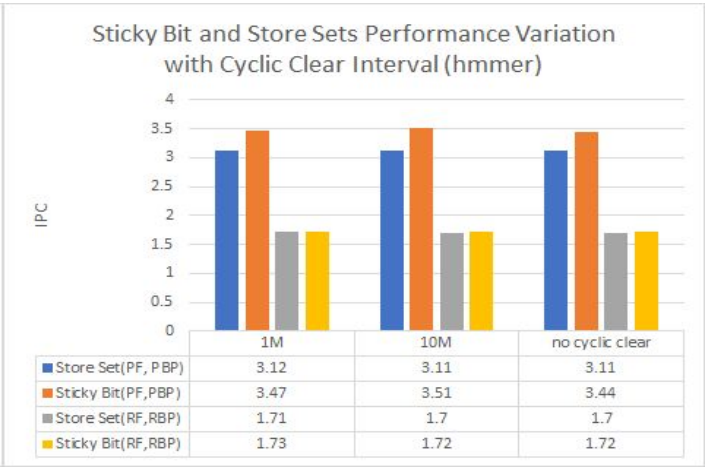
Graph 1

It can be seen that the performance improvement for *astar* benchmark does not improve much after implementation of store sets and sticky bit predictor as compared to non-speculative execution. On the other hand, the improvement in IPC is much larger for *hmmer* benchmark as compared to non-speculative execution. This can be accounted for by a large number of memory operations in the *hmmer* benchmark as compared to *astar*. Hence, the number of false dependencies are reduced in the *hmmer* benchmark by a large

extent which results in the spike in the IPC.



Graph 2



Graph 3

It can be seen from the graphs that the store sets predictor has performed better than the sticky bit predictor for the *astar* benchmark whereas it has performed worse than sticky bit prediction in *hmmer*. This could be explained by the fact that since *hmmer* has a lot of memory operations as seen from Graph1, a particular load can have dependencies with many store instructions (different store instructions but same store address as load address). Therefore, the sticky bit will perform better since irrespective of the store, the load is stalled once the bit is set. Store sets on the other hand use the Store PC to speculate dependencies between a load and a store (store PC will be different and load will be speculated to not have any dependency, resulting in memory-order violations).

6. Future Work

There is not much improvement in the performance after implementing the sticky bit predictor and store sets as mentioned in the literature. However, as discussed in the earlier sections, we also implemented the store set predictor by augmenting it to the issue queue. The next step will be to successfully run the simulator beyond 10M instructions and get the expected performance from the simulator. Another way to boost performance which comes to mind is to have a confidence counter attached to every entry in the SSIT which can decide whether there should be a load-store dependence entry in the LFST table or not. The mechanism for decrementing the confidence counter (if the dependency was incorrect) and incrementing the confidence counter (if the dependency was correct) must be deployed along with the 2-bit counter. The 2-bit counter could be saturated at 3 and 0 respectively. The dependency will be assumed correct if the counter is greater than or equal to 2 and if not, the load is allowed to execute as was the case with blind speculation.