CS752 Project Report

A Study on Memory Dependence Predictor

Danni Wang and Yuting Liu

University of Wisconsin – Madison

Abstract

With the coming era of out-of-order processors, memory performance has become the more and more important as the bottleneck limiting the performance of parallelism exploration. Modern dynamically scheduled processors require for higher instruction his especially when involved in memory access. With the support of memory dependence prediction, memory performance is demonstrated the great improvement under the SPEC2006 benchmark. We implement the naive predictor as the base model and implement the store-load pair predictor, a technique which dynamically identify the store-load pairs and assign a synchronization mechanism. Together with the original implementation of store sets in gem5 environment, we evaluate and compare these different memory dependence predictors within the context of a multi-scalar processor.

Key words: memory dependence predictor, store-load pair, store set

I. Introduction

With the development of multi-scalars to allow for the execution of out-of-order programs, the instruction level parallelism is explored with great effort. The performance of the processors will be significantly influenced by data dependencies, of which there are two types: register dependencies and memory dependencies. Instead of register dependencies that can be determined and resolved in the earlier stages, memory dependencies would take more effort to be determined. Moreover, with the incorrect execution order, all relevant executions of memory instructions should be re-executed, which would lead to the performance penalty.

The loss of performance on memory dependences is due to memory-order violations and false dependences on unrelated stores. Thus the goal of memory dependence predictor is to execute the load instruction as early as possible while not incurring memory-order violation. To satisfy the goal, the mechanisms such as store sets and store-load pairs are proposed to explore more instruction level parallelism hidden by memory dependences.

Many research have focused on memory dependence prediction and different mechanisms to avoid the memory-order violations have been proposed. Steely, *et al.*, of Digital Equipment Corporation filed a patent on a general framework allowing loads to speculate around prior stores and synchronizing the execution of the store load pair where the dependence exists [1]. Hesson, *et al.*, of IBM filed a patent for the store barrier cache [2], which keeps the track of stores that tend to cause memory-order violations, where store can set a bit in the barrier cache to indicate the memory-order violation they may cause. Moshovos et al. published a comprehensive description of memory dependence prediction to identify the memory dependencies [3] and different types of memory dependence predictors, especially for the out-of-order processors. The implementation of store sets is also proposed by Chrysos in the published paper, where the index based table is implemented to support the functionalities of store sets to solve the problem.

In this paper, we will mainly focus on the memory dependence, which occurs when a store instruction and subsequent load instructions access the same location. The scheduler would facilitate to decrease the possible memory-order violations with the prediction based on the observation of the history behaviors and delay the execution of loads that are predicted to cause the violation. Our work is conducted in Gem5 O3 CPU, involving (1) study the memory dependence unit in O3 CPU in gem5, which predicts the memory reference instructions using store sets, (2) modify the memory dependence unit to implement the naive predictor, which actually does not use memory dependence predictor and allows the load instructions to execute out-of-order and commit memory-order violations, (3) implement a different predictor based on store-load pairs, which dynamically identify the store-load pairs and assign a synchronization mechanism, and (4) evaluate the performance of these different types of memory dependence predictors.

The rest of this paper is organized as following. Different types of memory dependence predictors would be illustrated and compared with other alternatives in Section II. Then in Section III we analyzed our simulation results to demonstrate the performance of the memory dependence predictors, where the implantation of the approach that will not involve in the memory dependence prediction would be considered as the basic model and other implementations making effort on memory dependence prediction would be focused on to analyze its performance and the potential probability to make the improvement. Finally, we would make the conclusion of the store-load pair model and the further work on memory dependence predictors.

II. Memory Dependence Predictor

To measure the importance of memory dependence predictors, there are two basic approaches, not involved in any prediction of memory dependencies. Two main sources of performance loss are false dependences and memory-order violations, when compared with a perfect memory dependence predictors. Then following illustration of the models of store set and store-load pair are explained in details about their mechanism and the benefits in the execution of out-of-order programs. Store sets are already implemented in O3 CPU, which are considered as the reference model in this paper to make the comparison with our implementation of the store-load pair model.

2.1 Perfect Memory Dependence prediction

The ideal model of the perfect memory dependence predictor does not cause any memory-order violations and not impose false dependencies either, which would not be achieved in practice. We only use the perfect memory dependence predictor to analyze the advantages and disadvantages of other types of memory dependence predictors based on the performance comparison.

With the analysis of the performance with different implementation under our simulator, the conclusion of the main reason lead to the performance degradation of above two approaches. The performance of *No speculation* policy would be severely impacted because loads are not allowed to speculate around stores. On the other hand, the observation of the naive speculation proves that frequently squash and re-execution of instructions would be responsible for the poor performance, provided the ideal simulation results from the perfect memory dependence predictors.

2.2 Naive speculation (no speculation)

No memory dependence predictor is used for naive speculation. It allows the load instructions to execute out-of-order, committing memory-order violations. Thus naive predictor is just implemented as to execute loads when their register dependencies were ready, independent of their memory dependencies.

The advantage of naive predictor is, obviously, no false dependence. However, it incurs great performance penalty on memory-order violations, called memory trap penalty, a term as the number of cycle between the first time when a load is fetched to the next time the load fetched after a memory-order violation is detected [5]. Although the memory trap penalty can be reduced by re-executing only the load instruction and its dependent instructions [6], memory-order violations still negatively impact performance by occupying the entries in instruction queue and thus reducing the instruction level parallelism.

2.3 Store-set

Store sets are mainly based upon two assumptions: the first is that the historic behavior of memory-order violation can be well learned to predict the future memory dependencies and the second that the memory dependence is limited in the scope of one store corresponding to multiple stores and multiple dependent loads on the same store.

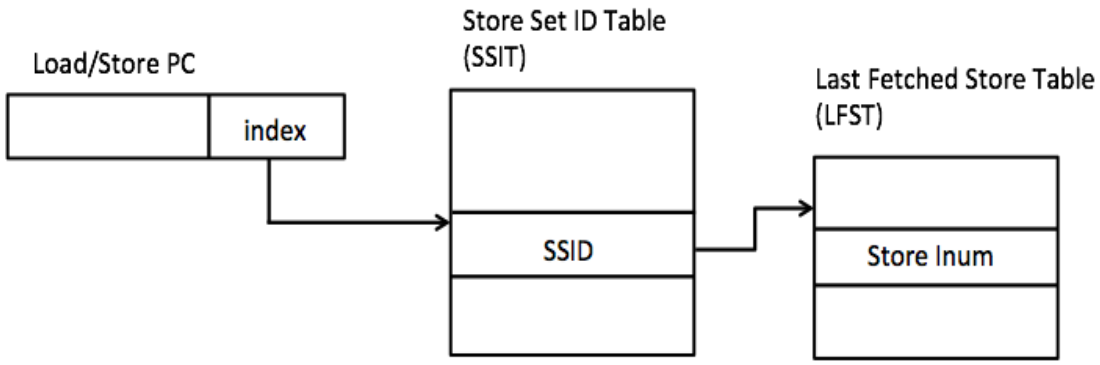


Fig.1 Implementation of store sets memory dependence prediction

As the Fig1 shown above, store sets are indexed by instruction PC. At the start, all the entries of *Store Set Identification Table* (SSIT) is invalid. If the memory-order violation is committed, the belonging store set would be created in the SSIT with allocated *Store Set ID* (SSID). SSIT would add the coming store and load instructions to its corresponding store set by SSID, which is used to access and update the entry in *Last Fetched Store Table* (LFST). LFST is a table that maintains the dynamic information about the most recently fetched store for each store set. Store inum in LFST uniquely identifies the instance of each instruction in flight.

If the coming instruction is the load, it would get a valid store set with the corresponding valid SSID. Then it will access the LFST and fetch the inum of the most recently fetched store instruction to decide whether the instruction can be issued with the resolved memory dependence.

Considering another situation of the coming store, its found valid SSID will lead the store instruction to its valid store set. The dependence would be forced between the new store and the store instructions found in LFST. Obviously, the information about the last recently fetched store should be updated with the new table and the coming store will insert its own inum into the LFST.

The specific mechanism is required to solve the conflicts when the relationship between dependent load and store instructions are not one-to-one. When memory-order violation occurs, if neither SSID of the load and store instructions is valid, the new store set will be created and then assigned to both. In another situation where the load's SSID is valid and the store's SSID is not, the store will inherit the SSID from the load and become part of the load's store set. Otherwise, overwriting the store's old SSID with the new one is proved to be efficient to implement the store set. However, this rule inherently limits the number of store sets to which one store instruction could belong, which may create new possibilities for memory-order violations.

Another situation that many loads depend on the same store is fairly common, which will occur when there is one writer and multiple reader of the value store in a certain memory location. There are several situations may happen when we consider about the dependency between loads and the corresponding stores. (1) A load can depend stores on different paths. (2) A load can dependent on multiple stores to fields of a structure packed in a data word that are all read together. (3) Assumption is made that memory write-after-write hazard are also treated as dependencies so that a load can depend on a series of stores to the same location. Due to different types of situation the implementation would encounter, the implementation of store sets in gem5 platform involves in several configurations. The first one is that the store set can contain as many stores as necessary regardless of its finite size in practice, but a store can only reside in one store set. If many memory-order violations can arise with one store, it only belongs to the store set of the load which it conflicts last. This configuration will assist in prohibiting multiple loads dependent on the same store. There is another configuration that the size of a store set is limited to one, allowing each load in the program to specify at most one store dependence, which prohibits one load from being dependent on multiple stores. Thus when a memory-order violation occurs, the store replaces any prior store in the load's store set.

The store set is already implemented in O3 CPU as the memory dependence predictors, so we need not to make effort on its implementation. We use the store set as the reference model to compare its performance and analyze its advantages with our implementation of store-load pairs.

2.4 Store-load pair

The store-load pair speculation is to dynamically identify the store-load pairs that are likely to be data dependent and then synchronize the instruction pairs [7]. For the first part, past history is used to identify and track the store-load pairs. For synchronization, a condition variable is used to build the association between the

store-load pair, which indicates that the load with true dependences can only be executed after the stores signal it.

Based the mechanism discussed above, the store-load pair predictor can be implemented with two tables, the hardware structures that cache the information for history and synchronization. Memory dependence prediction table (MDPT) whose fields consist of valid bit, load PC and store PC, contains the history of mis-speculations. It is used to identify the store-load pairs, each entry predicting that the load instruction is likely to depend on the store instructions. Considering the limited size of MDPT and the number of mis-speculation pairs, some replacement policy should be applied and thus random replacement policy is chosen in our study.

Memory dependence synchronization table (MDST) is the other table for the implementation of store-load pair predictor. It provides a condition variable to synchronize the instruction pair that is detected by the MDPT. Specifically, the fields in MDST include valid bit, load PC, store PC, load ID, store ID, tag and empty/full flag. The load ID and store ID are the instruction sequence numbers, used for the return value of the predictor that specifies which instruction should wait and which instruction should issue. The tag is the data address that the memory instruction access, which is important for the mapping between MDPT and MDST. The tag, together with the load PC and store PC, specifies the dependence edge. This dependence edge can distinguish between the different dynamic instances and thus uniquely map the condition variable to dynamic dependences. Finally, the empty/full field provides the function of condition variable, whose value determines whether a load instruction can issue or wait for its dependent stores. Figure2 shows the hardware structures required to support the store-load pair predictor.

(a) (b)

Fig.2 support stuctures for store-load pair predictor implementation

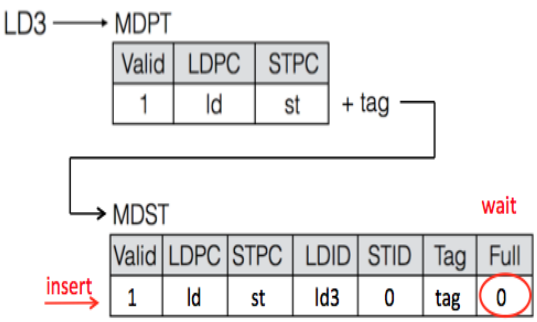
(a) shows the fields of MDPT and (b) shows the fields of MDST

The mechanism of how it works can be illustrated by Figure3. Part (a) shows a simple loop of memory reference instructions, where the load instruction in each loop depends on the store instruction in previous loop. Assume LD2 and ST1 are violated and thus their PCs are cached in the MDPT. For LD3 and ST2, consider two situations based on different memory order.

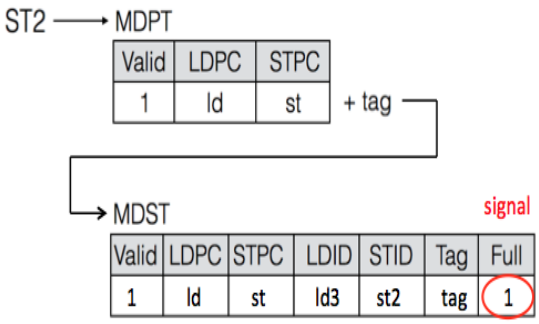
If LD3 accesses the memory before ST2, shown in part (b) and part (c) in Figure3, firstly the PC address of LD3 is used to access the MDPT and matched with an entry. The contents in this entry, namely the load PC and store PC, together with the tag which is the data address of memory access, are then used to search the MDST for synchronization. Since there is no match, an entry is allocated in MDST and the load PC, store PC, load ID and tag are filled in this entry. Meantime, the valid bit is set to one and the empty/full flag is set to zero, which indicates that the load must wait. Then the ST2 accesses the memory and it will set the empty/full flag in the corresponding entry to one, which signals the waiting load. Later, this entry in MDST is released since the synchronization has been completed.

If ST2 executes before LD3, shown in part (d) and part (e) in Figure3, it will allocate an entry in MDST based on the match result in MDPT and set the empty/full flag to one. Then the LD3 accesses the memory, when it find the matched entry in MDST, it will be issued directly since the empty/full flag has already been set to one by its dependent store before. The entry will also be freed after synchronization.

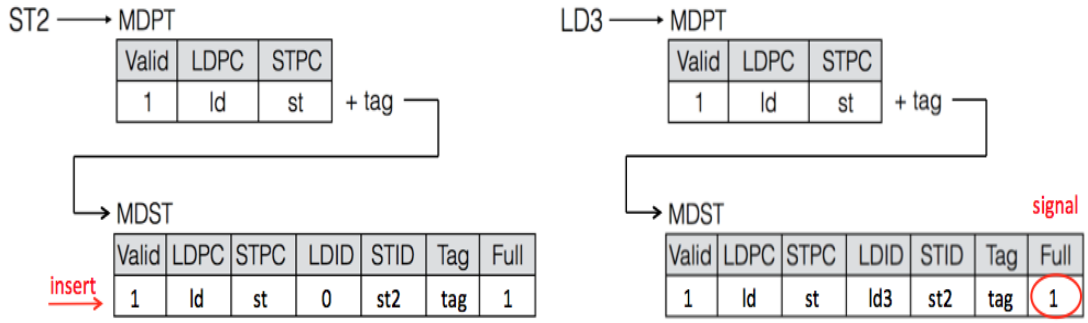
(a)



(b)



(c)



(d) (e)

Fig.3 synchronization of memory dependences for store-load pair predictor

With the distributed hardware of MDPT and MDST to support the memory access, the large pool of memory dependences can be tracked from. However, most each source of memory accesses need only use its local copy of the two tables most of the time. A load instruction would use both tables in the same manner as described earlier. However, on the other hand the store instruction behaves more obscure that after identifying the store in a local MDPT, store needs to broadcast to all copies of the MDST. Each copy of the MDST searches its entries to find any allocated synchronization entry. Any prediction update to the local MDPT should also be broadcast in order to maintain the concurrency between two tables.

III. Evaluation

3.1 Simulation environment

Accurate memory dependence predictions are becoming more and more important with the development of wide-issue out-of-order processors. Our implementation of store-load pair is simulated in gem5 using X86 ISA on the basis of O3 CPU, which is extended from Alpha 21264.

Following Table 1 provides the main parameters of the memory model. The size of the instruction queue is chosen to be 128 entries, on which we deployed the performance sensitivity analysis by increasing its size. For store-load pair predictor, the MDPT size is 64 entries and the MDST size is 128 entries.

Size of instruction queue	128 entries
Instruction cache	128K 2-way set associative
Data cache	128K 2-way set associative write-back
Data cache port number	4
maximum instructions issued per cycle	8
Unified second level cache	8M direct mapped. write-back ß

Table 1. The CPU model

We used the SPEC2006 benchmark as a test suite to simulate different types of memory dependence predictors. To avoid the long simulation times of some programs, we ran our simulations until the retired instructions reach 100,000,000.

3.2 Evaluation on different memory dependence predictors

Figure 4 and Figure 5 show the results of three types of memory dependence predictors: naive, store-load pairs, and store sets with several chosen test programs including perlbench, bzip2, gcc, mcf, and sjeng (shown in table 2). We use three different metrics, latency, IPC and the number of memory references, to compare the performance of different memory dependence predictors. The number of memory references can reflect how many memory-order violations happen during execution and thus is an important metric for evaluating memory dependence predictors.

The performance results show that the simulation is benchmark dependent, so table 2 provides the information about the five test programs we have tested on. The language, application area and brief description are listed in the table.

Unfortunately, both Figure4 and Figure5 leads to the same conclusion that the store set has the best performance and then the store-load pair, compared with the base model of naive implementation. Our implemented store-load pair is only slightly better than na ïve, especially for the bzip2 and mcf programs, where

their run time and IPC are very close but much worse than store sets. We have dig into the codes to check whether little performance improvement has been made with this program.

Benchmark	Language	Application area	Brief description
400.perlbench	C	Programming Language	Derived from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).
401.bzip2	C	Compression	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.
403.gcc	C	C Compiler	Based on gcc Version 3.2, generates code for Opteron.
429.mcf	C	Combinatorial Optimization	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
458.sjeng	C	Artificial Intelligence: chess	A highly-ranked chess program that also plays several chess variants.

Table 2. Benchmark descriptions

Figure 6 shows the number of memory references, which implies the number of memory-order violations. We can reach the conclusion that store sets predictor eliminate much more memory-order violations, which is one of the important reasons why it reveals better performance. However, our implemented store-load pair does not avoid many violations as expected. When digging into our implementation, the main reason about this poor performance is the way of synchronization, which requires further improving.

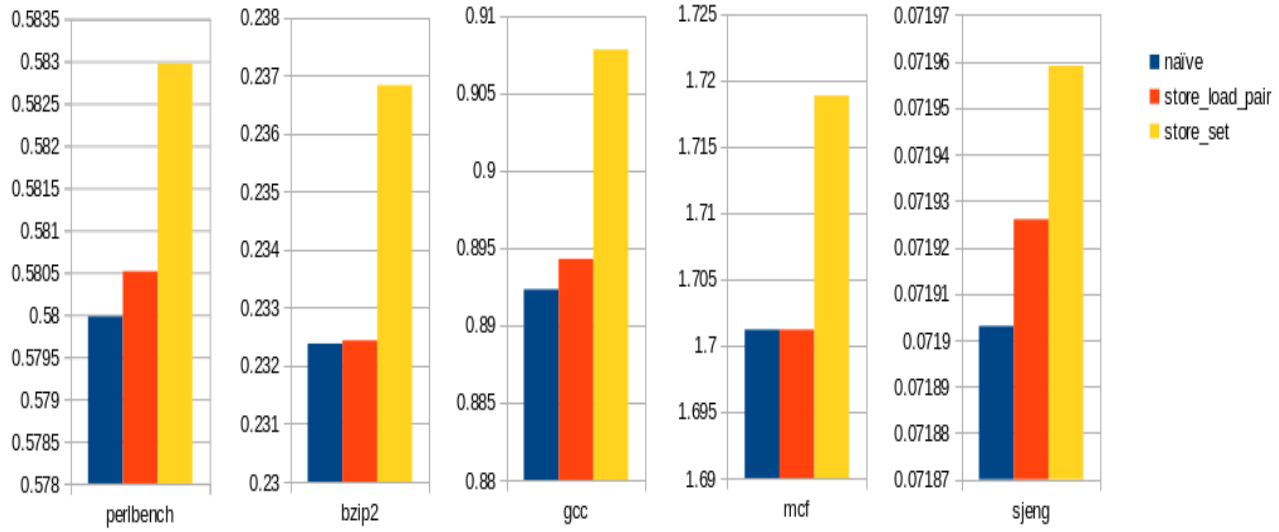


Fig.4 The SPEC2006 latency performance of three memory dependency

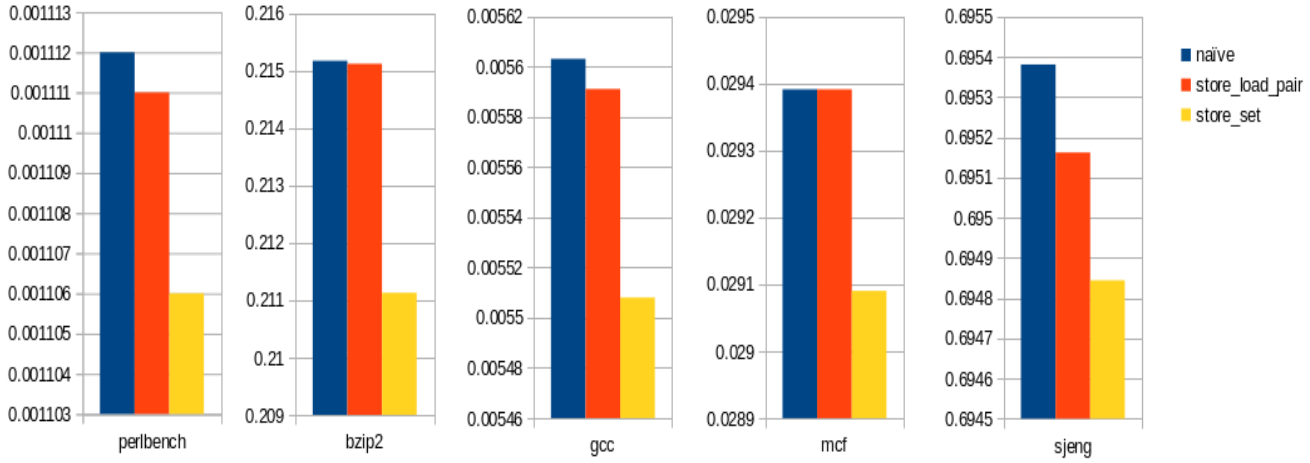


Fig.5 The SPEC2006 IPC performance of three memory dependency

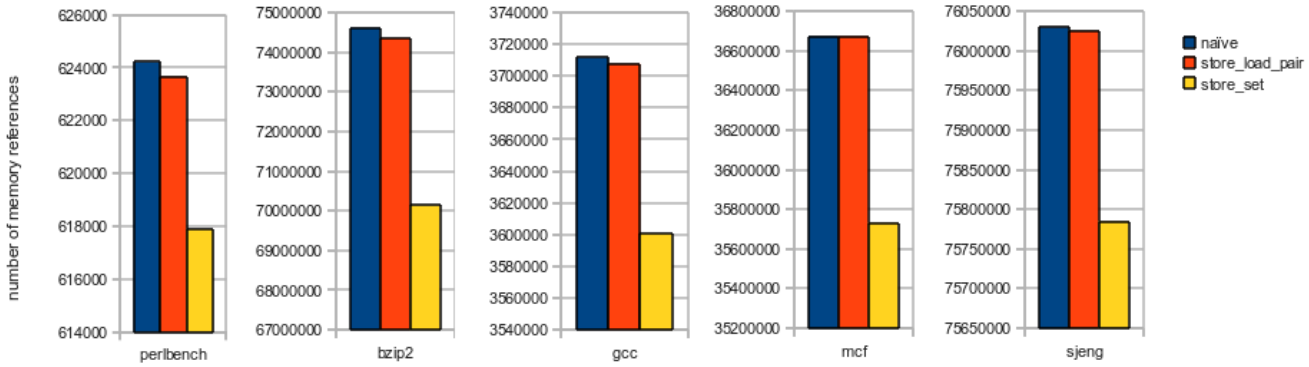


Fig.6 The number of memory references of three memory dependency

3.3 Evaluation on the effect of window size for store-load pair predictor

It is mentioned that significant performance benefits will be achieved when instruction windows get larger in [3]. Therefore, we simulated the store-load pair predictor with different instruction window size to demonstrate this feature. The instruction window size in O3 CPU is the number of reorder buffer (ROB) entries. We make the number of ROB entries grow from 16 to 256, by power of two.

Figure7 and Figure8 show the latency and IPC for the different window sizes. We can conclude that the performance is improved obviously with larger window size since larger window size exposes more memory dependences. But the bzip2 program is an exception, the performance bounds back if the window size grows to 128 entries. In general, larger window size brings better performance and the performance would become flat at some threshold value which is workload dependent.

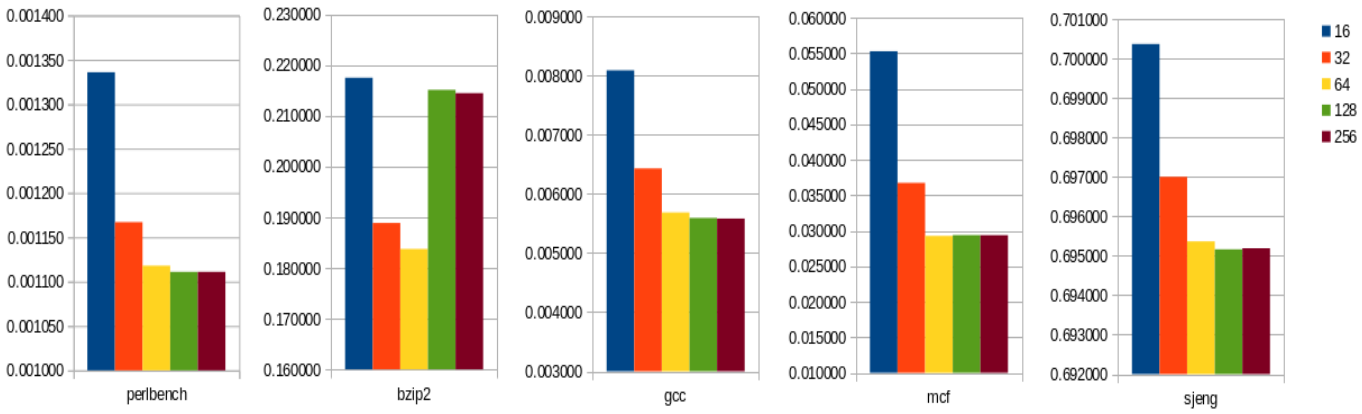


Fig.7 The SPEC2006 IPC performance of three memory dependency

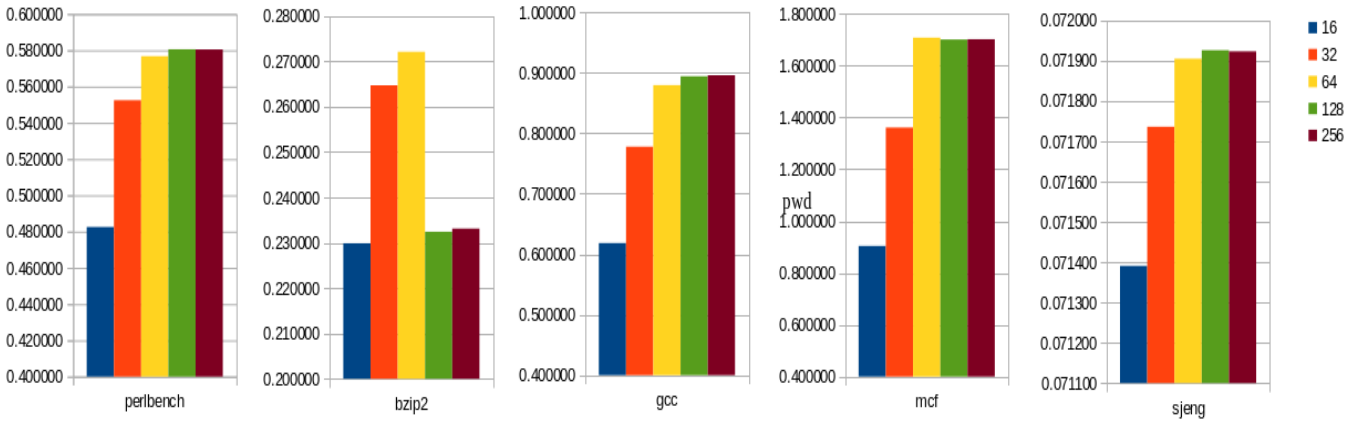


Fig.8 The SPEC2006 IPC performance of three memory dependency

IV. Issues

For our implementation of store-load pair predictor, the evaluation results do not show the better performance over store sets predictor, which is abnormal. The main issues are due to the synchronization part.

4.1 Incomplete synchronization

Different from the MDPT, no replacement policy is applied for MDST because the MDST entries would be released after synchronization is completed and because replacing the entries would affect the synchronization, which leads to incomplete synchronization. However, there would still exist incomplete synchronizations even without replacement. This is because MDPT only provide the prediction of dependent store-load pairs based on the history behaviors.

If the history shows that the load instruction is likely to depend on some store instruction, it would create an entry in MDST and set the empty/full flag to zero. But the fact is that this load instruction is independent at this time, what would happen then? Obviously, there would be no store instructions to signal it and thus this situation would cause deadlock problem. Since our simulation is performed on one CPU, we can ignore this problem for now, but it should be carefully handled for multiple CPUs.

If the history shows that the store instruction is depended by some load instruction, it would create an entry in MDST and set the empty/full flag to one. However, the fact is different from the prediction, which indicates that no load instructions would depend on it. In this situation, the MDST entry would never be released since no load instructions would find the entry and the synchronization would never be completed.

4.2 Poor performance due to spinning

Because of our failure to use system calls in the store-load pair implementation, which put the thread to sleep when waiting and wakeup the corresponding thread when signaling, we only use spinning when waiting for the load instruction to be signaled. Obviously, it would occupy the CPU but do nothing, which wastes time and thus have a really bad influence on performance when simulating on one CPU.

V. Conclusion

Memory dependence prediction explores more instruction level parallelism in out-of-order processors. Good prediction will allow the load instructions to execute as early as possible and avoid memory-order violations and thus yield great performance improvement. With the different types of memory dependence predictors mentioned above, it is obvious that the implementation of the memory dependence predictors would greatly improve the performance of out-of-order programs. Once the memory performance is no more the bottleneck limiting the performance of CPU, there would be left great potentiality to make the revolutionary progress in the era of mutiscalars with different levels of parallelism.

Our contributions are (1) implementing the naive predictor as non-speculation base model for O3 CPU, (2) implementing the store-load pair predictor, (3) evaluating the different performance for naive, store sets and store-load pair predictors, and (4) demonstrating the performance improvement of store-load pair predictor for larger window size.

The drawbacks include (1) lack of another non-speculation model which contains lots of false dependences with no memory-order violations, (2) using spining for synchronization which negatively affect performance, and (3) lack of metric such as the number of false dependent instructions and the number of memory-order violated instructions when analyzing and comparing the different memory dependence predictors.

Further work may invlove (1) implementing the synchronization using system calls, (2) evaluating the predictors based on more characteristic metris in addition to latency and IPC, and (3) figuring out the pattern for each predictor where the predictor would have better performance or worse performance.

References

[1] S. Steely, D. Sager, and D. Fite. Memory reference tagging. US. Patent 5619662 Filed. Aug. 1994, Issued Apr. 1997.

[2] J. Hesson, J. LeBlanc, and S. Ciavaglia. Apparatus to dynamically control the out-of-order execution of load-store instructions. US. Patent 5615350 Filed Dec. 1995, Issued Mar. 1997.

[3] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization and synchronization of data dependences. ACM SIGARCH Computer Architecture News. ACM, 1997, 25(2): 181-193.

[4] A. Moshovos and G. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In Proc. MICRO-30. IEEE Computer Society, 1997: 235-245.

[5] P. Petersen and D. Padua. Static and dynamic evaluation of data dependence analysis. Proceedings of the 7th international conference on Supercomputing. ACM, 1993: 107-106.

[6] G. Chrysos and J. Emer. Memory dependence predcition usint store sets. ACM SIGARCH Computer Architecture News. IEEE Computer Society, 1998, 26(3): 142-153.

[7] G. Tyson, T. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In Proc. MICRO-30, Dec. 1997.

[8] Pflücker López, Otto Fernando. Memory Dependence Prediction Methods Study and Improvement Proposals. 2011.