Memory Dependence Prediction using Store Sets

George Z. Chrysos and Joel S. Emer
Digital Equipment Corporation
Hudson, MA 01749
{chrysos,emer}@vssad.hlo.dec.com

Abstract

For maximum performance, an out-of-order processor must issue load instructions as early as possible, while avoiding memory-order violations with prior store instructions that write to the same memory location. One approach is to use memory dependence prediction to identify the stores upon which a load depends, and communicate that information to the instruction scheduler. We designate the set of stores upon which each load has depended as the load's "store set". The processor can discover and use a load's store set to accurately predict the earliest time the load can safely execute. We show that store sets accurately predict memory dependencies in the context of large instruction window, superscalar machines, and allow for near-optimal performance compared to an instruction scheduler with perfect knowledge of memory dependencies. In addition, we explore the implementation aspects of store sets, and describe a low cost implementation that achieves nearly optimal performance.

1. Introduction

Modern superscalar processors such as the Alpha 21264[1], MIPS R10000[2], HP-PA8000[3] and Intel Pentium Pro [4], allow instructions to execute out of program order to find more instruction level parallelism (ILP). These processors must monitor data dependencies to maintain correct program behavior. There are two types of data dependencies. Register dependencies occur when one instruction writes a register and a subsequent instruction reads the same register. Memory dependencies occur when a store instruction writes to a memory location and a subsequent load instruction reads that same location.

Register dependencies are determined in the instruction decode stage by examining instructions' register operand fields. Memory dependencies cannot be determined as early, because they require the computation of the memory address, which occurs after register operands are ready and the instruction is issued.

This lack of information about memory dependencies at instruction decode time is a problem for an out-of-order instruction scheduler. If the scheduler executes a load before a prior store that writes to the same memory location, the load will read the wrong value. In this event the load and all subsequent dependent instructions must be re-executed, resulting in a performance penalty. To avoid these *memory-order violations*, the scheduler could be conservative and prevent loads from executing until all prior stores have executed. This approach decreases performance because, in most cases, loads will be made *falsely dependent* on unrelated stores, unnecessarily delaying their execution. This dilemma has created the need for *memory dependence prediction*.

The goals of memory dependence prediction are 1) to predict the load instructions that if allowed to execute would cause a memory-order violation and 2) to delay the execution of these loads only as long as is necessary to avoid a such a violation [5]. When a memory dependence predictor makes a mistake, it fails to satisfy one of these two goals. To quantify the success of a memory dependence predictor, we count the number of memory-order violations and the number of false dependencies created by the predictions. This paper will use these two metrics, as well as overall program performance, to evaluate our memory dependence predictor.

Our memory dependence predictor is based upon the concept of store sets. A *store set* for a specific load is the set of all stores upon which the load has ever depended. A load's store set can be approximated in hardware by first allowing speculation of all loads around older stores. If a load executes before a store upon which it depends, the processor detects a memory-order violation when the store is executed and adds the store to that load's store set. Essentially the processor discovers and remembers a load's store set during program execution. The store set is then used to predict which stores a load must wait for before executing.

In this paper, we focus on the performance impact of memory dependence prediction based on store sets in an eight-wide superscalar out-of-order processor. We describe our CPU model and simulation environment in Section 2. In Section 3, we illustrate the benefit of memory dependence prediction by considering two alternatives and comparing their performance to that of a perfect memory dependence predictor. Section 4 discusses related work. We explain store-sets-based memory dependence prediction in Section 5, and propose and evaluate an implementation in Section 6. In Section 7, we analyze the performance of our memory dependence predictor on a benchmark where it performs less than optimally.

2. Simulation Environment

Accurate memory dependence prediction becomes more important for wider-issue processors with larger instruction windows. Therefore, we focus on the need for memory dependence prediction for next-generation out-of-order, speculative-execution processors. Our CPU model represents a processor with roughly double the caches and issue width of the Alpha 21264, and executes the Alpha

instruction set. We chose the size of the instruction queue, 128 entries, based upon performance sensitivity analysis by increasing the size of the instruction queue by powers of two until no further performance could be gained for our machine configuration. We model an aggressive fetch unit, which can fetch multiple basic blocks in a cycle, and a large McFarling-style choosing branch predictor [9]. The model parameters are listed in Table 2.1.

Table 2.1

CPU Model
• 128 entry instruction queue
• 128K 2-way set-associative Instruction cache
• 128K 2-way set-associative write-back Data cache
• 8 Instructions maximum issued per cycle
• 4 D-Cache Ports (any combination of loads and stores)
• 8M Direct Mapped, Write-Back Unified Second Level Cache

We assume that the processor detects memory-order violations by keeping a table of the effective addresses of all loads while they are in-flight speculatively. That table is checked for each store to see if a speculatively executed load actually depended on the store. If the processor detects a memory-order violation, it must have some recovery mechanism to salvage the correct architectural state. Our simulator recovers from a memory-order violation by *trapping* the load involved in the memory-order violation, *i.e.,* squashing that load and all subsequent instructions in program order, whether they depend on the load or not. It then re-fetches the instruction stream starting with the load in violation. Our CPU model can re-fetch the instruction stream the cycle after a memory-order violation is detected.

We used the SPEC95 benchmarks as a test suite. To avoid long simulation times, we ran our simulations until our simulator had retired 100 million instructions. We skipped over initialization code, and warmed up the caches before starting the actual simulation. The benchmarks were compiled with the standard DEC C and FORTRAN compilers with full optimizations under Digital Unix 4.0.

3. Motivation

To measure the importance of memory dependence prediction, we first considered two approaches that do not predict memory dependencies: *no speculation* and *naive speculation*. We considered these approaches because they are virtually free in hardware, giving us a lower bound in performance. Also, they allow us to measure the maximal impact of the two sources of performance loss in memory dependence prediction, *false dependencies* and *memory-order violations*, when compared with a perfect memory dependence predictor.

3.1 No Speculation

We configured our simulator to require that all load instructions wait until all prior store instructions have issued before being allowed to issue. This conservative approach eliminates all memory-order violations, but also creates many false memory dependencies. Loads waiting for a store instruction must wait for the store's address *and* data value register to be ready before issuing.[1] For some loads, the additional delay of the false dependence will be hidden by the out-of-order execution engine. Other loads will be on the critical path and the unnecessary delay will have a direct impact on performance. Overall, we found the performance of this scheme to be poor. Table 3.1 lists the rates at which false dependencies occurred, *i.e.*, the number of times that a load waited unnecessarily for a store, when running with the no speculation model. The large rate of occurrence of these false dependencies gives a crude indication that the performance impact of these false dependencies will be great. In fact, as we show in Figure 3.1, some benchmarks' execution time increased by a factor of two or more compared with a perfect memory dependence predictor.

3.2 Naive speculation

To examine the opposite extreme, we configured our CPU simulator to execute loads when their register dependencies were ready, independent of their memory dependencies. Again, no memory dependence predictor is used. The processor allows the loads to execute out-of-order, committing memory-order violations and causing traps. Table 3.1 shows the number of memory-order violations per thousand retired instructions. Between 0-7% of all dynamic loads resulted in memory-order violations for the SPEC95 benchmarks. This metric is only intended to give an indication of performance impact. Like branch misprediction or cache-miss rate, the actual performance impact can often only be determined by knowing the penalty.

To estimate this penalty we calculate the *memory trap penalty* as the number of cycles between the first time a load is fetched to the next time the load is fetched after a memory-order violation has been detected. That time can vary from load to load, and program to program depending on the time it takes for the conflicting store to execute. The memory trap penalty indicates the average number of cycles in which no useful instructions were fetched. This gives an indication of the performance impact due to memory-order violations. Table 3.1 shows the average memory trap penalty for each of the SPEC95 benchmarks when running with the naive speculation policy.

Tyson and Austin [8] suggest that the memory trap penalty can be effectively reduced by re-executing only the load instruction and its dependent instructions, but not re-executing any other instructions that were fetched after the load but are not dependent on it. Designing the processor to have the ability to replay only dependent instructions could significantly reduce the memory trap penalty, but could also negatively impact performance. In order to replay only dependent instructions, typically those instructions must remain in the instruction queue until they retire. This is in contrast to the DEC Alpha 21264, where instructions are aggressively de-allocated from the instruction queue when

1 A processor could split a store into two operations, the first being the effective address computation and the second being the delivery of the data value to the memory. This could reduce the time needed to resolve store-load dependencies. These "split store" solutions are not considered in this paper.

they issue. This enables a smaller instruction queue to find more instruction level parallelism because issued instructions are removed from the queue allowing newer instructions to participate in bidding for execution resources. A smaller instruction queue is advantageous when designing a short cycle time processor. Our goal is to virtually eliminate memory-order violations, making the memory trap penalty unimportant, while retaining the advantage of the early instruction queue de-allocation.

Table 3.1

	Naive Speculation		No Speculation
Spec95 Program	**Memory Order Viols Per 1K Instrs**	**Memory Trap Penalty (Cycles)**	**False Dep. Per 1K Instrs**
go	6	13	157
m88ksim	20	12	168
gcc	5	15	187
compress	11	15	129
xlisp	11	14	179
ijpeg	23	15	150
perl prim	20	15	215
perl scrab	10	15	185
vortex	7	19	215
tomcatv	4	22	264
swim	2	36	224
mgrid	0	18	262
applu	18	22	212
apsi	7	35	247
fpppp	10	17	275
wave5	24	21	188
turb3d	6	16	213

3.3 Perfect Memory Dependence Prediction

To quantify the potential benefit of memory dependence prediction, we ran the same benchmarks through a perfect memory dependence predictor, which does not cause memory-order violations. Also, the perfect predictor does not impose false dependencies, omnisciently causing loads to wait for exactly the right store. We compared the performance of the perfect memory dependence predictor to that of the no speculation and naive speculation approaches. Figure 3.1 shows the performance of each benchmark running on our simulator, in instructions per cycle (IPC).

The performance of naive speculation tends to be better than that of no speculation in most cases. Most of the programs, however, suffer significant performance degradation compared with the perfect predictor. Clearly, we would like to allow loads to speculate around stores, since the no speculation policy severely impacts performance. On the other hand, memory-order violations from naive speculation result in significant performance degradation due to squashing and re-executing instructions. Thus, it appears a memory dependence predictor would be beneficial.

4. Related Work

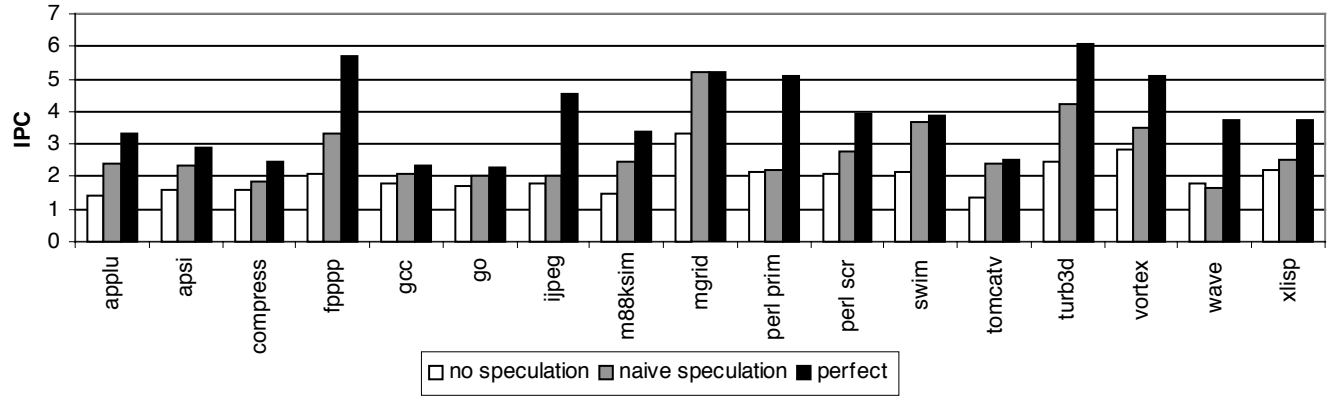
Memory dependence prediction has been the subject of recent work in industrial and academic computer architecture research. In this section, we examine some existing proposals for allowing speculation while avoiding memory-order violations.

Steely, *et al*., of Digital Equipment Corporation filed a patent on a general framework for allowing loads to speculate around prior stores, and synchronizing the execution of loads and stores which tend to cause memory-order violations[11]. Their proposal describes assignment of tags to load and store instructions that cause memory-order violations, and orders execution among memory instructions that have been assigned the same tag.

Hesson, *et al*., of IBM filed a patent for the *store barrier cache*[6], which keeps track of stores that tend to cause memory-order violations. Stores causing memory-order violations set a bit in the store barrier cache. When a store with that bit set is fetched, the processor inhibits all subsequent loads from executing until that store executes. The intent is to reduce memory-order violations by enforcing strict ordering around stores that tend to conflict with nearby loads, effectively creating store barriers.

Independently, Moshovos *et al*.[5] published a comprehensive description of memory dependence prediction. This is the first published work identifying that memory dependencies are problematic for out-of-order machines. They also discovered that it is important to delay depend-

Figure 3.1: Performance of No Speculation, Naive Speculation and Perfect Prediction



ent loads, but only as long as necessary to avoid memory-order violations and not impose false dependencies. They proposed a mechanism to avoid memory-order violations; a pair of fully associative structures that hold store-load pairs that have caused memory-order violations in the past. These structures direct the order of execution for subsequent instances of the instructions.

Other work uses memory dependence prediction to supply the consumers of dependent loads with data sooner. In [7] Moshovos and Sohi use memory dependence prediction to guide forwarding of data values from stores to dependent loads. Work by Tyson and Austin[8] combines memory dependence prediction and load value prediction [12] to satisfy the consumers of a load early. The ideas explored in our paper might also benefit from using these ideas to further increase performance.

Our paper explores the premise that the history of memory dependencies can accurately predict future memory dependencies. In order to do this we stress the concept of a store set, which contains all prior memory dependencies for a load. The concept of store sets is similar to an idea mentioned in [7]. We show, in an idealized model, that store sets are actually a good construct for use in predicting memory dependencies in the context of a large instruction window machine. We then propose novel algorithms to create a fast, reasonably low-cost "common tag" style[11] implementation of a store-sets-based memory dependence predictor that achieves nearly optimal performance using direct mapped structures.

5. Store Sets

The concept of store sets is based upon two underlying assumptions. The first is that the historic behavior of memory-order violations is a good predictor of future memory dependencies. The second is that it is important to predict dependencies of loads where one load is dependent on multiple stores or multiple loads depend on the same store. In this section, we describe the concept of store sets and their use in memory dependence prediction. Once we have established that the use of store sets is effective for memory dependence prediction, we go on to describe a low-cost implementation in Section 6.

5.1 Concept

We define each load instruction in a running program as having an associated set of store instructions, called its *store set*. A load's store set consists of all stores (identified by their PC) upon which it has ever depended. A processor will approximate the load's store set by keeping track of the stores that have caused the load to suffer memory-order violations in the past.

When a program begins executing, all of the loads have empty store sets, and the processor allows naive speculation of loads around stores. When a load and store execute in the wrong order, causing a violation, the store PC is added to the load's store set, *e.g.*, set **A**. If another store conflicts with that same load, that store PC is also added to set **A**. The next time the processor sees that load, it requires that the load execute after any recently fetched stores in store set **A**. When the load is fetched, the processor will determine which stores in the load's store set were recently fetched but not yet issued, and create a dependence upon those stores. Loads that never cause memory-order violations will have no imposed memory dependencies, and will execute as soon as possible. Loads that cause memory-order violations will be dependent on only those prior stores upon which they have depended in the past. If one of the stores in store set **A** also causes a memory-order violation with another load, it becomes part of that load's store set also. Loads and stores are identified by their PCs. Here is an example:

Example 5.1

```
PC
0     store C
4     store A
8     store B
12    store C

28    load B  store set { PC 8 }
32    load D  store set { (null) }
36    load C  store set { PC 0, PC 12 }
40    load B  store set { PC 8 }
```

Each load's store set has been annotated. Notice that a load can have multiple store dependencies. For example, the load at PC 36 depends on both the stores at PC 0 and PC 12. Also multiple loads can depend on the same store. For example, the loads at PCs 28 and 40 both depend on the store at PC 8.

This situation where multiple loads depend on the same store is fairly common, occurring when there is one writer and multiple readers of a value. One load depending on multiple stores is somewhat more obscure. Three situations where this can happen are: 1) A load can depend on stores from different paths, *e.g.*, if (expr) then x = a; else x = b; ..... c = x; 2) a load can depend on multiple stores to fields of a structure packed in a data word that are all read together, *e.g.*, writes of the red, green and blue components of a color structure; and 3) assuming that memory write-after-write hazards are treated as dependencies, a load can depend on a series of stores to the same location; *e.g.*, multiple spills to the same location.

To evaluate the importance of allowing multiple loads to depend on the same store, and one load to have multiple store dependencies, we created three configurations of our simulator. First, we created a configuration in which each load in the program has its own store set, and that store set can contain as many stores as necessary, but a store can only reside in one store set. When a store causes a memory-order violation, it is eliminated from any store set it is in, and placed in the store set of the load with which it conflicted last. Essentially this prohibits multiple loads from being dependent on the same store.

In the second configuration, we limited the size of a store set to one, allowing each load in the program to specify at most one store dependence. This prohibits one load from being dependent on multiple stores. Note that unlike the last configuration, we do allow one store to exist in as many store sets as necessary in this configuration. When a memory-order violation occurs, the store replaces any prior store in the load's store set.

Figure 5.1: Effect on Run Time When Not Allowing Multiple Dependence Flexibility

Lastly we created an "infinite" configuration, in which neither the store set size nor the number of store sets in which a store can appear is confined. Figure 5.1 shows the normalized increase in run time for each of the first two configurations when compared with the third configuration. Clearly, either of these two constraints on store-sets would significantly impact overall IPC in some applications.

As an aside, we also expanded the second configuration to allow one load to depend on 2, 4 or 8 stores. The 8-way associative structure resulted in no performance degradation over not constraining the number of stores per load. The 4-way structure was also fairly good. `Perl (primes)` was the only benchmark that suffered significant performance degradation with the 4-way structure. In Section 6, we describe a method for using direct mapped structures and retaining multiple dependence flexibility for both cases described above.

5.2 Performance

To evaluate the performance of store-set memory dependence prediction, we used the "infinite" configuration described above. Each dynamic load encountered in the simulation was classified as one of the following: not predicted (for loads that have empty store sets), predicted correctly, falsely dependent, or memory-order violation. The performance of the predictor can be evaluated by the quantity of memory-order violations and false dependencies. The data for SPEC95 is shown in Figure 5.2. For most of the benchmarks, the infinite predictor does well. As expected, it effectively eliminates memory-order violations for all of the benchmarks. A few of the benchmarks have a significant number of false dependencies. `Applu` and `xlisp` suffer from false dependencies the most, and less so for `vortex`, `gcc`, `go` and `swim`.

The infinite store set predictor could suffer false dependencies if a memory dependence does not exist for every execution of a particular store-load pair. Once a store PC is placed in a load's store set, it stays there for the rest of the run, requiring that the load wait for the store in every future instance of the two PCs executing. If the store-load pair access the same memory location infrequently, many false dependencies are imposed to save a few memory-order violations.

This intermittent memory conflict between a store and a load encourages the notion of a "working" store set that contains stores that the load depended on recently. To estimate this "working" store set, we use a feedback that

Figure 5.2: Infinite Store Sets Predictor - Dynamic Load Breakdown

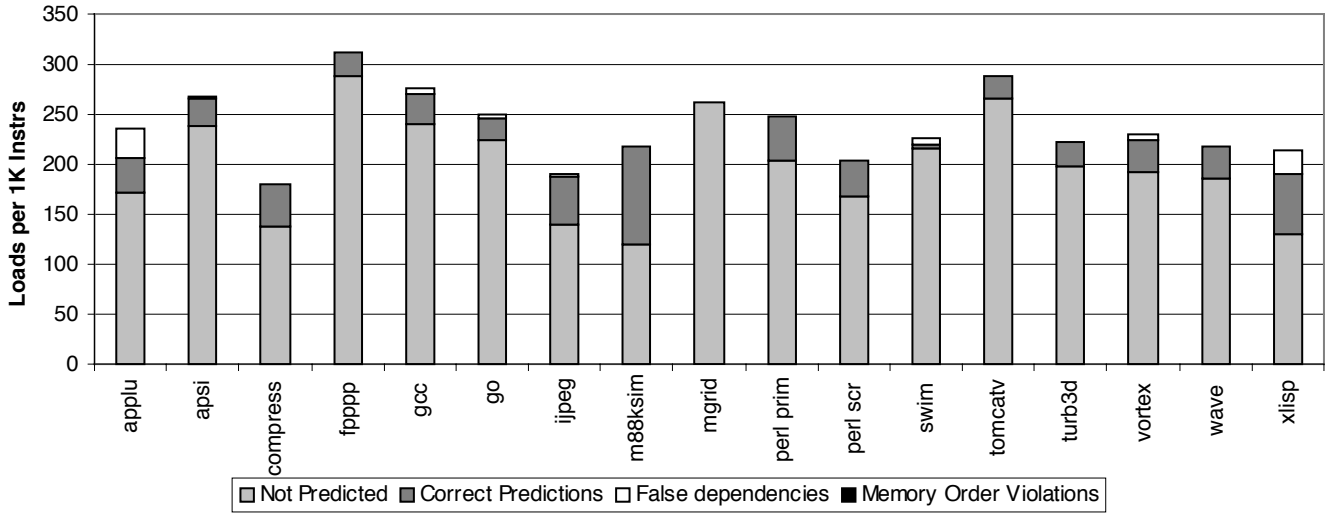
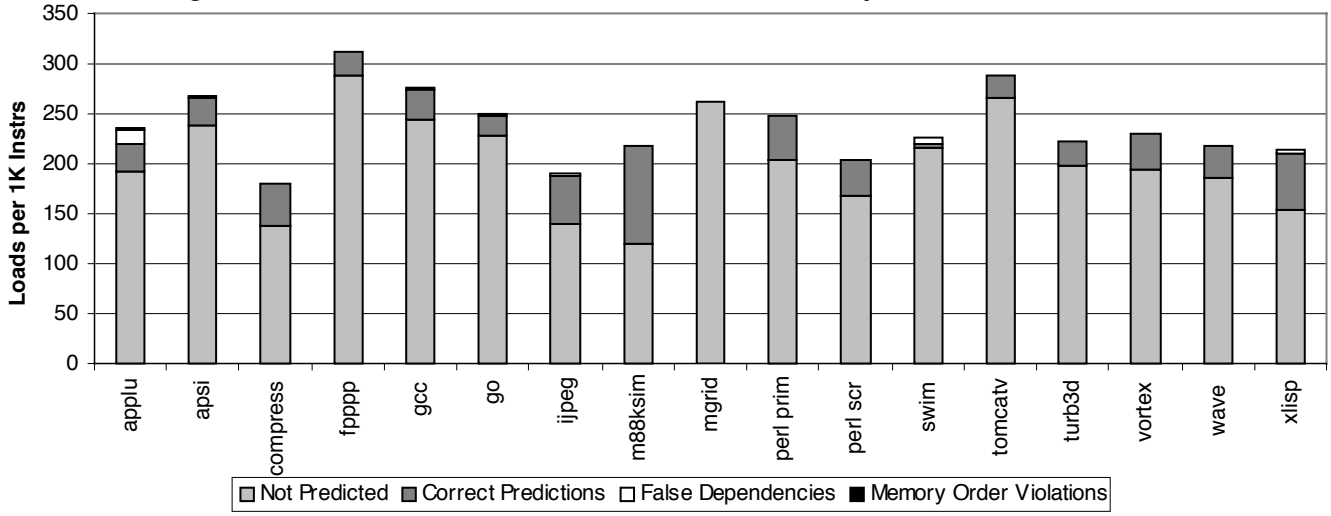


Figure 5.3: Infinite Store Sets Predictor with 2 Bit Counters - Dynamic Load Breakdown



reinforces or decays prediction decisions. In particular, using a method commonly found in branch predictors, we chose to couple a simple two-bit saturating counter scheme [10] with the infinite store-set memory dependence predictor to see if we could reduce false dependencies. A two-bit counter was appended to each store in a load's store set. When a memory violation occurred, the two-bit counter would be set to its maximum value. After a load is forced to order with a store in its store set, the load's address is compared to the store's address to verify that a memory dependence did exist. If a real dependence did not exist, the counter is decremented by one. If a real dependence did exist, the counter is incremented by one. The load is forced to wait for a store in its store set only if the high bit of the counter is set. This is an attempt to reduce false dependencies, but could allow memory-order violations to recur if the pattern of memory dependence is more sophisticated than the two-bit counters can detect.

We reran the SPEC95 benchmarks with the two-bit counter configuration, again classifying the loads into four categories. The results in Figure 5.3 show that two-bit counters work well for reducing the number of false dependencies without increasing the memory-order trap rate to significant levels. The only remaining benchmark with a significant number of false dependencies is `applu`. A detailed analysis of `applu` appears in Section 7.

To quantify the performance of the infinite memory dependence predictor using store sets, we compare the performance in instructions per cycle (IPC) of the SPEC95 programs for different configurations of our simulator. Figure 5.4 compares the performance of the infinite memory dependence predictor with and without two-bit counters to the perfect memory dependence predictor.

Figure 5.4 shows that the infinite memory dependence predictor achieves close to optimal performance for all of the benchmarks. The biggest performance degradation was in `applu`, which is 10-14% worse than optimal. This was expected since `applu` also exhibited the highest number of false dependencies as shown in Figures 5.2 and 5.3. The two-bit counters slightly improved performance for `xlisp` and `applu`, but perhaps not enough to warrant their use.

In some rare instances, the IPC of the perfect predictor configuration was slightly less than that of the infinite predictor. This is because in our model, the "perfect" configuration also requires the elimination of executing wrong-path instructions in the event of a branch misprediction. Instead, the fetcher stalls until the mispredicted

Figure 5.4: Performance of Store Set Memory Dependence Prediction

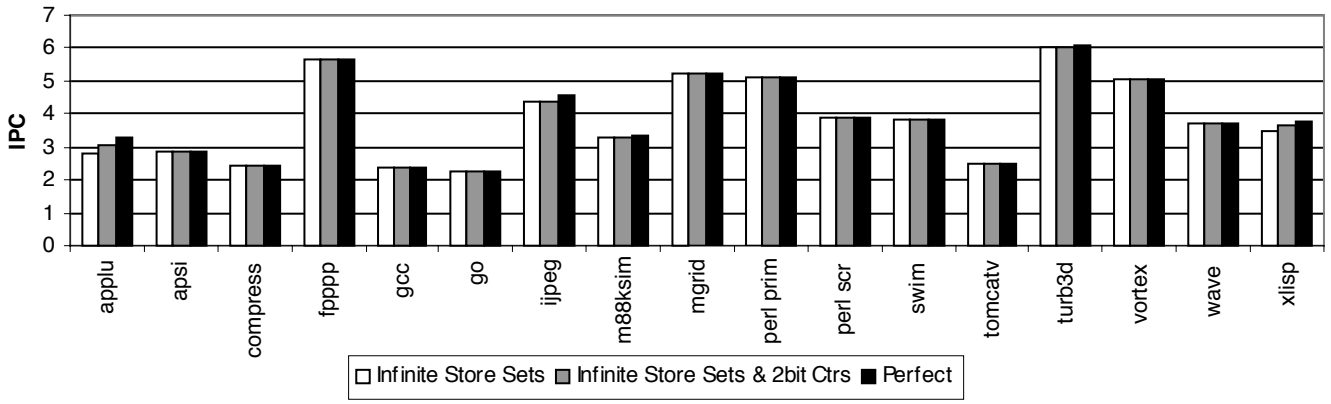
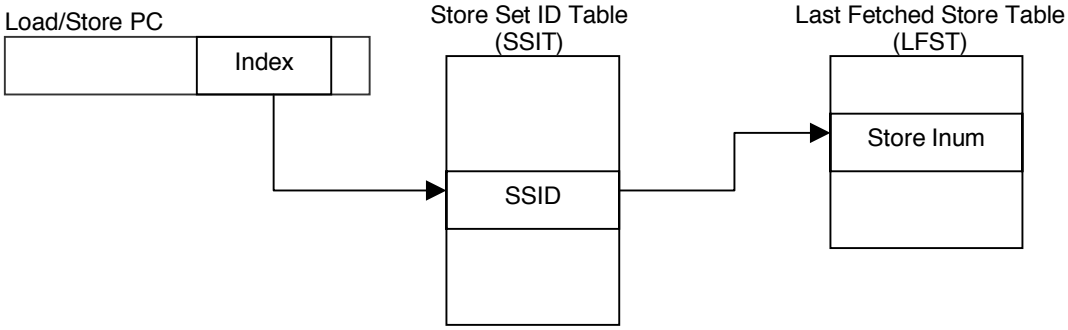


Figure 6.1: Implementation of Store Sets Memory Dependence Prediction



Loads and stores index into the SSIT to get their store set identifiers, which are used to access and update the LFST. The store inums that are found in the LFST indicate the memory dependence prediction.

branch resolves. We found this to be a minor effect (+/- 2%) due to the opposing effects of prefetching and cache thrashing that the wrong-path execution causes. The results indicate that if we can approximate the performance of the infinite store-set memory dependence predictor, we would have a solution with nearly optimal performance.

6. Store Set Implementation

We find that by imposing a few restrictions on store sets, we can retain the performance of the infinite memory dependence predictor, while creating a low-cost solution. The hardware used to implement store sets will consist of structures with finite dimensions. Limiting the size of the structures will force unrelated loads to share store sets, potentially introducing false dependencies. We also limit store PCs to exist in at most one store set at a time. This is similar to configuration two in Section 5.1, except that we propose algorithms to merge store sets, allowing two loads that depend on the same store to share a store set.

Since store sets allow a load to be dependent on multiple stores, it would theoretically be necessary to have a mechanism that delayed the load until all the stores in the store set had executed. Constructing such a mechanism could be expensive. We would like to make the load dependent on just one of those stores, but we do not know which of the load's store dependencies will be the last to execute. To solve this problem, we have conservatively constrained stores within a store set to execute in order. This is accomplished by making each of the stores dependent on the last fetched store within its store set. Each store specifies one dependence and each load specifies one dependence to form an in-order chain resulting in correct program behavior.

Requiring that the stores in a store set execute in order has the benefit that it eliminates complicated write-after-write hazard detection in the processor. If two sequential stores that are in the same store set both write to location X and are followed by a load to location X, the load really only depends on the second store in the sequence. Since we enforce ordering within store sets, special hardware is not needed to make this distinction.

6.1 Components

Our implementation of memory dependence prediction with store sets consists of two tables. The first is a PC indexed table called the *Store Set Identifier Table* (SSIT) that maintains the store sets using a common tag for each load and the stores in its store set. The second is called the *Last Fetched Store Table* (LFST) and maintains dynamic information about the most recently fetched store for each store set. The information in this table is the *inum* of the store, which is a hardware pointer that uniquely identifies the instance of each instruction in flight.

Recently fetched loads access the SSIT based on their PC and get their *store set identifier* (SSID). If the load has a valid SSID, then it has a valid store set. It will access the second table, the LFST, and get the inum of the most recently fetched store instruction that was a member of its store set.

Recently fetched stores also access the SSIT. If the store finds a valid SSID, then it belongs to a valid store set. The store must then do two things. First, it must access the LFST and get the most recently fetched store instruction in its store set. The new store will become dependent upon the store it found in the LFST. Second, it will update the table, inserting its own inum, since it is now the last fetched store in that particular store set.

After a store instruction issues, it accesses the LFST and invalidates the entry if it still refers to itself. This ensures that loads and stores will only be made dependent on stores that have not yet issued. If the stores are from different code paths and not executed every time the load is executed, only the store that is fetched will access the SSIT and modify the LFST. The load will always be dependent on the appropriate store and the stores will never be forced to depend on each other, since they are executed at different times. Figure 6.1 shows a diagram of the tables.

When the CPU recovers from a misspeculation (branch mispredict, jump mispredict, or memory-order violation) the SSIT does not need to be modified. Ideally, the LFST must roll back each entry to the last live store that wrote that entry. Although we modeled that behavior, we believe that simply marking aborted stores as done in the table would be sufficient.

The mechanism for memory dependence prediction using these tables is best explained by example. At the start of a program, assume that all the entries in the SSIT are invalid. Initially, store and load instructions will access the table and get no valid memory dependence information. If a load commits a memory-order violation, a store set is created in the SSIT. The load and store instructions involved in the conflict will be assigned a store set identifier, say SSID X. SSIDs can be assigned in a number of ways. We chose an exclusive-or hash on the load's PC. SSID X will be written into two locations in the SSIT; the first location will be indexed by the load PC, and the second by the store PC. The next time that store PC is fetched, it will read the SSIT entry indexed by its PC. Since the SSID is valid, it uses that store set's SSID to access the LFST where it finds no valid recent fetched instructions from store set X. So, it will not be made dependent on another store. The store proceeds to write its own inum into the LFST. When the load instruction is subsequently fetched, it accesses the SSIT and then the LFST with SSID X. The LFST conveys to the instruction scheduler that the load is dependent upon the store instruction it finds there. This time, the instruction scheduler will impose a dependence between the load and store, in a similar manner to the way it imposes register dependence constraints.

If the load later conflicts with a different store, the SSIT is notified of the new memory-order violation. SSID X is copied into the SSIT entry indexed by the new store's PC. Now there are three entries in the SSIT that point to SSID X. The next time the two stores and the load are fetched, the second store will depend on the first store, and the load will depend on the second store.

Note that stores are constrained to be in only one store set at a time to keep the SSIT and the LFST simple. A store could be part of more than one store set if we allowed the SSIT to hold more than one SSID in each entry. The multiple SSIDs would then have to access the LFST, increasing the number of read ports needed. Also, each load would then be dependent on up to two stores, adding more size and complexity to the instruction scheduling hardware that needs to impose dependencies.

The tables depicted in Figure 6.1 allow a store to be in only one store set at a time. This apparently violates the premise that a memory dependence predictor should allow multiple loads to be dependent on the same store. In the next subsection, we describe an algorithm that can use the simple hardware in Figure 6.1 and retain that functionality.

6.2 Store Set Assignment

When a memory-order violation occurs, entries are created in the SSIT. When neither the load's nor the store's SSID is valid, one is created and assigned to both. If the load's SSID is valid and the store's is not, the store inherits the load's SSID, and becomes part of the load's store set. We have not yet discussed what happens if the store already had a valid SSID. Overwriting the store's old SSID will effectively remove the store from its old store set and put it into a new one. This has the undesirable effect of limiting a store to be in only one store set at a time. To govern the decisions made when determining whether to overwrite one valid SSID with another, we have established some *store-set assignment rules*.

Limiting the number of store sets to which a store can belong could create new possibilities for memory-order violations. Multiple loads may cause memory-order violations due to the same store, and since a store can only belong to one store set, a conflict can occur where the loads compete to have the store in their own store set. This can result in poor behavior, similar to a cache thrash where two loads always cause each other to miss.

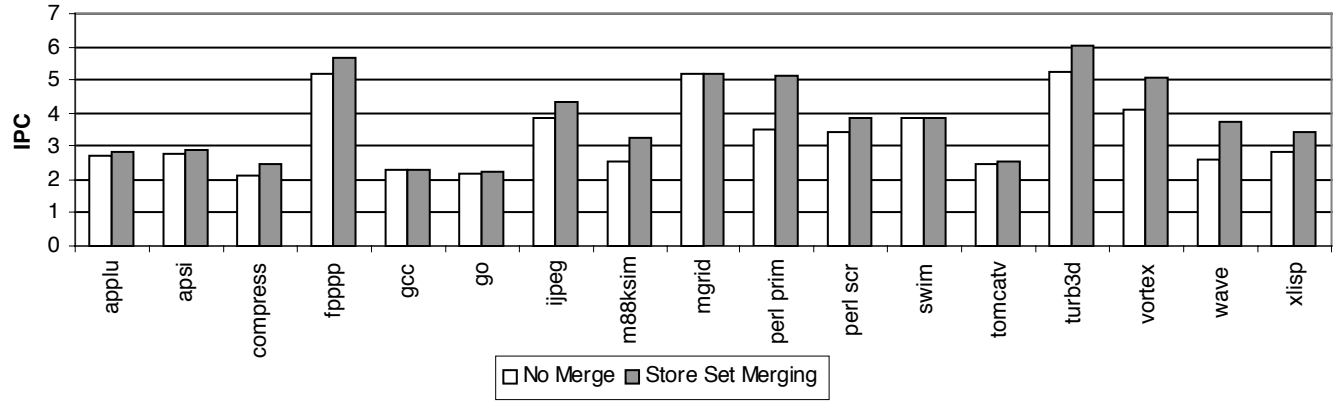
Example 6.1
Ld PC 1 -> Store Set 1 { St PC X, St PC Y, St PC Z}
Ld PC 2 -> Store Set 2 { St PC J, St PC K}

In Example 6.1, two loads have conflicted with stores in the past. Ld PC 1 has conflicted with the three stores specified in store set 1, and Ld PC 2 has conflicted with the two stores specified in store set 2. Now suppose that Ld PC 1 causes a memory-order violation when executing out of order with St PC J. We'd like to add St PC J to store set 1, but since we only allow a store to exist in one store set at a time, we must also remove it from store set 2. The next time Ld PC 2 executes, it traps because it conflicts with St PC J, and the memory dependence predictor did not enforce a dependence between them because St PC J is no longer in store set 2. It is apparent that if we are not careful, we can have an oscillation where Ld PC 1 and Ld PC 2 take turns trapping and removing St PC J from the other load's store set. To avoid this destructive interference, the store set assignment rules promote *store-set merging*. The store set assignment rules pertain to when a store-load pair has caused a memory-order violation. The rules are:

1. *If neither the load nor the store has been assigned a store set, one is allocated and assigned to both instructions.*
2. *If the load has been assigned a store set, but the store has not, the store is assigned the load's store set.*
3. *If the store has been assigned a store set, but the load has not, the load is assigned the store's store set.*
4. *If both the load and the store have already been assigned store sets, one of the two store sets is declared the "winner". The instruction belonging to the loser's store set is assigned the winner's store set.*

Rule four alludes to an arbitration algorithm that chooses a "winner" among two store sets. The arbitration algorithm should be one that, when asked to choose a winner among two specific sets, will always choose the same set. The simple algorithm we employ is choosing the store set with the smaller store set number. This algorithm will converge, and the store sets will be merged after some constant number of traps. Although there are more complicated algorithms that we could use, we found that this one works well. Figure 6.2 shows the performance benefit

Figure 6.2: Performance Improvement Due to Store Set Merging



of dynamic store set merging by comparing a run using the store set assignment rules, to a run where a store is automatically assigned the SSID pertaining to the most recent load with which it had a memory-order violation. Without store set merging, we lose an average of 12% performance. The data shows the importance of being able to predict memory dependencies for multiple loads that depend on the same store.

To understand how the store set assignment rules promote dynamic store set merging we look back at Example 6.1. Suppose Ld PC 1 causes a memory-order violation by executing before St PC J. We have the conditions for rule four to fire: the store involved in the violation is already in a store set and the load's store set is non-empty. Now we must declare a "winning" store set. Let's assume that we declare store set 1 the winner; now St PC J is removed from store set 2 and placed in store set 1. In the near future, Ld PC 2 causes a memory-order violation by executing before St PC J, since it has been removed from its store set. Again, we have the conditions for rule four. If store set 2 were declared the winner, Ld PC 1 and Ld PC 2 would alternatively cause memory-order violations because of St PC J. Therefore, rule four should choose store set 1 as the winner again. This would cause Ld PC 2 to adopt store set 1. If we keep going, Ld PC 2 will then cause a memory-order violation with St PC K, since St PC K is no longer in Ld PC 2's designated store set. St PC K is in its own store set (although no load points to it), and Ld PC 2's store set is not empty. Again, rule four should specify the winner, and once again to avoid oscillating memory-order violations, it should choose store set 1, bringing Ld PC 2's final conflicting store into its new store set. Now, both Ld PC 1 and Ld PC 2 point to store set 1, and all the stores are in store set 1. All further memory-order violations are avoided by this dynamic merging of store sets 1 and 2.

Merging store sets could potentially create false dependencies of one load on another load's stores. We found no performance impact resulting from such false dependencies.

6.3 Prediction Feedback

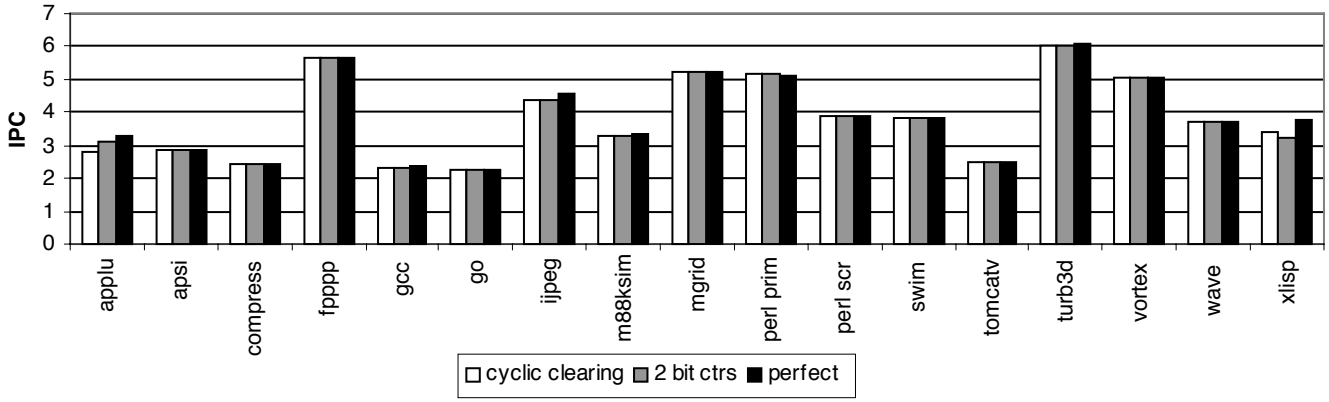
As a program runs, the SSIT registers valid SSIDs, creating valid store sets. We have not yet described any mechanism that invalidates entries in the SSIT. Because there are no tags in the SSIT, every load and store instruction that accesses the table uses the information that it finds. Two loads, one that needs a store set, and one that does not, might both index into the same SSIT entry. This can have a negative impact on performance, potentially causing false dependencies for loads that are not in danger of committing memory-order violations. Also, without a method of invalidating the table, all the entries in the SSIT will eventually become valid, randomly imposing ordering restrictions on whatever program executes in the processor.

As in Section 5.2, this argues for the use of a "working" store set. We use two methods to achieve this effect. First, we employ a *cyclic clear* algorithm on the valid bits in the SSIT. This simply means that every so often (in experiments we clear every one million cycles), the valid bits of the SSIT are cleared, and the table must retrain. Second, we again use the two-bit counter scheme that was added to the infinite store set memory dependence predictor. If false dependencies are incorrectly imposed on certain loads, the counters will decrement and eventually invalidate the entry, reducing the number of false dependencies.

We included two-bit counters in our implementation by appending each SSIT table with a two bit counter. When a store instruction first creates its entry in the SSIT, the two-bit counter is set to its maximum value. As long as the high bit of the counter is set, the instruction scheduler enforces memory dependencies predicted by the SSIT and LFST. The addresses of a predicted store-load dependence are compared to determine whether there is a false dependency. If the addresses do not match, there is a false dependency, and the counter associated with the store is decremented; otherwise it is incremented.

The counters are used only for store instructions, just as in the infinite predictor. Figure 6.3 shows the performance of what we found to be *sufficiently large* SSIT and LFST tables (16K and 512 entries, respectively) with two-bit counters, compared with the performance of a cyclic clear scheme. The two-bit counters help the performance of `applu`, reducing the number of false dependencies. `Xlisp`, however, shows a performance degradation, as opposed to the improvement seen when using two-bit counters in the infinite store-set predictor. This is caused by destructive aliasing in the SSIT. A load that does not

Figure 6.3: Implementation of Store Sets vs. Perfect



need a store set decrements the counter of an SSIT entry that another load uses to obtain memory dependence information. This results in allowing a significant number of memory-order violations, which degrades performance.

The comparison in Figure 6.3 argues against the use of two-bit counters in the SSIT. The extra space and complexity of maintaining the counters, and the address comparitors needed to determine whether a false dependence was imposed, are not worth the minor improvement in `applu`. Also, `xlisp` degrades about as much as `applu` improves, so there is no clear performance advantage to using two-bit counters. Clearing the valid bits of the SSIT on some large regular interval can easily be implemented with two counters, one that specifies a clearing interval, and the other that sweeps through the SSIT, invalidating entries. We look more carefully at the specific case of `applu` in the analysis section.

6.4 Table Sizes

An important implementation consideration is the size of the SSIT and LFST tables. We used the cyclic clearing implementation described in the last subsection, and varied the size of the SSIT and LFST to determine the performance/cost curve. Figure 6.4 shows our results for the SSIT sizing. The performance is relative to a configuration with sufficiently large SSIT and LFST tables (again, 16K and 512 entries respectively). For the experiments varying the SSIT table size, the LFST table was left sufficiently large. The data shows that a 4K entry SSIT does not compromise performance, and 1K entries is still acceptable.

To see the effect of reducing the table size more clearly, we look at the benchmark whose performance is most greatly influenced by the SSIT size, `fpppp`. Figure 6.5 plots false dependencies for `fpppp` across the various SSIT table sizes. As the size of the SSIT shrinks, loads and stores begin to alias. False dependencies are imposed because loads and stores that do not require ordering will use the same store-set table entry as those that do. Figure 6.5 shows that performance is correlated with the number of false dependencies. Adding tags to the SSIT would effectively reduce the number of SSIT entries needed, but would add the cost of a tag array and comparison logic. When the table becomes small, the advantage of the memory dependence predictor is virtually gone, approaching the performance levels of the no-speculation approach as described in Section 3.1.

The performance impact resulting from reducing the size of the LFST is similar to that of the SSIT. The number of entries in the LFST limits the total number of store sets. Reducing the number of total store sets causes loads and stores that suffer memory-order violations to share store sets with one another. The number of store set entries needed is much smaller than the number of entries in the SSIT because store sets can be shared more effectively. If two loads share one store set, it is quite possible that one

Figure 6.4: Peformance Sensitivity to Number of Entries in SSIT

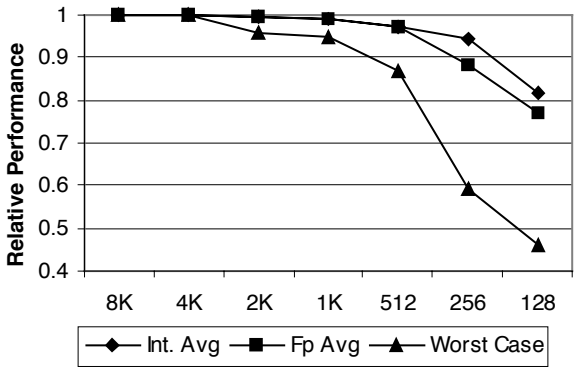


Figure 6.5: Fpppp - Peformance vs. Number of Entries in SSIT

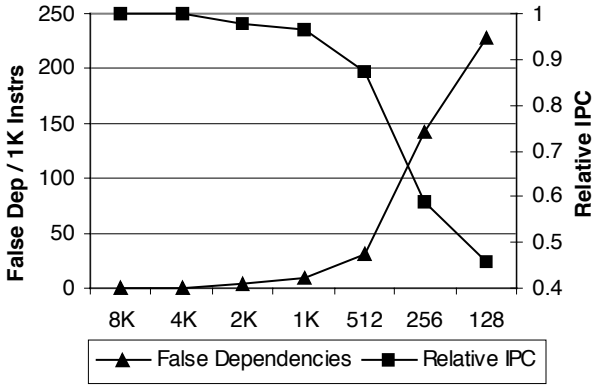
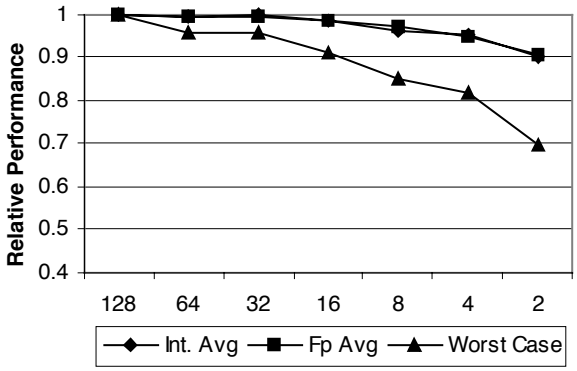


Figure 6.6: Peformance Sensitivity to Number of Entries in LFST



load's conflicting stores and the other load's conflicting stores are never in the processor at the same time. This means that although one store set identifier is shared, temporally one load and its stores never intersect with the other load and its stores; therefore no false dependencies are created. Figure 6.6 shows the relative performance of constructing an LFST with varying numbers of entries. Only 128 entries are needed for full performance. This also reduces the size of the SSIT table because the SSIDs will be only 7 bits.

Our proposed implementation of store sets memory dependence prediction has a 4K entry SSIT and a 128 entry LFST, and uses cyclic clearing to invalidate the SSIT every million cycles. The performance comparison to perfect is shown in Figure 6.7. Since we did not choose to implement the two bit counters, `applu` suffers a performance loss of 14%. The remaining benchmarks are close to optimal performance. Once again, in rare cases, the perfect model has a slight degradation due to the same reason as mentioned in Section 5.2. For comparison, we've included an implementation of the store barrier cache[6] with 32K entries, one for each potential store instruction in a 128K instruction cache. Our store set implementation had an average of 34%, and as high as 100% performance improvement over the store barrier cache for the benchmarks we tested.

7. Analysis

Of all the benchmarks that we considered when evaluating store-set memory dependence prediction, `applu` had the largest performance degradation compared with a perfect predictor. We looked closely at this benchmark to understand why it was suffering a 10-14% performance degradation. Looking back at Figure 5.2, we observed that the infinite store set predictor was producing a fairly large number of false dependencies. Adding two-bit counters to each store in a store set significantly reduced the number of false dependencies, but a considerable number still remained. Also, adding the two-bit counters caused memory-order violations. Although it is a small number (about 1.75 per 1K retired instructions), memory-order traps can have a significant performance penalty. Here, two-bit counters are not predicting memory dependencies well. It turns out that all of the memory-order violations and false dependencies are coming from few load instructions that exhibit a recurring pattern of dependencies.

A good example of the difficulty in predicting memory dependencies in `applu` is found in the routines "blts" and "buts", both of which contain the following loop:

Example 7.1

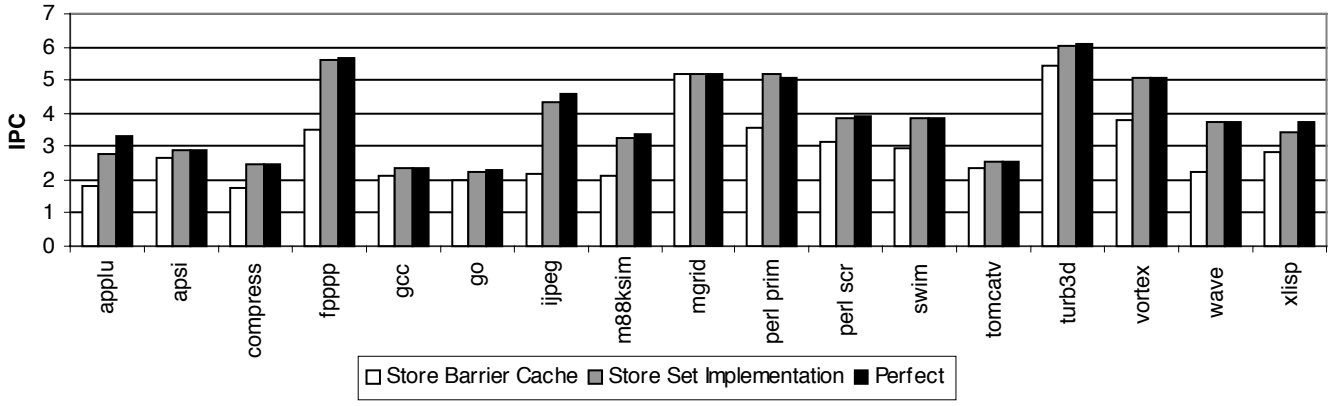
```
0    do ip = 1, 4
1       tmp1 = 1.0d+00 / tmat(ip,ip)
2       do m = ip+1, 5
3          tmp = tmp1 * tmat(m,ip)
4          do l = ip+1, 5
5             tmat(m,l) =  tmat(m,l) - tmp * tmat(ip,l)
```

This triply nested loop creates a scenario where a load and store sometimes conflict. The problem is on line 5. The store instruction that writes tmat(m,l) and the load instruction that reads tmat(ip,l) will sometimes conflict. To see this more clearly, we reduce the problem to:

Example 7.2

```
do ip = 1,4
  do m = ip+1, 5
    do l = ip+1, 5
      load  tmat[ip,l]    ; PC A
      store tmat[m ,l]    ; PC B
```

Figure 6.7: Final Store Set Implementation vs. Perfect



Multiple iterations of the loop are in-flight at once in the processor, which means that multiple instances of the load and store PC are in-flight at once. Consider a point in the loop execution when *ip* equals 3. The inner loops will produce this sequence of the load and store PCs:

Example 7.3

PC			
A	load	tmat(3,4)	ip=3, m=4, l=4 – iter V
B	store	tmat(4,4)	
A	load	tmat(3,5)	ip=3, m=4, l=5 – iter W
B	store	tmat(4,5)	
A	load	tmat(3,4)	ip=3, m=5, l=4 – iter X
B	store	tmat(5,4)	
A	load	tmat(3,5)	ip=3, m=5, l=5 – iter Y
B	store	tmat(5,5)	
A	load	tmat(4,5)	ip=4, m=5, l=5 – iter Z
B	store	tmat(5,5)	

Here the load in iteration Z is dependent on the store in iteration W. When those two instructions execute out of order they cause a memory-order violation, and the store at PC B becomes a member of the load's store set. This is unfortunate, because the load will always wait for the most recent iteration of the store before executing. So, the next time this triply nested loop is executed, the second iteration load will wait for the first iteration store, the third iteration load will wait for the second iteration store and so on. Really, only certain iterations of the load are dependent on certain iterations of the store. The store set implementation that we have chosen does not keep track of iteration information. Most of the parallelism in the loop is serialized in hardware, resulting in a severe performance degradation for this loop. In fact, this loop executes twice as fast with the perfect memory dependence predictor as it does with the store set predictor. To eliminate this problem, a compiler could fully unroll the triply nested loop. This would create separate PCs for the loads and stores in different iterations. Then the store set memory dependence predictor would allow a load to specify which specific store conflicts with it. After we unrolled the triply nested loop manually, there was no performance difference between the perfect memory dependence predictor and our store set implementation.

One could also imagine a memory dependence predictor that could detect such patterns of conflict. This is a possible direction for future work.

8. Conclusion

Memory dependence prediction is important for future out-of-order processors. Poor or no memory dependence prediction will severely constrain the IPC of future designs. We showed that using the concept of store sets as the basis for memory dependence prediction in an idealized structure yields nearly optimal prediction accuracy. We discovered that an important advantage of store sets is that they handle cases where multiple loads depend on one store and one load depends on multiple stores. We then developed a low cost implementation based on direct mapped structures by: 1) limiting the total number of loads that can have their own store set, 2) limiting stores to being in at most one store set at once, and 3) constraining stores within a store set to execute in order. To retain the advantage of idealized store sets, we developed a set of store set assignment rules that enabled multiple loads to depend on the same store through dynamic store set merging. Our memory dependence predictor exhibited nearly optimal performance in a large instruction window, superscalar processor when compared with a perfect memory dependence predictor.

9. Acknowledgments

We thank John Edmondson and Bruce Edwards for their consultation and contribution of ideas to this work. We also thank Tryggve Fossum for supporting this effort. This paper has benefited from being reviewed by Eric Borch, Robert Cohn, Jennifer Cook, David Goodwin, Judy Hall, Milo Martin, Andreas Moshovos, Doug Sanders, and the anonymous ISCA-25 reviewers. We thank them for their time, valuable insights and corrections.

References

[1] D. Leibholz, R.Razdan. The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor. In Proc. *IEEE CompCon'97*, Feb. 1997.

[2] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *Hot Chips VII*, 1995.

[3] D. Hunt. Advanced Performance Features of the 64-bit PA-8000. In Proc. *IEEE CompCon'95*, Mar. 1995.

[4] Intel Corporation. *Pentium(R) Pro Developer's Manual*. McGraw-Hill, June 1997.

[5] A. Moshovos, S. Breach, T Vijaykumar, G. Sohi. Dynamic Speculation and Synchronization of Data Dependences In Proc. *ISCA-24*, June 1997.

[6] J. Hesson, J. LeBlanc, 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.

[7] A. Moshovos, G. Sohi. Streamlining Inter-operation Memory Communication via Data Dependence Prediction. In Proc. *MICRO-30*, Dec. 1997.

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

[9] S. McFarling. Combining Branch Predictors. *WRL Technical Note TN-36*, June 1993.

[10] J. Smith. A Study of Branch Prediction Strategies. In Proc. *ISCA-8*, May 1981.

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

[12] M.Lipasti, J. Shen. Exceeding the Dataflow Limit via Value Prediction. In Proc. *MICRO-29*, Dec. 1996.