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

Soner Önder
Dept. of Computer Science
Michigan Technological University
Houghton, MI 49931-1295

Rajiv Gupta
Dept. of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

Out-of-order issue superscalar processors can achieve very high degrees of instruction level parallelism by using a memory dependence predictor to guide dynamic instruction scheduling. With the help of the memory dependence predictor the 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 to eliminating the false dependencies that arise because of store ordering in the instruction window, our mechanism can take advantage of data value redundancy. We present an implementation of our technique using the store set memory dependence predictor and illustrate that our technique improves the performance of the predictor substantially at high issue widths. Our technique reduces the dynamic instances of loads and stores that need a predictor table entry by as much as 12 % and the amount of false dependencies by as much as 42 %. 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.

Keywords: *Memory disambiguation, store-set, wide-issue superscalar, instruction window, speculative execution.*

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

1 Introduction

In order to get high performance, out-of-order superscalar processors must issue load instructions as early as possible without causing memory order violations. One way to accomplish this task is to use a memory dependence predictor to guide instruction scheduling. By caching the previously observed load/store dependencies, a dynamic memory dependence predictor guides the instruction scheduler so that load instructions can be initiated early, even in the presence of a large number of unissued store instructions in the instruction window. Work in this area has produced increasingly better results [3, 8, 7, 2]. The problem of memory disambiguation and the 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. Various patents [11, 3] also aimed at identifying those loads and stores causing memory violations and synchronizing when such loads and stores are encountered. Although these techniques aim to efficiently predict memory dependencies, reducing the *false memory dependencies* remains an area which still has room for improvement.

False memory dependencies may be imposed by memory dependence predictors due to the changes in the dependency behavior of the program as the program executes, aliasing in the predictor tables, or more significantly because store instructions are not allowed to issue fully out-of-order. We refer to this type of dependencies as *store-store induced* dependencies and focus on the false memory dependencies of this type.

Introduction of store-store induced false memory dependencies into the instruction stream 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. It is the complexity of the memory order violation detection with out-of-order issuing of stores that makes memory disambiguators impose at least a partial ordering of store instructions in the issue window [2]. Unfortunately, such ordering of store instructions in the issue window creates false dependency chains which prohibits the dependent load instructions from issuing. When the predictor relies only on the load/store program counter values, the problem becomes a serious problem with loops that contain spill code as well as loops where only a few instances of the loop iteration are actually dependent on another iteration. This is because, such memory dependence predictors cannot distinguish between multiple instances of the same load or store instruction and make such instances dependent on each other. As a result, significant amounts of instruction level parallelism is lost leading to more pronounced performance losses at higher issue widths.

In this paper, we experimentally demonstrate that ordering store instructions in the issue window yields low performance for high issue out-of-order superscalar processors. 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 during retire time. Once the processor is equipped with our algorithm, we can allow unrestricted issuing of store instructions in the instruction window. 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.

The key characteristics of our memory violation detection algorithm are:

1. Our scheme handles the problem of false memory dependencies effectively even when the predictor relies only on load/store program counter values and the store instructions are allowed to issue fully out-of-order.
2. It is a simple scheme that allows out-of-order issuing of store instructions in the instruction window without causing false memory order violations.
3. Finally, our approach takes advantage of value redundancy of stored data values without explicit value prediction.

In the remainder of the paper, in Section 2, we first summarize the store set algorithm and present a thorough analysis of its performance. We illustrate that because the algorithm orders store instructions in the instruction window it suffers from false memory dependencies in loops with spill code and with occasional dependencies. We demonstrate that the effect of false memory dependencies increases significantly as the issue width of the machine is increased. In Section 3, we first discuss the problem of false memory order violations and its relation to the false memory dependencies. Next, we give an analysis of the underlying causes of false memory order violations and present our novel memory order violation detection algorithm that eliminates false memory order violations completely, and reduces false dependencies significantly. In section 4, we present an in depth performance analysis of the proposed solution and finally we conclude with a brief discussion of related work.

2 Store Set Algorithm: Performance Evaluation and Analysis

The store-set algorithm proposed by Chrysos and Emer [2] is a simple and very effective memory disambiguator that relies on the fact that the future memory dependencies can 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 to their respective sets. Since a load may depend upon multiple stores and multiple loads may depend on a single store, an efficient implementation of the concept may be difficult. In order to use direct mapped structures, Chrysos and Emer propose certain simplifying assumptions in their implementation which 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 the current store instruction. Current store instruction is made dependent on this store. Following, 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. As a result, the algorithm orders stores within a store set in program order, but allows multiple loads to be dependent on a single store.

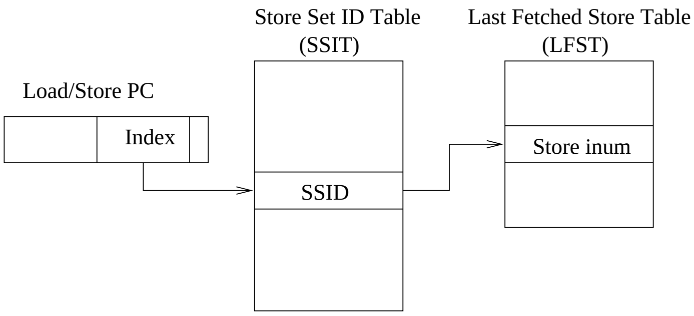


Figure 1: Store Set Implementation

The algorithm is reported to require modest table sizes. This is attributable to high locality of memory dependencies as well as the algorithm's approach to the store set creation. It has been indicated that the algorithm performs superbly using 4K or more SSITs and 128 or more entries of LFST. At an issue width of 8 instructions per cycle, the performance of the algorithm is within few percentages of what can be accomplished using an ideal memory disambiguator which has perfect knowledge of load and store dependencies.

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 identifies 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. The disambiguator uses memory address traces augmented with load/store sequence numbers to identify such dependencies with perfect accuracy.

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

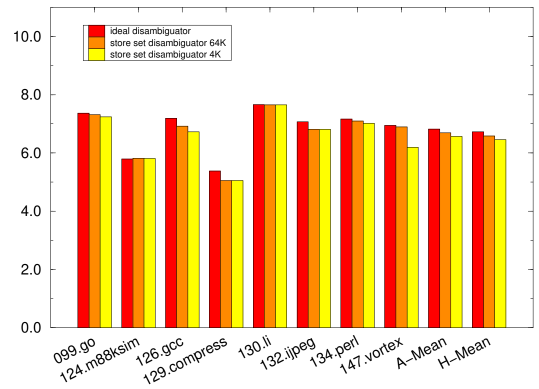
(a) Machine parameters

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

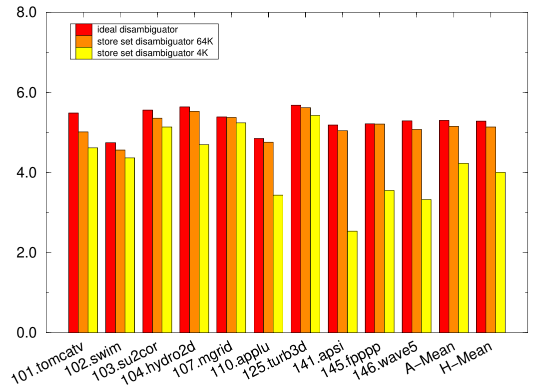
(b) Functional Unit latencies

Figure 2: Machine Configurations.

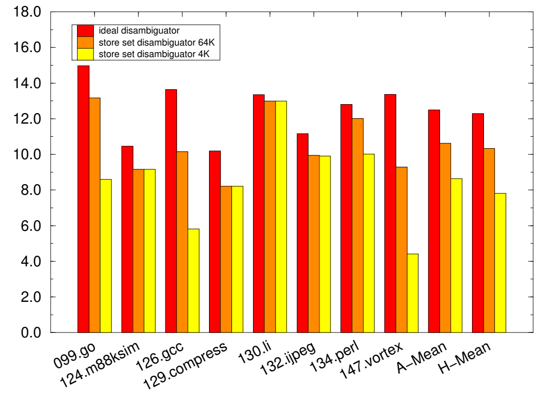
In order to evaluate the performance of the memory disambiguator, we kept the machines with ideal and store set disambiguators identical in all other aspects except the memory disambiguator. Both superscalar processors employ an ideal instruction fetcher that has perfect branch prediction and delivers 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 report the performance of the algorithm at both 4K entry tables as well as 64K tables which experience very few destructive aliasing. Other machine parameters used in the simulations are shown in Figure 2.



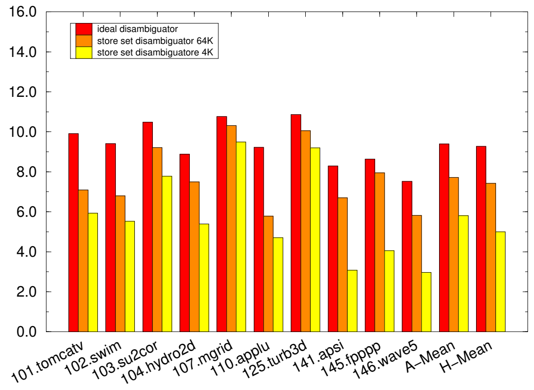
(a) 8 Issue - Integer Benchmarks



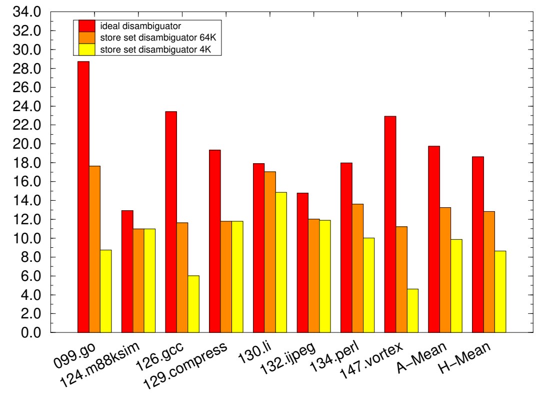
(b) 8 Issue - Floating point Benchmarks



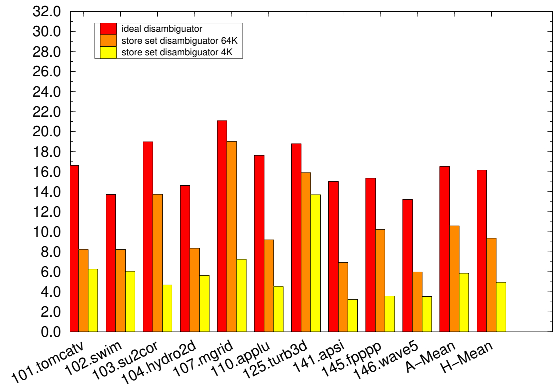
(c) 16 Issue - Integer Benchmarks



(d) 16 Issue - Floating point Benchmarks



(e) 32 Issue - Integer Benchmarks



(f) 32 Issue - Floating point Benchmarks

Figure 3: IPC values Store Set and Ideal cases

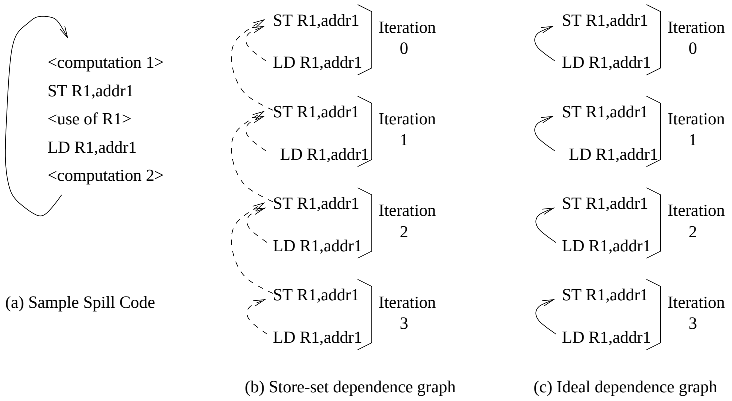
Our results for an 8 issue machine are shown in Figures 3(a) and 3(b). These results confirm the published performance of the store set algorithm with some minor differences. Although most benchmarks with the exception of 110.applu have been reported to perform well in the original study, we observed 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. Differences between our results and the published performance of the store set algorithm are attributable to using an ideal front end as well as using a different ISA (MIPS versus Alpha) and using different compilers (gcc versus DEC cc). 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 (see Figures 3(c), 3(d), 3(e) and 3(f)). 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%. When harmonic means are used, an additional 3 to 4% performance loss is observed with respect to the ideal disambiguator. With 4K tables, the algorithm can provide only 35% of the performance of the ideal disambiguator for floating point benchmarks and 50% for integer benchmarks. These results indicate that there is a significant room for improvement, especially at issue widths of 16 and above.

The Analysis. Although the cost of restart increases as the instruction window is enlarged, this is not the main reason behind the poor performance of the algorithm 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 essentially 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. At large issue widths, the available parallelism in the program is already being fully exploited. Therefore, the effects of the serialization of the operations cannot be hidden by other operations from the pool of available instructions.

Let us now examine in detail how the algorithm executes such loops. In Figure 4(a), an example loop that



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

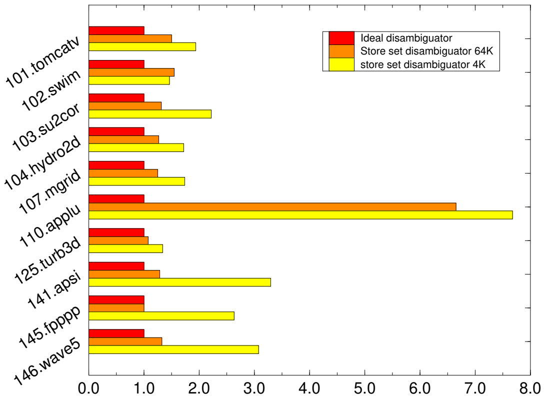
(e) Ideal schedule

Figure 4: Example spill code and its schedule

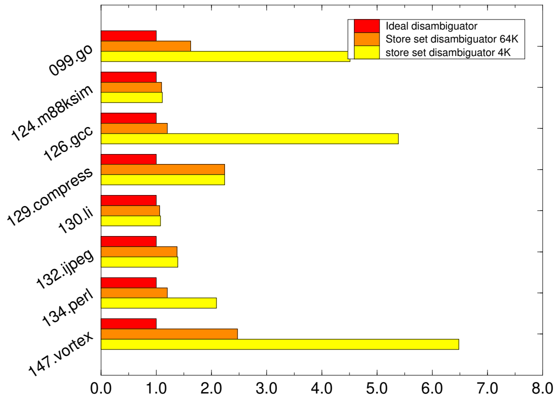
contains spill code is illustrated. When such a code sequence is executed using sufficient resources to issue more than one load operation per cycle, it takes only a few iterations to be unrolled before a memory order violation occurs. This is because, when there is no dependency information stored in 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 4(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 4(d)). In contrast, an ideal disambiguator would allow fully parallel operation of the multiple instances of the loop body, given sufficient resources (see Figure 4(e)).

In order to measure the effect of the serialization on the load latency, we have conducted experiments that studied the dynamic load latency and degree of load serialization. The results of these experiments are as follows:

Dynamic load latency. We measured the number of cycles it takes from the moment a load instruction is ready to issue to the moment it has the loaded value. We define this duration to be the *dynamic load latency*. Dynamic load latency is a function of the dependencies imposed upon load instructions. Therefore, it is a cumulative quantity that includes both the true dependencies of the program as well as scheduling/disambiguation imposed dependencies. To obtain the contribution of the falsely imposed dependencies, we normalized the dynamic load latency values by dividing it with the dynamic load latency value obtained using an ideal memory disambiguator (see Figure 5). In addition to the normalized dynamic load latencies, we have computed the standard deviation of dynamic load latency across all load instructions executed by the benchmark programs both for the store set algorithm and the ideal memory disambiguator. We then normalized the standard deviation values for the store set algorithm by dividing it with the corresponding value of the ideal memory disambiguator (see Figure 6).

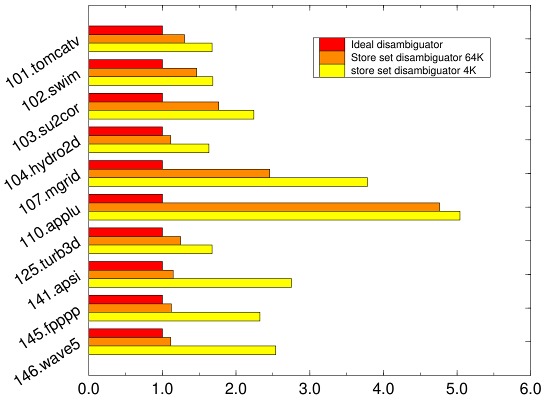


(a) Floating Point Benchmarks

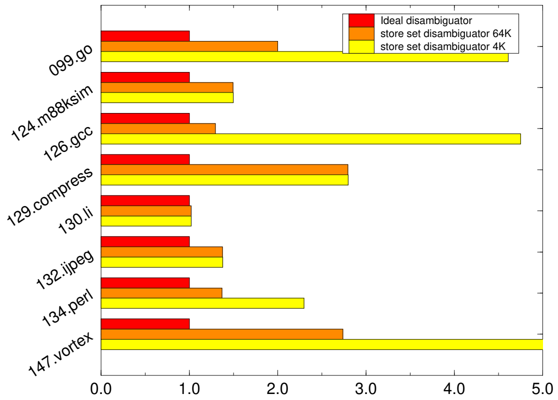


(b) Integer Benchmarks

Figure 5: Normalized Average Dynamic Load Latencies

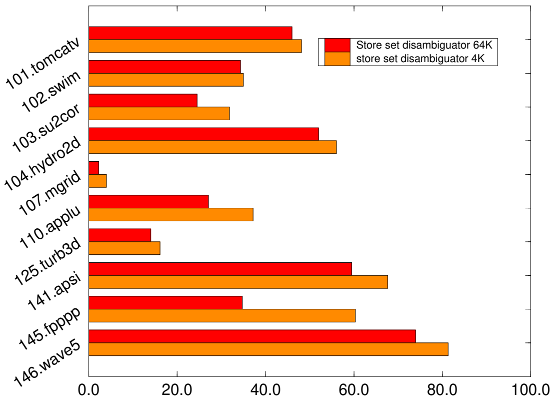


(a) Floating Point Benchmarks

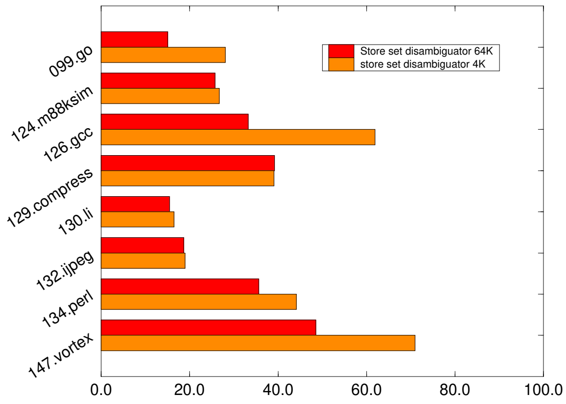


(b) Integer Benchmarks

Figure 6: Normalized Standard Deviation Values for Dynamic Load Latencies



(a) Floating Point Benchmarks



(b) Integer Benchmarks

Figure 7: Percentage of Serialized Load Instructions

As shown in Figure 5, the measurements of the normalized dynamic load latencies exhibit large values for those benchmarks which perform poorly whereas benchmarks which perform well have very small values. Similar behavior is observed in the standard deviation values shown in Figure 6. These results are indicative of long chains of dependent instructions that create large fluctuations in the average load latency. We also observed that the harmonic mean values of the load latencies are uniform across the benchmark spectrum and quite close to the ideal disambiguator. On the other hand, the arithmetic means show great degrees of fluctuation, and they have the worst values for those benchmarks which do not scale.

Load serialization. We determined the degree of the serialization of load instructions through load-store-store dependency chains. We measured the amount of serialization of load instructions by identifying the dynamic load instructions which are blocked from issuing for one or more cycles although their predicted provider store instruction is ready to issue. In Figure 7, we report the percentage of total dynamically executed load instructions which have been serialized.

Measurements of the dynamic percentage of serialized load instructions also consistently support the previous observations. As it can be seen from the experimental results shown in Figure 7, the three benchmarks, namely, `107.mgrid`, `125.turb3d`, and `130.li` which perform well as the issue width is increased have very low percentages of serialized load instructions. All the remaining benchmarks which perform poorly at high issue widths have significant percentage of the load instructions serialized.

3 Out-of-order Store Issue Algorithm

We have shown through a detailed analysis 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 4 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 4(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**. On the other hand, when the memory disambiguator imposes an ordering on those stores which may have the same effective address, false memory order violations will not be observed since a load instruction which is dependent on a later store instruction would not issue before all store instructions preceeding the store instruction it is dependent on issue.

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.

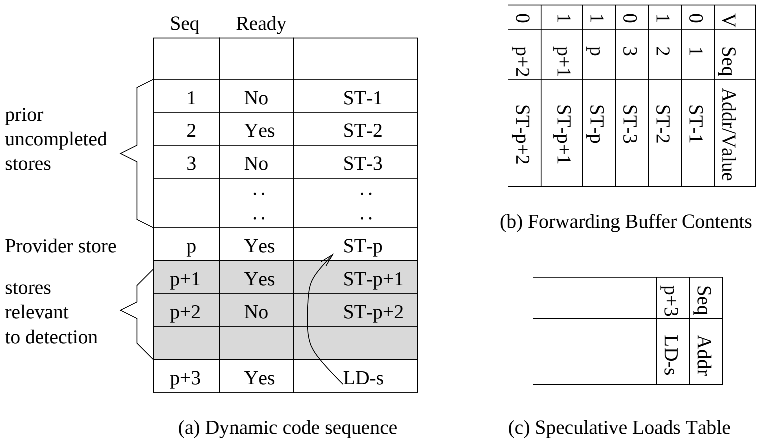


Figure 8: Speculative issuing of loads

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 8(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. In other words, 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 8(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 simpler and yet more 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. Since the processor experiences very few exceptions due to memory order violations when equipped with a good memory dependence predictor, the additional penalty of late restart is not high. 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, as it will be seen shortly, 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 instruction 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 instruction.

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

   If the addresses do not match, no action is taken.
4. Once the load instruction 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.

Please 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 8. 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. Please 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. Although we continue to use a memory dependence prediction mechanism that relies only on the load and store program counter values, we can effectively handle problems arising from multiple instances of the same load or store instruction.

Taking advantage of value redundancy. 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. A review of the workings of the store set algorithm should explain why.

During the initial start-up, there is no information in the SSIT and LFST tables to guide the scheduling. Because of the blind speculation of loads, loads acquire values either from the forwarding buffer or directly from the memory. When the actual store instruction that the load is truly dependent on retires with a value that is the same as the load instruction's value, the speculation is successful and no new entries are created in the SSIT and LFST tables. In other words, **the load speculates and executes successfully before the store it is actually dependent on**. The machine will continue to speculate the same load instruction until a violation occurs. Once a violation occurs, the load instruction will not be speculated further since it will wait for its producer store. In other words, the machine takes advantage of the value redundancy as long as it is beneficial to do so. By speculating in this manner we do not incur any performance overhead compared to an address only approach. Instead, those load instructions that can take advantage of the data redundancy are automatically selected by the algorithm. Although this process is not directed *intelligently* as in [1], we obtain similar benefits by not paying the penalties associated with a technique that speculates indiscriminately. The net effect of the technique is reduced load access latency.

In the next section, we demonstrate quantitatively that for most benchmarks, the technique indeed reduces the load access latency below what is possible with an ideal disambiguator which faithfully observes the true dependencies.

4 Performance Evaluation

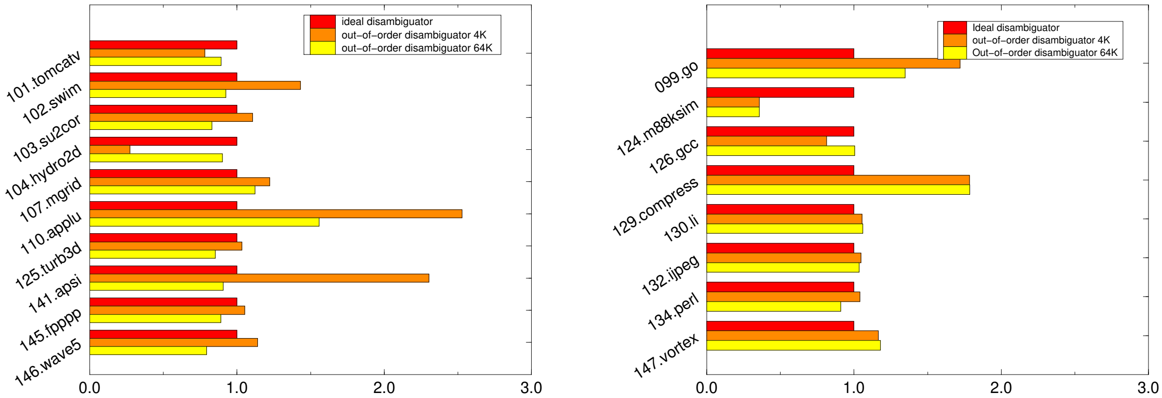
In order to assess the performance potential of store set memory disambiguator with our modified back-end we have designed a series of experiments. We have fully implemented the algorithm and executed the `SPEC95` benchmarks with their training or test inputs. The processor parameters have been kept as before(Figure 2). In these experiments, we compare the performance of the out-of-order memory disambiguator with both the ideal disambiguator as well as the original store set algorithm when it is appropriate.

Dynamic load latencies. Normalized average dynamic load latencies for out-of-order disambiguator is shown in Figure 9. It is interesting to note that with the exception of `110.applu` and `107.mgrid`, all normalized dynamic load latencies for floating point benchmarks are below 1.0. In other words, out-of-order disambiguator yields better dynamic load latencies than the ideal memory disambiguator. This is expected, especially with those programs with significant degrees of data redundancy. Such data redundancy occurs when the actual store instruction a load is truly dependent on is data redundant with respect to the stale value in the memory, or other issued but not yet committed store instructions. Since the ideal disambiguator makes a load wait for precisely the exact producer store instruction, in those cases where the store instruction is data redundant it makes the load instruction wait much longer. Although not to the same level of uniformity, a similar behavior is observed also among integer benchmarks. Our technique results in a significantly longer dynamic load latency only in case of `129.compress`. Nevertheless, the figure is still well below of the original store set algorithm (2.23 versus 1.79). `124.mk88ksim` shows an outstanding performance providing a dynamic load latency which is only a fraction of the ideal disambiguator (0.355). Normalized standard deviation values follow a similar trend (Figure 10) indicating that the success of the technique is uniform throughout the benchmark's execution.

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. When we analyze the `instructions per cycle` figures, the benefits of using our approach become increasingly clear as the issue width is increased. 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% (see Figures 11(a) and 11(b)).

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 (see Figures 11(c) and 11(d)). The performance difference further widens to 42% and 52% with floating point and integer benchmarks when the issue width is increased to 32 instructions (see Figures 11(e) and 11(f)). 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.

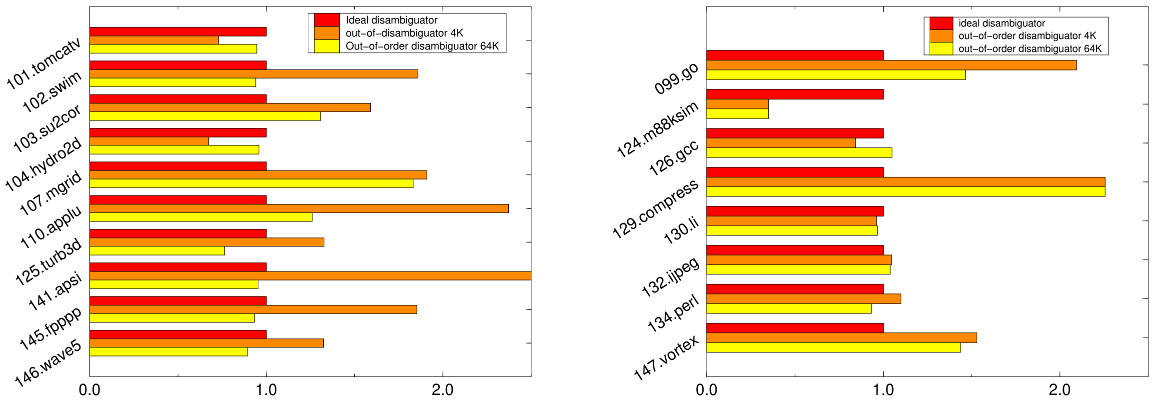
Scalability for different table sizes. In order to further compare the performance of our approach to that of the original algorithm, we executed both algorithms at predictor table sizes of 4K, 8K, 16K, 32K and 64K entries. We observed experimentally that the size of the `LFST` table is not critical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



(a) Floating Point Benchmarks

(b) Integer Benchmarks

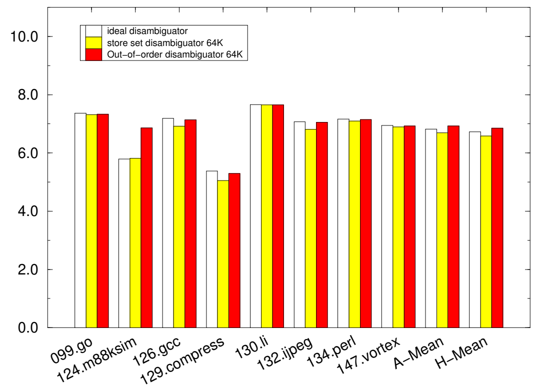
Figure 9: Normalized Average Dynamic Load Latencies



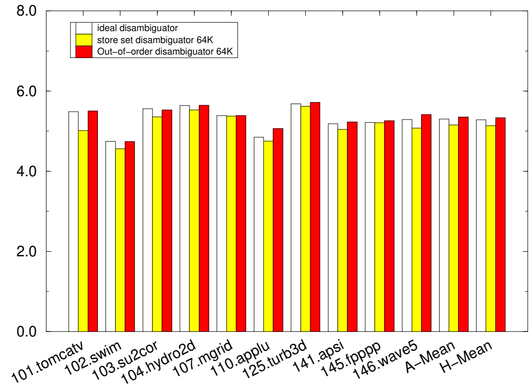
(a) Floating Point Benchmarks

(b) Integer Benchmarks

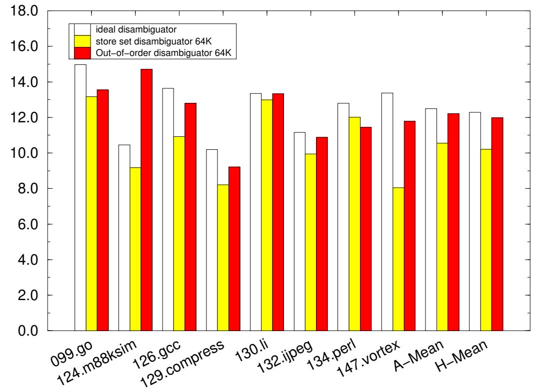
Figure 10: Normalized Standard Deviation Values for Dynamic Load Latencies



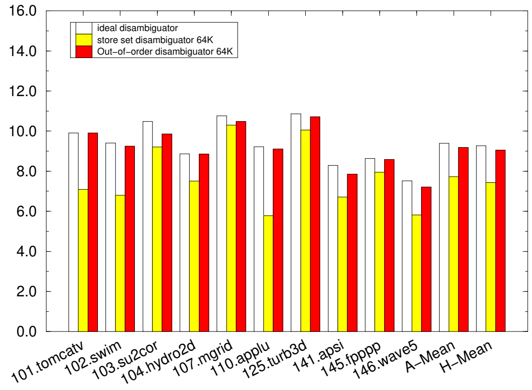
(a) 8 Issue - Integer Benchmarks



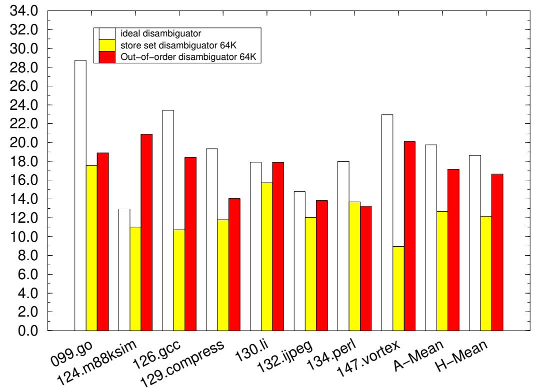
(b) 8 Issue - Floating point Benchmarks



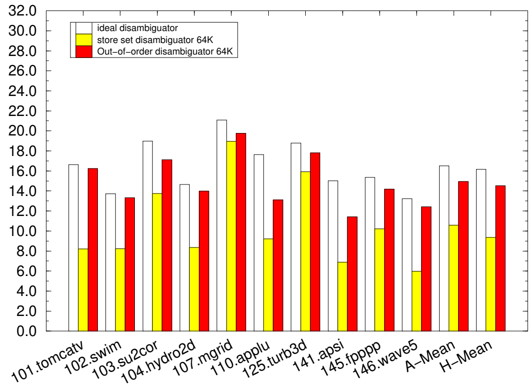
(c) 16 Issue - Integer Benchmarks



(d) 16 Issue - Floating point Benchmarks

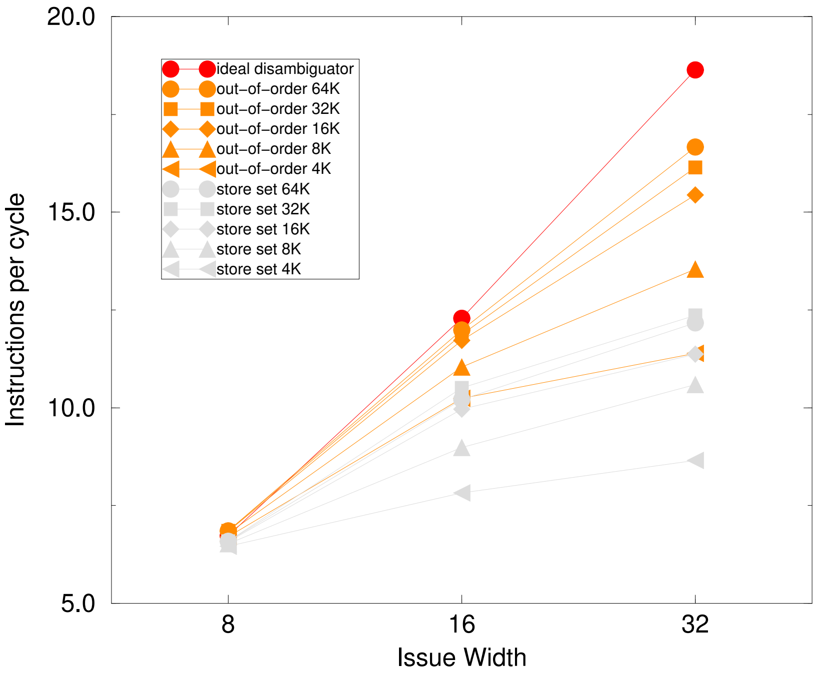


(e) 32 Issue - Integer Benchmarks

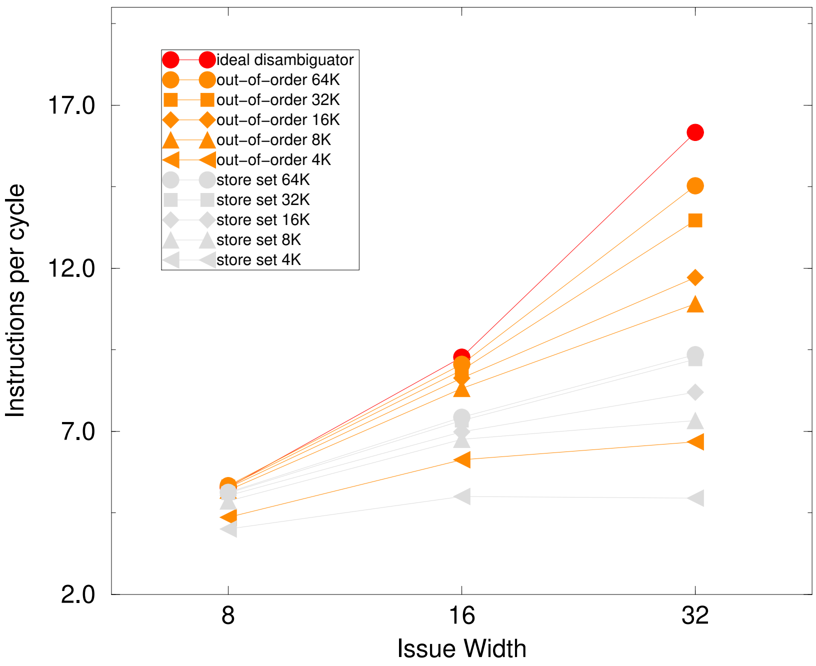


(f) 32 Issue - Floating point Benchmarks

Figure 11: IPC values Out-of-order Store Set, Store Set and Ideal cases



(a) Scalability of integer Benchmarks

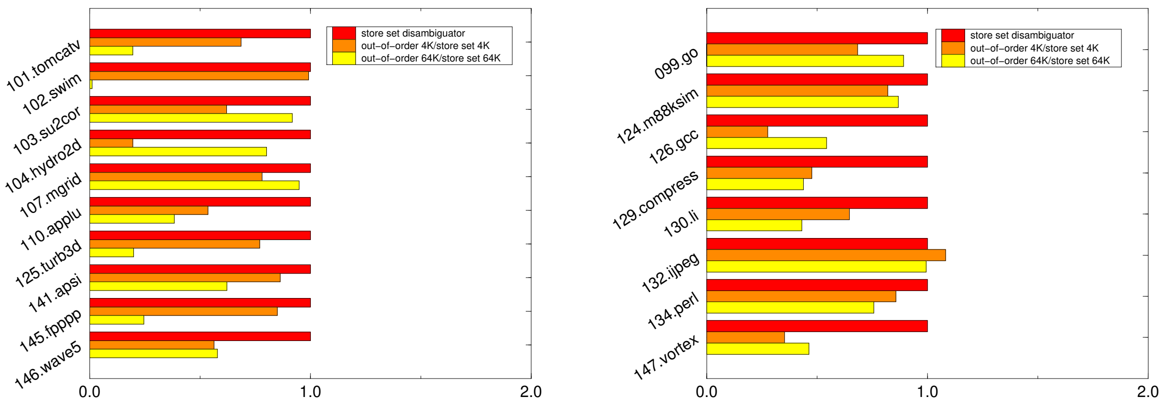


(b) Scalability of Floating point Benchmarks

Figure 12: Scalability of Out-of-order Algorithm

As it can be seen from the graphs in Figure 12(a) and Figure 12(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 and the comparison of the original store set and our algorithm is given in Figure 13.

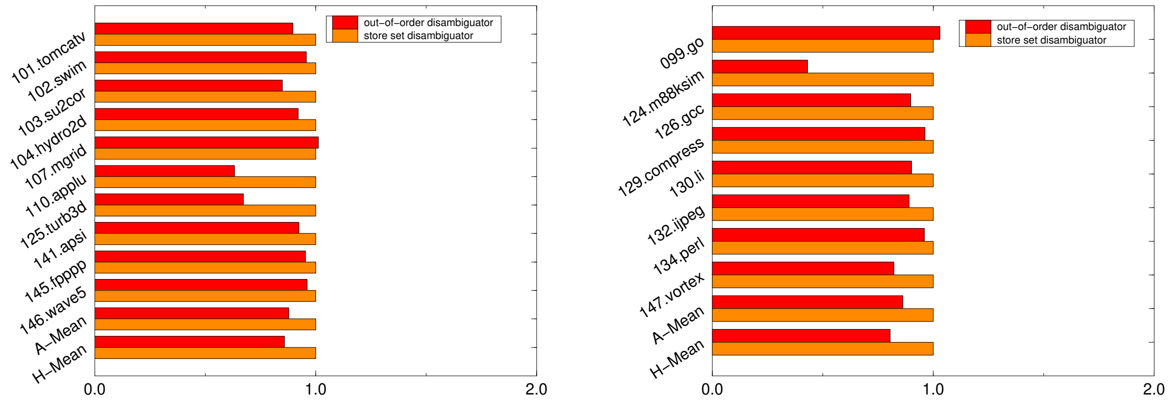


(a) Floating Point Benchmarks

(b) Integer Benchmarks

Figure 13: Normalized False Memory Dependencies

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. In order to verify that this is indeed the case, we measured the number of dynamic instances of load and store instructions for which there is a matching SSIT entry. We normalized this number by dividing it with the values produced by the original store set algorithm. The results of these experiments are shown in Figure 14.



(a) Floating Point Benchmarks

(b) Integer Benchmarks

Figure 14: Normalized Counts of Load/Store Instructions Synchronized Through SSIT Table

5 Conclusion

We have presented an effective mechanism for reducing false memory dependencies when using a memory dependence predictor. We have shown that we can allow full out-of-order issuing of store instructions in the instruction window using our memory order violation detection mechanism. We have also shown that we can take advantage of value redundancy even when we are not 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.

Finally, we would like to mention a study by Reinman and Calder that 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 out-of-order disambiguator presented in this paper out-performs both the original store set algorithm as well as an ideal memory dependence predictor, further studies are needed to verify their conclusion that the value prediction out-performs all other techniques.

Acknowledgments

The authors would like to thank George Z. Chrysos and Joel Emer for clarifying certain issues which arose when we implemented the original store set algorithm.

References

[1] Brad Calder, Glenn Reinman, and Dean M. Tullsen. Selective value prediction. In *Proceedings of the 26th International Conference on Computer Architecture*, pages 64–74, May 1999.

[2] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 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] Stephan Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. A Novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *The 31st Annual IEEE-ACM International Symposium on Microarchitecture*, pages 216–225, December 1998.

[5] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996.

[6] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, October 1996.

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

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

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

[10] Glenn Reinman and Brad Calder. Predictive techniques for aggressive load speculation. In *The 31st Annual IEEE-ACM International Symposium on Microarchitecture*, December 1998.

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