Dynamic Memory Disambiguation in the Presence of Out-of-order Store Issuing *

Soner Onder
Department of Computer Science
Michigan Technological University
Houghton, MI 49931-1295

Rajiv Gupta
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Abstract

With the help of the memory dependence predictor the instruction scheduler can speculatively issue load instructions at the earliest possible time without causing significant amounts of memory order violations. For maximum performance, the scheduler must also allow full out-of-order issuing of store instructions since any superfluous ordering of stores results in **false memory dependencies** *which adversely affect the timely issuing of dependent loads. Unfortunately, simple techniques of detecting memory order violations do not work well when store instructions issue out-of-order since they yield many* **false memory order violations**. *By using a novel memory order violation detection mechanism that is employed in the retire logic of the processor and delaying the checking for memory order violations, we are able to allow full out-of-order issuing of store instructions without causing false memory order violations. In addition, our mechanism can take advantage of data value redundancy. We present an implementation of our technique using the store set memory dependence predictor. An out-of-order superscalar processor that uses our technique delivers an IPC which is within 100, 96 and 85 % of a processor equipped with an ideal memory disambiguator at issue widths of 8, 16 and 32 instructions respectively.*

1 Introduction

To achieve high performance, superscalar processors must issue load instructions as early as possible while causing few memory order violations. One way to accomplish this task is to use a memory dependence predictor to guide instruction scheduling. Based upon previously observed load/store dependencies, a dynamic memory dependence predictor guides the instruction scheduler to selectively issue load instructions earlier than prior unissued store instructions in the instruction window. Work in this area has produced increasingly better results [2, 3, 8, 7, 11]. Memory disambiguation and communication through memory has been studied extensively by Moshovos and Sohi [7]. Dynamic memory disambiguators proposed mainly used associative structures aiming to precisely identify the load/store pairs involved in the communication. Chrysos and Emer [2] introduced the store set concept which allowed using direct mapped structures without explicitly aiming to identify the load/store pairs precisely, yielding much more efficient utilization of the predictor space. Although these techniques aim to efficiently predict memory dependencies, reducing the *false memory dependencies* remains a significant problem.

False memory dependencies may be imposed by memory dependence predictors when changes occur in the dependency behavior of the program as it executes, aliasing in the predictor tables, or because store instructions are not allowed to issue fully out-of-order. We refer to the latter as *store-store induced* dependencies and focus on avoiding them. Introduction of such false memory dependencies by memory dependence predictors is not without a reason. Simple memory violation detection schemes yield *false memory order violations* when store instructions are allowed to issue fully out-of-order since they cannot correctly detect the memory order violations in such a setting [2].

In this paper, we experimentally demonstrate that ordering store instructions in the issue window yields low performance at high issue widths. We present a simple and effective scheme for the detection of memory order violations that allows full out-of-order issuing of store instructions in the instruction window. Our technique works by delaying the checking for memory order violations and identifying the dependencies in order

*This work is supported by NSF grants CCR-9704350, CCR-9808590, EIA-9806525, and a grant from Intel Corporation.

during retire time. Since no ordering of store instructions in the instruction window is imposed, no superfluous ordering of dependent loads occur even when we use a predictor that relies only on the program counter values of the load and store instructions. As a result, our approach significantly reduces the false memory dependencies. We apply our solution to the highly successful *store set memory disambiguator* [2] and demonstrate that with a simple modification to the predictor algorithm and using our processor back-end, we can greatly enhance the performance of the algorithm at high issue widths. Furthermore, our approach can take advantage of the value redundancy of store instructions to the same memory locations, achieving even greater degrees of instruction level parallelism.

In the remainder of the paper we first demonstrate that the effect of false memory dependencies increases significantly as the issue width of the machine is increased. Next we present and evaluate our novel memory order violation detection algorithm that eliminates false memory order violations completely, and reduces false dependencies significantly. Finally we conclude with a brief discussion of related work.

2 Store Set Algorithm: Performance Evaluation and Analysis

The store-set algorithm [2] relies on the fact that the future memory dependencies can often be correctly identified from the history of memory order violations. In this respect, a *store set* is defined to be the set of store instructions a load has ever become dependent. The algorithm starts with empty sets, and speculates load instructions around stores blindly. When memory order violations are detected, offending store and the load instructions are allocated store sets and placed in them. In general a load may depend upon multiple stores and multiple loads may depend on a single store. In order to obtain a simple implementation, Chrysos and Emer limit a store to be in at most one store set at a time as well as the total number of loads that can have their own store set. Furthermore, stores within a store set are constrained to execute in order. With these simplifications, only two directly mapped structures shown in Figure 1 are needed to implement the desired functionality.

When new load and store instructions are fetched, they access the store set id table (SSIT) to fetch their store set identifiers (SSIDs). If the load/store has a valid SSID, it belongs to a store set. Store instructions access the last fetched store table (LFST) to obtain a hardware pointer to the last store instruction that is a member of the same store set which was fetched before

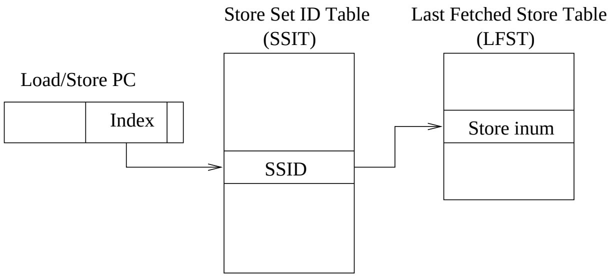


Figure 1. Store Set Implementation

the current store instruction. Current store instruction is made dependent on this store. Next, recently fetched store instruction puts its own id, that is, a hardware pointer to itself, into the table. Similarly, load instructions are made dependent upon the store instruction whose id is found in the LFST table. The algorithm orders stores within a store set in program order, but allows multiple loads to be dependent on a single store.

The Evaluation. We have implemented the store set algorithm as well as a base processor that implements an "ideal" memory disambiguator using the ADL language [9] and simulated the SPEC95 benchmarks at various issue widths. The ideal memory disambiguator relies on program traces to identify the provider store instruction instance for each of the load instruction instances. Hence, for a given load, the ideal disambiguator indicates whether or not the store instruction on which the load is truly dependent has been issued.

Issue width	8, 16, 32 instructions
Fetch width	Equal to the issue width
Retire width	Twice the issue width
Instruction Window	Issue width ** 2
Functional Units	Issue width Symmetric Functional units.
Instruction fetch	Perfect
Dcache	Perfect
Memory ports	Issue width / 2

(a) Machine parameters

Functional Unit	Latency (cycles)
Load	2
Integer division	8
Integer multiply	4
Other integer	1
Float multiply	4
Float addition	3
Float division	8
Other float	2

(b) Functional Unit latencies

Figure 2. Machine Configurations.

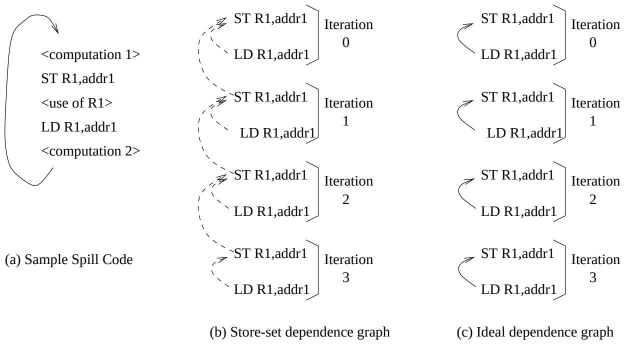
In order to evaluate the performance of the memory disambiguator, we kept the machines with ideal and store set disambiguators identical in all other aspects except the memory disambiguator. Both superscalar processors employ an ideal instruction fetcher that delivers issue width instructions every cycle. Similarly, the issue window is a large central window implementation which can schedule instructions as soon as the data dependencies for an instruction are satisfied. In order to show the effects of the predictor table size on the performance, we consider 4K as well as 64K entry tables. Other machine parameters used in the simulations are shown in Figure 2.

Our results for an 8 issue machine confirm the published performance of the store set algorithm. We observed that all benchmarks show performance losses compared to the ideal disambiguator with the exception of `107.mgrid` and `145.fpppp`. With 4K tables, benchmarks `110.applu`, `141.apsi`, `145.fpppp` and `146.wave5` demonstrate significant performance losses. However, with 64K tables the algorithm closely matches the performance of the ideal disambiguator. On the average, the store set algorithm achieves over 97% of the performance of the ideal disambiguator for floating point benchmarks and over 98% for the integer benchmarks with 64K tables. With 4K tables, the corresponding values drop to 80% for floating point benchmarks and 96% for integer benchmarks.

When we simulated the same machine configurations for the issue widths of 16 and 32, we observed that performance loss as a result of non-ideal memory disambiguation becomes quite significant. Among the integer benchmarks, both `126.gcc` and `147.vortex` show significant performance losses at an issue width of 16 and above and only `130.li` continues to perform well as the issue width is increased. A similar behavior is observed among the floating point benchmarks. With the exception of `107.mgrid` and `125.turb3d`, all benchmarks indicate significant performance losses compared to an ideal disambiguator. At an issue width of 16 and 64K tables, store set can achieve about 85% and 82% of the performance of the ideal disambiguator for integer and floating point benchmarks respectively. With 4K tables, the algorithm can achieve about 69% of the ideal performance for integer benchmarks and 61% for floating point benchmarks. At an issue width of 32 and 64K tables, performance drops further to 67% and 64%. These results indicate that there is a significant room for improvement, especially at issue widths of 16 and above.

The Analysis. At high issue widths, the algorithm experiences performance losses because it forces the issuing of store instructions within a store set to be in-order. In-order issuing of the stores within a store set in turn causes dependent loads to issue in-order. While this restriction is not significant for a wide range of cases, it creates significant degrees of false memory dependencies with two types of loops.

One of them is the case where certain iterations of a loop occasionally become dependent on another iteration as in the case of `110.applu`. The other involves loops with register spill code. In both cases, the algorithm serializes loop iterations once appropriate store set entries are created since the algorithm cannot distinguish between multiple instances of the same load and store instructions. In this case, all the instances of the same store instruction become members of the same set and are forced to issue in-order. Limitations of the algorithm become more pronounced at high issue widths because at small issue widths there is still ample amount of unexploited parallelism to hide the effects of serialization. However, the effects of the serialization cannot be hidden by other available instructions at high issue widths.



Cycle	Instructions	
0	ST-0	
1	LD-0	ST-1
2	LD-1	ST-2
3	LD-2	ST-3
4	LD-3	...

(d) Store-set schedule

Cycle	Instructions			
0	ST-0	ST-1	ST-2	ST-3
1	LD-0	LD-1	LD-2	LD-3

(e) Ideal schedule

Figure 3. Example spill code and its schedule

Let us now examine in detail how the algorithm executes such loops. In Figure 3(a), an example loop that contains spill code is illustrated. When such a code sequence is executed using sufficient resources to issue more than one load operation per cycle, it takes only a few iterations to be unrolled before a memory order violation occurs. This is because, when there is no dependency information stored in the tables, any of the loads can issue once they compute their addresses. As a result, any load which is truly dependent on the store at the same iteration may issue before that store. Once this happens, a memory order violation is detected and the store set entries are allocated. From this point on, following instances of these loads and stores share the information stored in SSIT and LFST which yields the dependence graph shown in Figure 3(b). The algorithm correctly makes a load dependent on the proper store by means of the LFST table. However, since the algorithm forces stores within the same store set to issue in order, for the given set of loads and stores the generated schedule allows at most one load instruction to execute per cycle (see Figure 3(d)). In contrast, an ideal disambiguator would allow fully parallel operation of the multiple instances of the loop body, given sufficient resources (see Figure 3(e)).

3 Out-of-order Store Issue Algorithm

We have shown that for high performance we need to allow full out-of-order issuing of store instructions so that dependent load instructions can also issue fully out-of-order. Unfortunately, without at least a partial ordering of store instructions in the instruction window the algorithm would have suffered much more significant levels of performance losses because of *false memory order violations*.

False memory order violations. A simple mechanism for memory violation detection checks load addresses of the speculatively issued loads when a store instruction is issued and upon an address match raises a memory order violation condition. In this approach false memory order violations occur because this mechanism cannot decisively figure out the set of store instructions which should participate in the memory order violation detection process for a given load.

To illustrate why this is the case, let us reconsider the example shown in Figure 3 and suppose that we remove the dependence edges between the store instructions when we are using the store set disambiguator, allowing store instructions to issue fully out-of-order. In this case, a store belonging to an earlier iteration may be blocked whereas a store belonging to a later iteration may have a chance to proceed. For example, in Figure 3(c) the store from the second iteration, ST-2, may proceed before ST-1 which belongs to the first iteration. When ST-2 issues it makes its dependent load LD-2 eligible to issue in the next cycle. When LD-2 issues, it gets the *correct value* from the forwarding buffer. The processor however remembers that the load has been issued speculatively and makes an entry regarding this load in the *speculative loads table*. When the store ST-1 issues, it finds that a load with a sequence number greater than its own that computed the same address has issued before the store. In this case, an exception is flagged which is in fact a *false memory order violation*. In other words, removing the store ordering in the instruction window would convert all store ordering induced false memory dependencies to false memory order violations.

Precisely detecting memory order violations. In order to detect memory order violations correctly when store instructions are allowed to issue freely, we need to identify precisely the set of store instructions which should participate in the memory order violation detection process. If an issuing store instruction is not the member of this set with respect to a given speculatively issued load instruction, we should not let this particular store instruction raise a memory order violation for the load instruction in question.

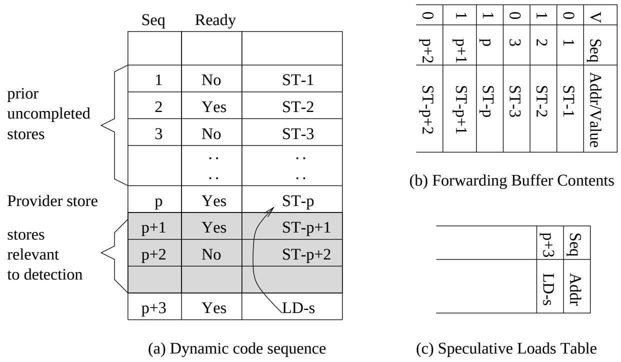


Figure 4. Speculative issuing of loads

In order to see how we can identify the set of store instructions which must participate in the decision process, let us consider the sequence of store instructions and the load instruction shown in Figure 4(a). Given this dynamic sequence of instructions, assume that the load LD-s is predicted to be dependent on the store ST-p which is indicated through the dependence edge. In this configuration, store instructions which are above the provider store instruction ST-p should not participate in the memory order violation detection process for the speculatively issued load instruction LD-s assuming that the addresses of ST-p and LD-s match. Only store instructions which follow the provider store instruction, namely, ST-p+1 and ST-p+2 should raise an exception if they compute an effective address that is the same as the load LD-s. Thus, the provider store instruction splits the set of uncompleted stores into two sets, and only the ones that follow the provider store instruction should participate in the decision process. In the case that the load instruction obtains its value from the memory, all the prior stores should be involved for checking the memory order violations with respect to the speculatively issued load LD-s.

One possible solution in this case is to include the sequence number of the provider store (or a special identifier if it is memory) in the speculative loads table. In this case, issuing store instructions may check their sequence numbers against the sequence number of the provider as well as the sequence number of the load instruction to decide that if they fall into the shaded region in Figure 4(a). If that is the case and the address generated by the store instruction matches to that of the load, an exception may be raised.

A straightforward implementation of the above mechanism leads to a complex piece of hardware due to the following reasons. First of all, maintaining explicit temporal information through sequence counters is not a trivial task because counters must be of finite size and when they overflow, the boundary conditions must be properly handled. Second, for any given store instruction the processor must execute the above algorithm in parallel against all the speculatively issued load instructions, which means that the required hardware must be replicated. Finally, executing the above algorithm on the critical path of the processor is very likely to slow down the processor clock. Next, we present a simple and effective solution.

Delayed exception handling and value matching. Our solution to the detection problem builds on two observations. First, the temporal information needed is implicitly available during the retire phase of the instruction execution. In other words, if we can delay the detection of the memory order violations to the retire phase, we do not need to maintain the temporal information explicitly. Once we move the detection logic to the retire phase, memory violation detection entails deciding whether or not the provider store has retired. If that is the case, following store instructions are from the shaded region and they should check for memory exceptions. Of course, if the provider is the memory, the provider store has already retired, and in this case all retiring store instructions will be involved in the checking. Second, for correctness, we do not need to identify the exact store instruction that provided the value. We only need to verify that given a set of store instructions, the load has obtained the same *value* as the value stored by the last store instruction to the same memory address. This technique eliminates the need for special handling of the case where memory is the provider. Furthermore, this method can take advantage of the data redundancy available in the programs. Given these observations, we can now devise the following scheme that works quite well:

1. We delay the checking for exceptions in case of memory references until the store retires.

2. We expand the *speculative loads table* to contain a value field where the value the load instruction has obtained is stored.

3. We allow an exception bit associated with the load instructions stored in the reorder buffer or in speculative loads table be set or reset by store instructions as they retire. In other words, each retiring store instruction compares the value it has stored, as well as the address into which the data has been stored, to with that of the speculative loads:

   If the addresses match and values differ, it sets the exception bit associated with the load.

   If the addresses match and values match, it resets the exception bit associated with the load.

   If the addresses do not match, no action is taken.

4. Once the load is ready to retire, it checks its exception bit. If the bit is set, a roll-back is initiated and the fetch starts with the excepting load instruction. Otherwise, the load instruction's entry is deallocated from the speculative loads table.

Note that setting and canceling of exception bits as store instructions retire in this manner handles the problem of identifying the provider store instruction automatically. When the actual provider store instruction retires, both the address and the values will match, and the exception bit is reset. In other words, this instruction will serve as a sentinel signaling the beginning of the group of store instructions which should participate, nullifying the effects of all unrelated prior store instructions.

Now let us reconsider the example shown in Figure 4. Suppose that load LD-s has already been speculatively issued and obtained its value from the store ST-p. Further, assume that the store ST-1 has now been issued and has computed the same address as LD-s. Since ST-1 retires first in program order, it will raise the exception bit associated with the load LD-s. Any of the stores between store ST-1 and store ST-p may take the same action upon an address match and a value mismatch. However, when finally store ST-p retires, it will have both an address and a value match and will reset the exception. When the store ST-p+1 retires, if it computes the same address but the value is different, this is a true exception. The exception will be taken when the load instruction retires. Note that if any of the store instructions ST-p+1 or ST-p+2 in the shaded region have the same address as well as the same value, no exception will be raised and the machine will take advantage of the available data redundancy. The same observation holds for the values coming through memory.

Since we now have an effective solution to the problem of false memory order violations, we can completely eliminate false memory dependencies arising from store-store induced dependencies. To achieve this, we only need to introduce a small change to the original store set algorithm. We let load instructions become dependent on the store instruction they find in the LFST table entry, but we do not chain the store instructions which are members of the same store set, allowing all the store instructions to issue fully out-of-order constrained only by their own register dependencies. Load instructions however wait for the store instruction that they have been predicted to be dependent on. Thus no load instructions are serialized unnecessarily.

We have presented a simple technique that correctly indicates if a memory order violation has occurred by matching the value each retiring store instruction stores with that of the speculatively issued loads. Although there is nothing novel about the common-sense technique of determining correctness of speculated instructions by matching actual data values, the use of this approach in the context of the store set disambiguator is unique and yields high performance beyond what is achievable by an ideal disambiguator that faithfully observes the true memory dependencies.

4 Performance Evaluation

We have fully implemented the algorithm and executed the `SPEC95` benchmarks with their training or test inputs. The processor parameters have been kept as before (see Figure 2).

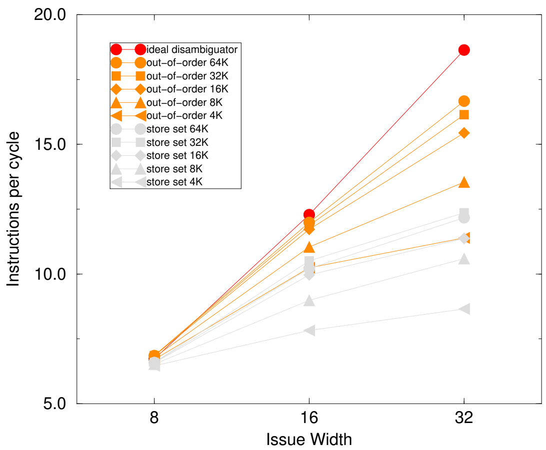
Instructions per cycle. For the measurement of the instructions per cycle figures we compare the harmonic means of the IPC values observed for the floating point and integer benchmarks with 64K tables. At an issue width of 8, the out-of-order algorithm slightly out-performs the ideal memory disambiguator. With floating point benchmarks, out-of-order disambiguator is better than the ideal memory disambiguator by 0.01% and with integer benchmarks it is better than the ideal disambiguator by about 1.8%. The out-of-order algorithm shows better performance than the original store set algorithm by 4% with both the floating point and integer benchmarks. In case of `124.m88ksim`, out-of-order algorithm is better than both techniques by about 18%

When we increase the issue width to 16, the out-of-order algorithm out-performs the original store set algorithm by as much as 18% with integer benchmarks, and 22% with floating point benchmarks The performance difference further widens to 42% and 52% with floating point and integer benchmarks when the issue width is increased to 32 instructions. In both cases, highly data redundant `124.m88ksim` continues to out-perform the ideal disambiguator and the out-of-order disambiguator closely follows the ideal disambiguator for other benchmarks, even at very high issue widths.

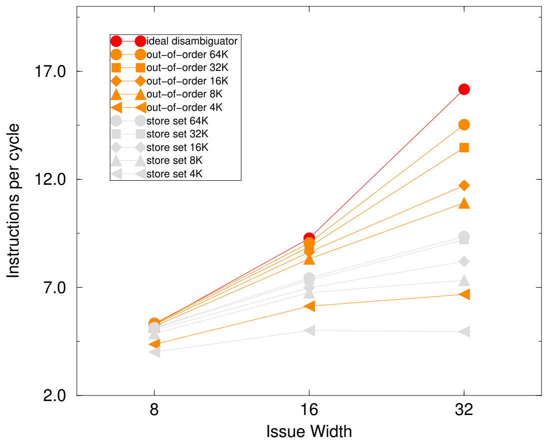
Scalability for different table sizes. In order to further compare the performance of our approach to that of the original algorithm, we executed both algorithms at predictor table sizes of 4K, 8K, 16K, 32K and 64K entries. We observed experimentally that the size of the `LFST` table is not critical and in these runs it was left to be sufficiently large.

As it can be seen from the graphs in Figure 5(a) and Figure 5(b), the original store set algorithm can out-perform our algorithm only when the original store set algorithm has a 32K entry table and the out-of-order disambiguator has a 4K entry table. For both integer and floating point benchmarks, with 8K entries our algorithm always out-performs the original algorithm even when the latter has 64K entry tables. When our algorithm is given a large predictor space, it matches the performance of the ideal disambiguator up to the issue width of 16, and very closely follows the ideal curve with slight performance loss. This small performance loss (about 10 % at an issue width of 32) originates from the cost of restarts. The cost of restarts are largely paid for by the gains we obtain through the exploitation of the data redundancy. Unfortunately, at these high issue widths the exploited value redundancy is not sufficient to compensate for all the restart costs. Nevertheless, it is natural to expect that our algorithm will out-perform the ideal disambiguator at all issue widths if the cost of restarts can be reduced by employing a mechanism which selectively reissues effected instructions instead of squashing a window-full of instructions. Such re-execution recovery is quite feasible with memory order violations. In case of memory order violations, the validity of the instructions are not questioned. Therefore, only a few instructions which uses the wrong value can be reissued instead of throwing away a window-full of instructions.

We observed that when we employ smaller predictor tables, the performance gap between the out-of-order disambiguator and the original store set algorithm widens further, indicating our success in reducing the false memory dependencies. The false memory dependencies were also measured directly to verify this observation. The false memory dependencies are reduced because of two reasons. The first reason is the preciseness of our approach in creating the store sets. When a conventional approach is used to detect the memory order violations, it may take some number of iterations and a number of false memory order violations before the true store instruction that a load is dependent on is discovered. In our case, this happens the first time a violation is encountered. Second, because our approach also takes advantage of value redundancy, it creates fewer number of table entries.



(a) Scalability of integer Benchmarks



(b) Scalability of Floating point Benchmarks

Figure 5. Scalability of Out-of-order Algorithm

5 Conclusion

We have presented an effective mechanism for reducing false memory dependencies that also exploits value redundancy without performing explicit value prediction. Our solution is an orthogonal solution that can be utilized with other types of memory dependence predictors. For example, the scheme proposed by Moshovos and Sohi based on MDPT/MDST associative structures [7] either forces a load to wait for all dependences predicted, or, MDPT entries are augmented to contain control flow information for each load/store pair. Using our scheme, there would not be any need to force a load to wait for all dependences predicted or any need for augmenting predictor entries with control flow information. Similarly, our approach can be used together with value prediction techniques [5, 6, 4, 1]. Specifically, a machine may employ selective value prediction to a subset of loads, whereas the remaining ones would synchronize through the dependence predictor employing our scheme. The verification mechanism our scheme uses would work properly with value prediction mechanisms without modifications.

Reinman and Calder studied performance gains that can be obtained using various predictive techniques for load value speculation including the store set as well as value predictors [10]. Given that in their store-set study store instructions are not allowed to issue out-of-order and our out-of-order disambiguator substantially out performs the original store set algorithm, further studies are needed to verify their conclusion that the value prediction out performs all other techniques.

References

[1] B. Calder, G. Reinman, and D.M. Tullsen. Selective value prediction. In *26th International Conference on Computer Architecture*, pages 64–74, May 1999.

[2] G.Z. Chrysos and J.S. Emer. Memory dependence prediction using store sets. In *25th International Conference on Computer Architecture*, pages 142–153, June 1998.

[3] J. Hesson, J. LeBlanc, and S. Ciavaglia. Apparatus to dynamically control the Out-Of-Order execution of Load-Store instructions. *US. Patent* 5,615,350, Filed Dec. 1995, Issued Mar. 1997.

[4] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A Novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *The 31st Annual IEEE-ACM International Symp. on Microarchitecture*, pages 216–225, Dec. 1998.

[5] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *29th Annual IEEE-ACM International Symposium on Microarchitecture*, 1996.

[6] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[7] A.I. Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin - Madison, 1998.

[8] A.I. Moshovos, S.E. Breach, T.N. Vijaykumar, and G.S. Sohi. Dynamic speculation and synchronization of data dependences. In *24th International Conference on Computer Architecture*, pages 181–193, June 1997.

[9] S. Önder and R. Gupta. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*, pages 80–89, 1998.

[10] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *31st Annual IEEE-ACM International Symp. on Microarchitecture*, Dec. 1998.

[11] S. Steely, D. Sager, and D. Fite. Memory reference tagging. *US. Patent* 5,619,662, Filed Aug. 1994, Issued Apr. 1997.