Memory Dependence Prediction using Store Sets: An Analysis

Mayuresh Gharat, Mithilesh Bhat, Suyog Mapara and Zubeen Dedhia

1. Introduction

Out-of-order execution of instructions gives better throughput than in-order pipelines since instructions can be dynamically scheduled while other instructions are stalled due to cache access latencies or data dependencies. Out of order execution certainly increases Instruction Level Parallelism (ILP) but precautions have to be taken to maintain the proper data dependencies between instructions. There are two types of data dependencies - register and memory dependencies. Register dependencies are solved at decode time, but memory dependencies are more difficult to handle for reasons explained further. Here, we review a mechanism to improve out-of-order scheduling for memory dependent instructions using the concept of 'Store-Sets' proposed by George Chrysos and Joel Emer in their paper "Memory Dependence Prediction using Store Sets'[1].

2. Motivation

An out of order processor must respect the data dependency between instructions. Register dependencies occur when an instruction writes to a register and a subsequent instruction reads from the same register. These dependencies involving registers are known at decode time and these register conflicts are eliminated using register renaming. Memory dependencies occur when an instruction writes to a memory location and a subsequent instruction reads from the same memory location. However, unlike register dependencies, these are known only after memory addresses are computed in the execute stage. And, if we wait till the outcome is known, we lose most of the benefit of the out of order core. We can instead do a speculative execution of memory access instruction and have a mechanism for backing out when we cause memory order violation. We investigate the viability of the concept of store sets as a mechanism to perform such a speculative execution.

3. Baseline System

Our system simulates a 8 way superscalar out of order pipeline with 2 stages in front-end (fetch, rename-dispatch) and 3 stages in back-end (issue, execute, commit). We assume perfect fetch, branch-prediction, no first level cache misses and no structural hazards. We also assume same latency of 1 cycle for all instructions and load-use penalty of 2 cycles. We study performance of our memory dependency predictor for different sizes of instruction window.

Our system is capable of detecting memory violation in the same cycle in which store executes and we flush the whole instruction stream after culprit load and restart fetch from that load in the next cycle.

4. Detecting and Recovering from Memory Order Violations

4.1 Detecting Memory Order Violations:

After every store is done, it checks re-order buffer for presence of any younger load that has already read the value it has just produced. In case it finds one or many such loads, it invokes recovery mechanism for oldest such load. A real system will probably use load and store buffers for this purpose (as traversing re-order buffer may be expensive). We did not simulate these structures as they should not affect our results.

4.2 Recovery Algorithm

We need to recover all the global state of the processor which consists of score-board, re-order buffer and map-table. In addition we need to restore the fetch stream as it was when conflicting load was fetched. We describe these algorithms below.

4.2.1 Implementation of Fetch Stream

Our fetch unit maintains an in-memory stream of fetched micro-ops implemented as a deque. During recovery we put flushed MicroOps at the front of the queue (in reverse order of fetch). Fetch routine always checks the fetch-queue before reading next MicroOp from the trace and removes first MicrOp from the queue if it is not empty.

```
UnExecuteMicroOp(MicroOp m)
For each valid architectural
destination D[i] for m
  free_reg=mapping[D[i]]
  mapping[d]=m.physicalRegToFree[i]
  put free_reg in free register
queue.
  scoreBoard[m.physicalDestination
[i]] = 0]
  Put MicroOp back on the fetch
stream.
```

The function above restores the maptable, scoreboard and fetch stream to a state which would have reached if MicroOp m was never fetched.

```
RecoverFromMemoryOrderViolation(Mic
roOpfaultingLoad)
      For each MicroOp m younger
      than the faultingLoad
      (including load itself) taken
      in reverse order from Re-
      order buffer
            UnExecuteMicroOp(m)
            Remove m from Re-order
            buffer.
```

5. Experiments and Analysis

Firstly we present argument for memory dependence prediction. We analyze performance of following hypothetical configurations.

5.1 No Speculation

We do not issue loads speculatively, instead we wait for all older stores in the instruction stream to complete. Memory violations are completely avoided at the expense of reduced parallelism. Loads which could have been scheduled earlier than unrelated stores are forced to wait in the instruction queue .

5.2 Naive speculation

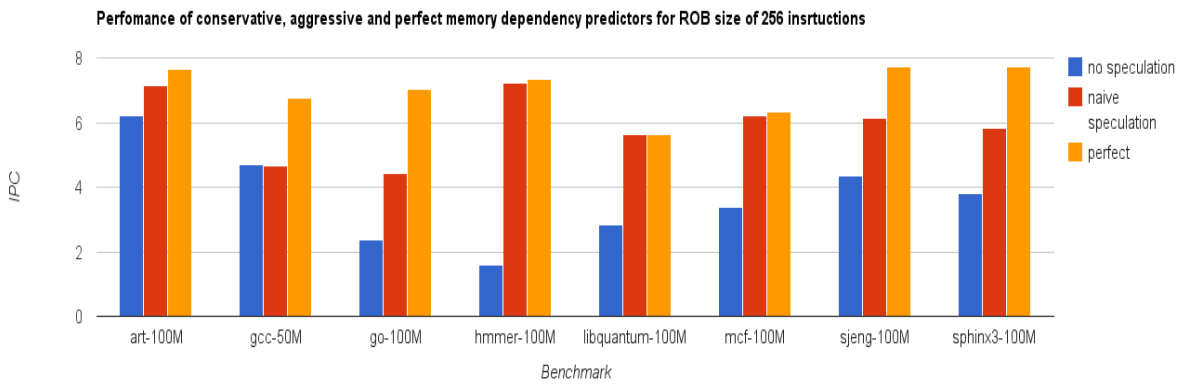
Loads are issued as soon as they are ready, irrespective of any stores present in the queue. When a store executes, all previous executed but non-committed loads are checked for the accessed memory address. In the event of a memory violation being detected, recovery is performed, as described earlier.

5.3 Perfect speculation

We take the advantage of the fact that memory addresses are available in the trace to get upper bound on the performance any memory dependency predictor can achieve. Here we issue loads only if it does not conflict with any of the older stores.

We observe that naive-speculation performs better than conservative approach in most cases reflecting speculative execution can be useful. We also note that both the methods are considerably worse than perfect memory dependence predictor indicating there is much scope for the improvement. The main observation here is that speculative scheduling of

Figure 1



loads is preferable to the play-it-safe approach.

Now we introduce the speculative load execution scheme called store sets suggested in [1].

6. Store Sets

6.1 Introduction

The basic assumption that is taken while using store sets is that by looking at the historic behaviour of memory order violations we can predict future memory dependencies.

A store set is a set of dependent store instructions for each load in a running program. Initially all loads have empty store-sets. For conflicting load and stores (execute in the wrong order, causing a memory order violation) the store PC is added to the load's store set. When a load is fetched, the processor will determine which stores in the load's store set were recently fetched but not yet issued, and create a dependence upon those stores. From now onwards, every time the processor comes across the load, it checks if the dependent stores in the store set have been recently executed or not issued, and creates a dependency. For loads that have no dependencies will execute as soon as possible; others will be dependent only on those stores that are present in the store set. Multiple situations of dependencies can arise:

- Multiple loads depend on the same store, or
- One load depending on multiple stores.

To take care of these conditions following configurations of store sets may be used:

- Each load in the program has its own store set with multiple stores, but a store can reside only in one store set. On a memory-order violation, a store is eliminated from any store set it is in, and placed in the store set of the load with which it conflicted last.
- Allow each load in the program to specify at most one store dependence. On a memory-order violation, the store replaces any prior store in the load's store set.
- No limits on either the store set size or on the number of store sets in which a store can appear.

6.2 Experiments and Analysis

Firstly we look at how well store set can perform given infinite resources. i.e we let any load in program to depend on any number of stores, and let store have multiple loads depending on it. To get these values we use store sets with no restriction on number of stores in given store set or number of store sets a store can belong.

In Figure 2, we see that best possible store set implementation is a significant improvement over naive speculation studied before. Also in most cases it approaches performance of perfect memory predictor. Thus we conclude that a system that approximates infinite store set implementation can benefit from store set idea.

Next we study following store set configurations:

- A load can depend upon multiple stores and a store can have multiple loads depending upon it.
- A load can depend upon multiple stores but a store can exist in only one store set at a time.
- A load can depend on at most one store but a store can have multiple loads depending on it.

Figure 2

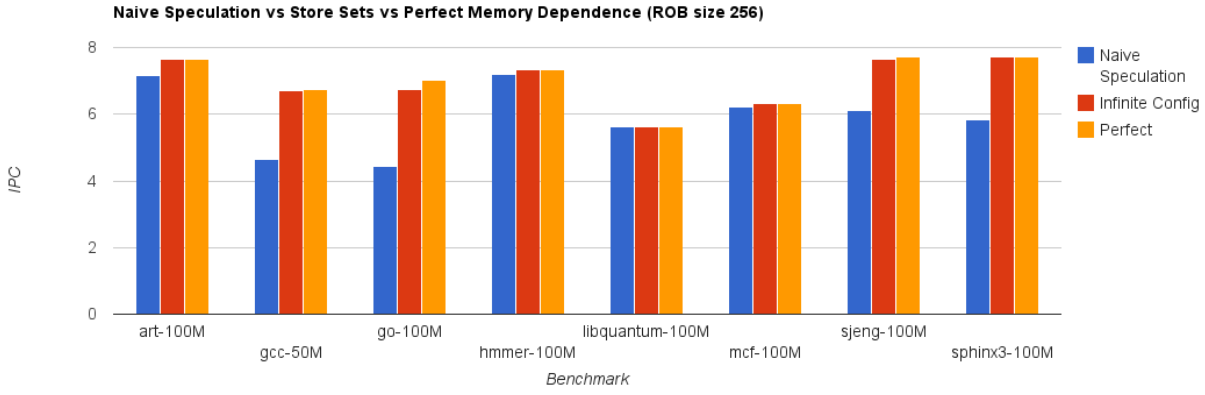
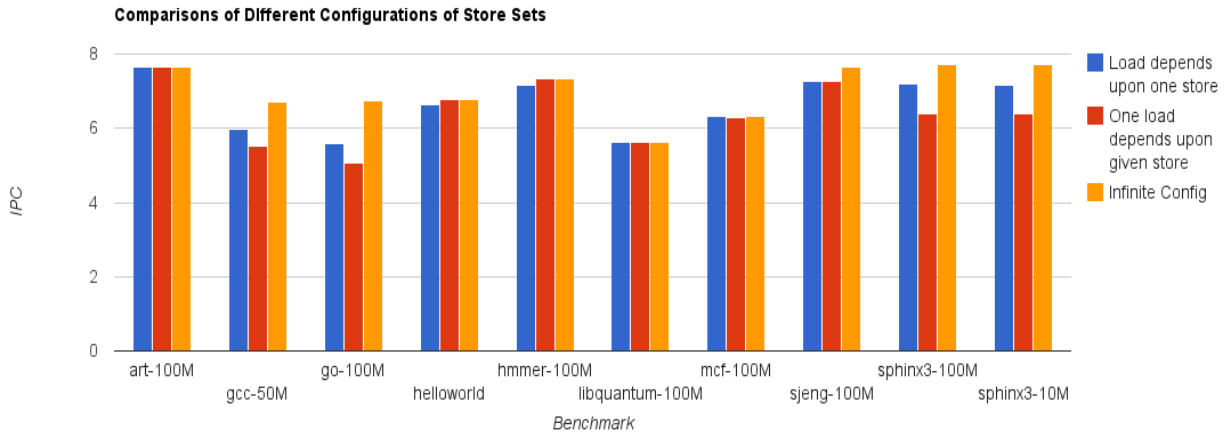


Figure 3

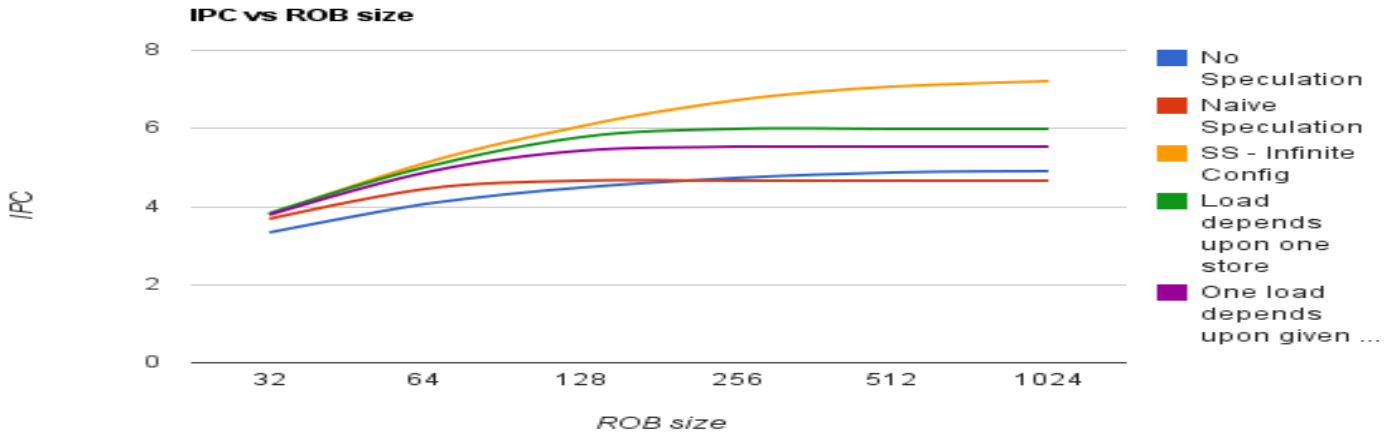


The results from Figure 3 show that enforcing either of the constraints of limiting the number of stores which a load depends on, or limiting the store sets to which a store can belong to leads to a decrease in performance of the processor. The configuration which allows multiple loads to depend on multiple stores, and stores to have multiple loads depending on them shows highest performance among the three. The result shows that a configuration with 'load depends upon at most one store' performs better than one which 'at most one load depends upon given store' This is expected as the former situation corresponds to 'one writer multiple reader' pattern which by far the common programming paradigm and restricting number of loads depending on given store will lead to more severe performance degradation.

Finally we study store set performance for different sizes of instruction window for GCC.

From Figure 4, we see that memory dependence prediction gains importance as the instruction window increases, we see that. At low ROB sizes, there is no significant difference in the IPC values for all the approaches. However, as the ROB size increases, the store set implementation performs markedly better than the no-speculation/naïve speculation. This is expected because as the ROB size increases, scope of the dynamic scheduling increases. This increases the chance of a load following store in the program order executing out-of-order, causing the memory violation. Thus, the store-set approach gives better performance than the other schemes. Another reason why store sets are more effective could be that at high ROB sizes, the chance that stalling due to register dependencies reduces, leaving memory dependencies as the major cause for delays.

Figure 4



7. Conclusion

Memory dependencies in out-of-order execution can affect the expected throughput of the processor. After observing that speculative execution of loads leads to better performance than a conservative approach, we conclude that this performance is subject to improvement by use of some predictive logic. We then decided to implement and analyze the Store-Sets approach, as proposed in [1], and found that it gives a huge improvement over the naïve speculation approach. Not only was there an improvement, but also this method also enables us to deal with conditions such as multiple loads depending on a single store and when one load depends on multiple stores. Some low cost implementations suggested in [1] were also implemented and found to be better than naïve speculation (but not as good as the infinite case). The (infinite) memory dependence predictor, which was implemented as suggested in [1], exhibited near optimal performance – as good as the perfect memory dependence predictor. Thus proving that a memory dependence predictor suggested in [1] is indeed an important piece of technology in order to extract maximum performance out an out of order processor. Hence, we rest our case.

8. Acknowledgements

We thank Prof. Milo Martin for giving us the opportunity and encouraging us to take up this analysis. We would also like thank our TA's James Anderson and Cem Karan for providing feedback and guidance for this project.

9. References

[1] Memory Dependence Prediction using Store Sets, Chrysos and Emer, in the Proceedings of the 25th Annual ACM/IEEE Conference on Computer Architecture, June 1998.