A Performance Simulation of Store Sets using SimpleScalar

Thomas Pratt and Chung-Hsuan Tung
Duke University

Abstract

Memory dependences present a challenge to dynamically scheduling load/store instructions. Since the memory address cannot be resolved until runtime, the dependence is hidden until the instruction is executed. One way to resolve memory dependences is load speculation, in which the core speculatively executes a load before the addresses are resolved. Store sets [1] are a mechanism to record memory dependences and allow load speculation based on the history of the stores that the loads depended on.

In this report, we simulate the use of store sets in the out-of-order performance simulator SimpleScalar [2]. We construct two data structures to record the history of target addresses for stores and associate them with loads. At the end of the report, we experimentally evaluate the performance gained by using store sets, explore the design space of the two data structures, and evaluate the store sets under various cache access latencies.

1. Introduction

Modern CPUs use out-of-order execution to exploit instruction-level parallelism (ILP). Most false dependences between instructions can be resolved by register renaming, which enables the core to freely reschedule instructions with false dependences and improves its exploitation of ILP. However, potential memory dependences cannot be resolved using this strategy. Memory dependences place a burden on dynamically rescheduling memory access instructions, especially load instructions. Loads which take a register as an input for their memory addresses are dependent on any stores that write to the same address in memory. The processor can't know for sure that these dependences exist before the values of the target addresses are resolved.

Modern processors use two main strategies to accelerate loads without violating memory dependences. One strategy is the load/store queue (LSQ), a data structure that allows the processor to bypass memory access by comparing the target addresses of loads and stores. The other strategy is load speculation, in which the processor executes loads before their memory dependences are resolved, predicting that there will be no dependences. Incorrectly speculating on a load produces incorrect values for instructions that depend on the load, because the load does not receive the value from the most recent store it depends on and therefore violates program order. This is called a *memory-order violation*. Incorrect speculation ("misspeculation") is inevitable, so processors that use load speculation must be prepared to recover when they execute incorrect instructions.

Store sets are a mechanism that improves the speculative execution of loads. Store sets were originally proposed by Chrysos and Emer in the paper "Memory dependence prediction using store sets" [1]. The idea is to associate stores' PCs with the loads they depend on, so that the load can be rescheduled without waiting for independent stores. Store sets act as a memory dependence predictor that allows the processor to speculatively execute loads more accurately.

The rest of the report is structured as follows. In section 2, we briefly describe the concept of store sets. In section 3, we introduce the SimpleScalar simulator as the platform we will use to evaluate the effectiveness of store sets. We explain the details of our implementation in section 4 and summarize our experiment results in section 5. The conclusion will be in section 6.

2. Store Sets

Most instruction set architectures (ISA) allow load and store instructions to specify a target memory address using a register. This flexibility is useful for programming, but it hides possible memory dependences from the

microarchitecture until the value in the operand register is calculated (i.e. the target address in memory is resolved). This hidden dependence is challenging for dynamic rescheduling since it limits the range over which the load can be rescheduled.

The most conservative way to ensure the correctness of a program is not to reschedule loads in runtime ahead of any store with an unresolved target address. However, this strategy will force some loads to wait for non-producing stores, which is unnecessary. In addition, loads are often long latency operations, and loads are common in most programs, so we want to issue them as early as possible. Resolving false memory dependences has the potential to be a massive boon for performance because it allows loads to be freely rescheduled (or "speculated on") with no consequences if we predict correctly. This increases the scheduler's ability to exploit ILP.

2.1 Terminology

If a load depends on a particular store, we call this store a "producing store" with respect to that load. If a load does not depend on a particular store, we call it a "non-producing store" with respect to that load. If a store set predicts that a load would depend on a particular store, we call this a "predicted producing store," if it predicts the opposite, we call this a "predicted non-producing store." In general, the word "prediction" refers to store sets predicting memory dependences unless noted otherwise.

2.2 Rescheduling Loads

The earliest time a load can be rescheduled to is immediately after its producing store. However, some non-producing stores may exist between the load and its closest producing store, preventing the machine from rescheduling the load into the best place.

Store sets [1] are a mechanism which, on a given load, predicts whether nearby stores are producing or non-producing so that the processor can schedule the load as early as possible. When a load is available for rescheduling in the instruction window, its associated store set is searched to find the closest predicted producing store (which marks the earliest possible place where the load can be rescheduled). This prediction is based on the store's PC.

2.3 PC-based Prediction

Store sets take advantage of the fact that most loads are independent of recent stores and that memory dependencies are highly correlated to the loads' and stores' PCs. For example, array arithmetic within a loop may create a load-store pair where a dependence exists and can be predicted by PCs. An example of this is shown in figure 1.

```
1 A[0] = 0;
2 for (int i = 1; i < 3; ++i)
3     A[i] = A[i-1]+1;
```

Example C code for a simple array manipulation

```
1 ST  0, A[0]     // store 0 to A[0]
2 ADDI $0, 1, $1  // set iterator $1 to 1
3 LOOP:
4 SUBI $1, 1, $2  // $2 = $1 - 1
5 LD  A[$2], $3   // load A[$2] to $3
6 ADDI $3, 1, $4  // $4 = $3 + 1
7 ST  $4, A[$1]   // store $4 to A[$1]
8 ADDI $1, 1, $1  // $1 = $1 + 1
9 BLT $1, 3, LOOP // Jump to LOOP if $1 < 3
```

Example assembly code for a simple array manipulation

Figure 1: The load in assembly code line 5 depends on the store in line 7. The dependence can be predicted with PCs, even though the target addresses keep changing.

Figure 1 shows a common case in which memory dependence can be resolved by observing the PCs of memory access instructions. The C code in the example shows simple array arithmetic with an inter-iteration dependency. In the corresponding assembly code, for each iteration, the load in line 5 (PC equals 5) depends on the store in line 7 (PC equals 7) from the previous iteration. The dependence can be easily resolved if the load has recorded the PC of the producing store in its store set, which lets us predict that the load is dependent on this store.

2.4 Store Set Details

Store sets are records of the history of producing stores for each load. They are updated when misspeculations occur. A load is allowed to be rescheduled at the very beginning of the scheduling window if there is no predicted producing store in front of it. This strategy will

cause misspeculations and memory-order violations at first since the store sets start out as empty. When the target addresses of the previous (in program order) stores and the load are resolved, we check to see if the load was incorrectly rescheduled before a producing store. If so, we recalculate all possible incorrect instructions resulting from this incorrect load value, and we add the PC of the producing store to the load's store set. The next time we see a load at the same PC, its store set is searched to predict whether rescheduling will cause a memory-order violation.

3. SimpleScalar Simulator

SimpleScalar is a CPU performance simulator written in C that allows for multiple design configurations. It simulates various styles of core and produces performance statistics. High-level design decisions such as pipeline width, scheduling window size, cache access latency, and cache size are configurable in SimpleScalar. In this project, we use SimpleScalar's out-of-order simulator `sim-outorder.c` to evaluate the theoretical performance gain from implementing store sets.

3.1 Register Update Unit (RUU)

The most relevant feature of `sim-outorder` to understand for our project is the *Register Update Unit* (RUU).

`sim-outorder` uses the RUU to trace the instructions in program order when implementing out-of-order execution. The RUU combines the functionality of the instruction buffer and the reorder buffer, serving as the base data structure that holds all the visible instructions in the processor.

3.2 Pipeline Simulation

In the instruction *fetch* stage, the processor fetches instructions into a queue. It fetches as many instructions as possible within the limited scope of a single branch prediction and the cache line access. After an instruction is fetched, it is dispatched from the queue into the RUU.

During the *dispatch* stage, an instruction is placed into the RUU in program order. The processor also links each entry (called a *reservation station*) in the RUU to their dependencies. These links allow us to resolve register-level dependencies in the issue stage. At the same time, the load/store instructions are divided into 2 stages: one to compute its effective address, and one to access memory. Address computation is handled by RUU as a regular arithmetic operation, while the memory access is handled by the LSQ to ensure correct memory order. In this situation, the opcode fields of the load/store instructions in the RUU reservation stations will be replaced by simple additions that represent their memory address computations.

In the *issue* stage, the processor awakens instructions whose input operands are all ready and issues them to the execute stage if the necessary functional units are available. Any time the processor sees a store instruction, it simply marks the store as completed, and we record the PC-target address pairs for our store set implementation. These value pairs are stored in the SEAT described in section 4.2. For any load instruction, we apply store sets to recalculate its latency[1]. All of these changes are made with the goal of gathering statistics about our performance simulation and not executing instructions. In truth, sim-outorder has no execute stage, and instructions are executed in the dispatch stage.

The *writeback* stage is for instructions to update the register value. Sim-outorder updates the readiness of the operands for instructions in the RUU based on the results of previously executed instructions. This is done by tracing back the dependence link created in the dispatch stage.

The instructions are retired from the RUU in the *commit* stage. Stores now write their value into the data cache and retire from the LSQ.

3.3 RUU Implementation

The RUU is a circular queue that contains instructions in program order. Each entry of the RUU is a C struct, `RUU_station`, and the entire RUU is an array of `RUU_station`

[1] This recalculation accounts for the load latency calculated by sim-outorder, which in turn accounts for LSQ bypassing, cache access time, and TLB access time.

structs. Each `RUU_station` contains an instruction's bit pattern, opcode, PC, flags, and microarchitectural information. Micro-architectural information includes program sequence, dependence links, and the target address if the instruction is a load or store. There are flags that tell us whether the instruction is also in the LSQ, or whether it is in a special internal state such as mispredicted, ready, issued or not, complete or not. The LSQ shares the same structure as the RUU. The entire program relies on the RUU to maintain program order and uses ready queue and event queue to trace the execution order.

4. Implementation Details

Our implementation of store sets in SimpleScalar[2] is not a truly functional implementation, but rather a performance simulation. Our implementation gives us a perfectly accurate picture of whether store sets would have predicted a memory dependence on a given load, and whether that prediction was correct. It does not actually reschedule any loads or execute operations any earlier. A truly functional implementation is beyond the scope of this project.[3]

4.1 Interpreting performance change

The latency of loads is one limiting factor on our performance gain; if the latency of a load is 1 cycle (i.e. it hits in an L1 data cache with a hit time of 1), we cannot save more than 1 cycle by rescheduling that load. The main point of rescheduling loads is to hide the latency of the cache accesses that they require so that their dependent instructions do not have to wait.

The other limiting factor is the amount of space we have to reschedule any given load. We calculate this by checking if our memory dependence prediction would have enabled us to schedule the load earlier, and if so, track how many cycles earlier we could have scheduled it.

It may be the case that we reschedule the load and do not save any time. In fact, this is fairly common. It occurs when SimpleScalar would have scheduled the load's dependent instructions far enough in the future that the load would have completed before its dependents regardless of rescheduling. It is beyond the scope of our project to enable the simulator to check whether this occurred because we can only perform checks on an instruction while it is in-flight. If we had produced a truly functional implementation of store sets, this case would have been naturally accounted for.

Additionally, loads may depend on arithmetic operations that compute their effective addresses, which may in turn depend on earlier instructions. We've decided to ignore this dataflow problem and assume in our calculations that the load's non-memory dependencies are all satisfied before it enters the scheduling window.

Instead, we simply count the cycles that we would be able to save in the best case, where otherwise waiting dependent instructions would have been able to execute as soon as possible. This means that we interpret our results as the *maximum possible performance gain* from rescheduling loads according to store set memory dependence predictions.

4.2 Data Structures

We simulated store sets in SimpleScalar using two main data structures. One is the Store Entry Address Table (SEAT). This is a small queue of recently-executed store instructions. Each entry in the table contains the PC and the memory address of a single store, as md_addr_t values. It also contains the cycle on which the store was executed and added to the table, which we use to maintain a least-recently-used (LRU) replacement policy in the SEAT. The SEAT is updated every time we execute a store.

The other data structure is the Load Dependence Table (LDEPT). The LDEPT is more difficult to maintain than the SEAT. The LDEPT is a small queue of recent loads, each of which contains a pointer to a linked list of recent stores that it depends on (its store set). These recent stores are represented by SEAT entries.

[2] Our code can be found at `https://gitlab.com/2021_Duke_AdvCompArch/cs550_storesets/` .

[3] The main reason we decided it was unreasonable to write a truly functional implementation is that it would require the ability to misspeculate on loads and recover from misspeculation, which SimpleScalar has no built-in tools to implement. SimpleScalar currently performs no speculation on loads by waiting until all of a load's possible memory dependencies are resolved [2].

The LDEPT also uses an LRU replacement policy and keeps track of the cycle on which each load was added. The function `ldept_clean()` frees all the memory that is allocated with `calloc()` in the store set of the evicted load on replacement, which is not necessary for replacement in the statically-allocated SEAT.

Finally, each load in the LDEPT has a property that tracks the current number of stores in its store set (the "length" of the store set).

The SEAT and the LDEPT must each be at least as big as the RUU because otherwise we can't account for all the addresses in the RUU. As we'll see later, we wouldn't be able to determine when we've misspeculate on a load. This constraint becomes necessary to evaluate performance. This is a limitation of our implementation in the simulator; real hardware could still check for misspeculation if its data structures were smaller than its RUU.

4.3 Maintaining the LDEPT

Each time we execute a load, we add it to the LDEPT if its PC is not already present and replace the least recently used element. We then resolve the load's effective address. Once we know the address, we perform checks on the RUU to see if we've saved cycles or misspeculated. These checks are described in section 4.2.

After we've searched the RUU, we update the LDEPT unconditionally. The original paper updates a store set conditionally when they misspeculate, but this is simply done to save time; we can update the LDEPT unconditionally with no problems because no updates will need to happen unless there has been a misprediction.

To update the LDEPT, we search the SEAT for any stores with a matching effective address and add them to the current load's store set in the LDEPT, if they aren't already present.[4]

4.4 Calculating performance gain from rescheduling loads

We maintain the LDEPT from within `sim-outorder.c`, in the `RUU_issue()` function. This is because `RUU_issue()` is where the latency of loads is determined. Recall that the latency of a load is one limiting factor on performance gain.

We want to know whether the current load depends on any of the stores that are executing in its scheduling window. The RUU contains all the currently in-progress instructions. So, on a load, we can check the current contents of the RUU to see if any of the PCs present in the RUU match those in the load's store set.[5] If any of these store PCs match, we predict that this store is a producing store for that load. We track the position of the closest producing store and the closest non-producing store as we scan the RUU.

If the nearest store younger than the load is predicted to be a producing store, we of course cannot reschedule the load. So in this case we do nothing, and the latency of the load does not increase, since the default behavior of sim-outorder is to wait until all memory dependences are known before executing a load [2][3].

If the nearest store younger than the load is predicted to be a non-producing store, we predict that we can reschedule the load. We can also reschedule the load if there are stores in the RUU, but we predict that none of them are producing stores. These cases are equivalent except for the distance across which we can reschedule the load: if there is a producing store, we have to schedule the load behind that store, and if not we can schedule it as early as the scheduling window allows. Figure 2 is a visual representation of the decision flow described above.

[4] Once a store is present in a load's store set, we know the load has depended on that store once. If we don't update the address of the store, we might sometimes over-predict memory dependency because a store's effective address could change such that the load would no longer be dependent on it. However, this is an infrequent enough case that maintaining strict correctness in the LDEPT is not worthwhile in practical use cases, so we simply keep the same PCs of stores in the load's store set until it is evicted from the LDEPT.

[5] In truth, we only check the stores that are older than the current load in the execution order; that is, we increment the current `RUU_head` until we reach the load instruction we're interested in.

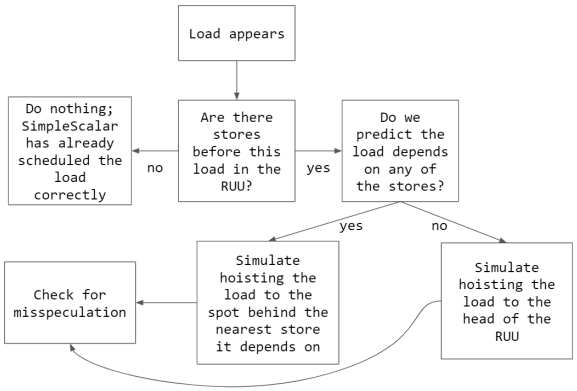

Figure 2: Flowchart of the decisions made when scanning the RUU during a load.

In a real processor, the scheduler could freely speculate on the load. This means it would hoist the load up to be sooner in the execution order, which means that instructions depending on the load could execute sooner. We simulate this behavior by tracking the number of cycles each of the loads would be able to be moved.

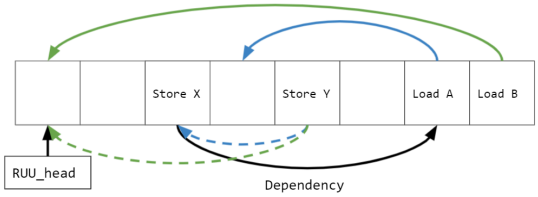

Figure 3: 8-entry RUU with example loads and stores. Load A depends on Store X. Solid arrows represent the maximum distance each load could be moved, while dashed arrows are the actual distance calculated. Blank RUU spots represent non-memory instructions.

Figure 3 shows the process of measuring saved cycles. We conservatively assume that sim-outorder has scheduled both loads as early as possible. Sim-outorder tries to do this: in `readyq_enqueue()`, the scheduling policy is described as "memory and long latency operands, and branch instructions first" [2]. In figure 3, we show an example where memory operations do not come first—it would be possible to move either Load A or Load B to the spot immediately after Store Y. Note that it's possible for this to occur even under sim-outorder's current scheduling policy. For one thing, both loads cannot occupy the same spot in the scheduling order, so one will necessarily occupy a location that is suboptimal to the location we've assumed. For another, the scheduler may decide that a long non-memory operation like multiply should occupy that spot.

In the default sim-outorder, any load is forced to wait behind the nearest store older than itself, whether the store is producing or non-producing. We calculate saved cycles as the distance between this nearest non-producing store and the spot the load would have been rescheduled to. This is how our assumption (that the load is scheduled as early as possible) is realized in our calculations.

One aspect of this calculation that may seem strange is the decision to use the positions of two stores to calculate the number of cycles saved, rather than the positions of the spots where the loads would go. We also assume that saving a cycle is equal to moving a spot up in the execution order, which is not necessarily true because of parallel execution. We address these decisions in detail in section 4.5.

Another important case to discuss is when two loads are near each other in the RUU and are rescheduled to two earlier places. Both loads might claim to save 3 cycles, but this does not necessarily mean we have saved 6 cycles, because some of the saved cycles could overlap (i.e. the same cycle of time could be counted twice). In short, we shouldn't double-count the number of cycles we've saved.

The true number of cycles saved in this situation is nebulous, and depends not just on the original and final places of each load but also the various producing and non-producing stores that could be present in the RUU for each load. Add to this the problem that more than just two loads could be near each other in the RUU, and we can see that there are an exponentially increasing number of checks that need to be made. Unfortunately, we were not able to solve this problem directly or devise a reasonable strategy to mitigate it. Because we already knew we would be calculating maximum performance increase, we decided to leave this corner case alone. This is another casualty of our ultimately necessary decision to simulate the behavior of store sets rather than implementing them outright.

Finally, if the number of cycles we've saved by moving the load is larger than the memory access time (latency) of that load, we

replace the number of cycles saved with the latency of the load, because all we've managed to do in this case is hide the latency of the load.

4.5 Handling Misspeculation

Here we introduce our process for detecting cases where the store sets would produce a misspeculation. This happens when a load's store set does not contain one or more producing stores for that load in the RUU. If this happened, a true implementation of store sets would allow speculation on the load, but rescheduling the load before its producing store would cause a memory-order violation.

When we're scanning from the head of the RUU to our load, we make note of every store that we find. If the store set predicts the store to be non-producing, we check the store's effective address. If the store's effective address is actually the same as the load's effective address (even though we predicted that it was different), then we have misspeculated.

Now, instead of saving time, we have to flush some portion of the RUU and recalculate the flushed instructions, which loses time. The store that we're most concerned with is the oldest misspeculated store that is younger than the position the load was rescheduled to. This store represents the earliest memory-order violation that we've created by misspeculating.

In cases where we haven't misspeculated, rescheduling the load will allow us to reschedule other dependent instructions earlier in the execution order as well. This saves time in cases where we speculate correctly, but during misspeculation recovery, it means that we'll need to flush from the earliest memory-order violation all the way to the tail of the RUU, instead of only to the original load position. There could be any number of instructions dependent on this incorrect load in-flight by the time we detect the misspeculation. For modern processors, it's not worth the extra complexity of detecting which of these instructions were correct to avoid redoing work; the optimal choice for performance is to flush the pipeline. Figure 4 shows an example of flushing the RUU after misspeculation.

Store Z is a memory-order violation because Load A will incorrectly receive its value from Store X in the given execution order. Everything after Store Z (in program order) will need to be flushed. The flushed instructions are shown in red. Store Y is not a memory-order violation because Load A does not depend on it.

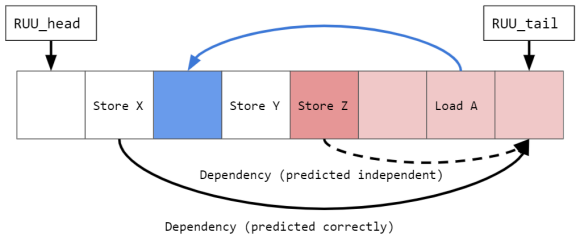

Figure 4: 8-entry RUU during misspeculation.

To calculate how much time we have lost by misspeculating, we simply assume that we've lost one cycle per instruction that was flushed. This is again a conservative assumption; in the best-case scenario we would lose fewer cycles because the processor would have scheduled some of these instructions to execute in parallel and could do so again during re-execution. In fact, for any program, the average cycles we lose by flushing the RUU is equal to the number of instructions flushed times the program CPI.

After we misspeculate, we update the LDEPT by searching the SEAT for any stores with the current load's effective address.

This concludes our discussion of calculating simulation statistics. Based on the implementation discussed in this section, we produced a variety of experiments and statistics.

5. Experiment Results

We ran three benchmarks to test the performance of store sets. The benchmarks are gcc, go, and anagram. The main variables we tested were the size of the SEAT and LDEPT data structures. We also changed the hit latency of the DL1 cache.

5.1 Instructions Per Cycle (IPC)

Instructions per cycle is a straight-forward evaluation of a micro- architecture design. Here we observe that IPC varies across different benchmarks. The graphs indicate that the performance of store sets varies widely when executing different applications. To better compare how store sets can help performance in

all cases, we normalize IPCs based on benchmarks.

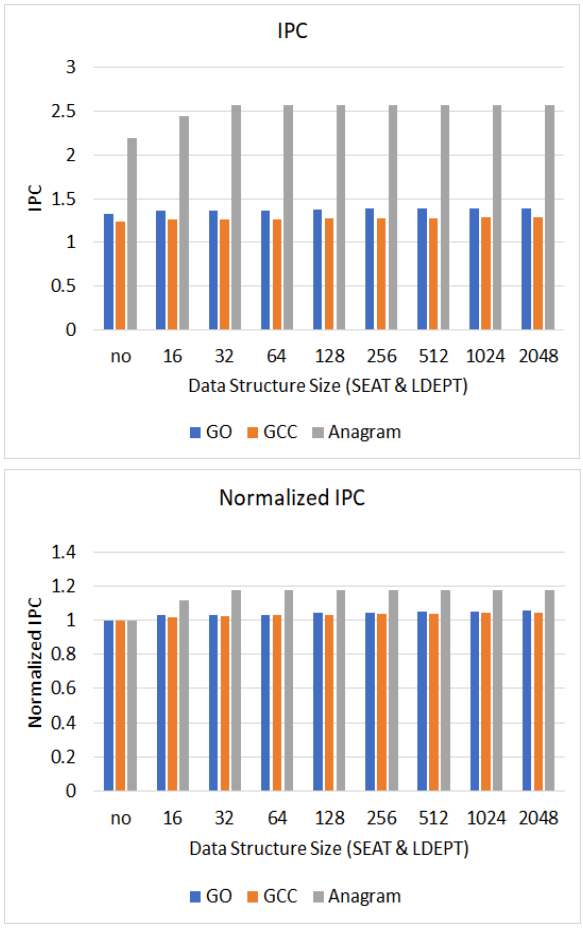

Figure 5. IPC and Normalized IPC for go, gcc, and anagram at various data structure sizes.

As we can see in Figure 5, IPCs increase somewhat by adding store sets. The increase in IPC was not as drastic as the original paper claims [1]. IPCs also do not grow accordingly when increasing the SEAT and LDEPT size. We explain this trend together with the misspeculation rates shown in Figure 6.

As discussed in section 4.5, mis-speculations decrease performance. More accurate speculation should intuitively lead to better performance. The larger size of the SEAT and LDEPT means a longer history we can use to predict memory dependences. We measure the numbers of misspeculations per 1k instructions in Figure 6 to demonstrate how increasing the sizes of these data structures can reduce the number of misspeculations.

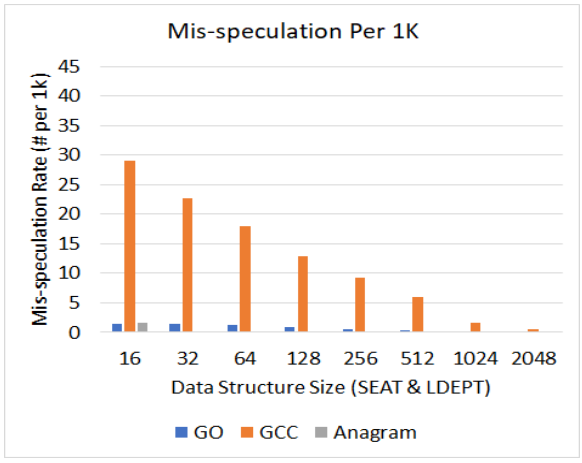

Figure 6: Misspeculations per 1K instructions on go, gcc, and anagram with various data structure sizes.

In Figure 6, misspeculations per 1k instructions decrease drastically as the sizes of the SEAT and LDEPT increase, but this decrease in misspeculations does not necessarily translate to an increase in performance. The number of misspeculations is very low even when the data structures are small, and even for the go benchmark which shows the most drastic change. In fact, increasing the data structures to sizes of 32 was enough to eliminate misspeculation entirely for the anagram benchmark. Misspeculations are a significant source of overhead, especially for larger RUU sizes (data not shown), and reducing misspeculations is a good way to improve performance with store sets. But increasing the sizes of the data structures produces diminishing returns, and not just because there are fewer and fewer misspeculations to shave off. The total number of correct speculations also decreased across all benchmarks as the sizes of the data structures increased, although not as significantly as misspeculations. In anagram, we reach optimum IPC with both data structures at sizes of 32.

5.3 Store Set Length

We have already found that a longer history of memory dependence will increase speculation accuracy. The average length of store sets when they are evicted is a measure we can use to evaluate how much history the store sets retain. A longer store set almost always means an entry in LDEPT has been updated

more times (and therefore been misspeculated on more times) before the entry is evicted by LRU replacement. As we can see in Figure 7, larger data structure sizes can effectively store longer history without changing the design of the store sets themselves at all. An interesting observation is that the store set length for the anagram benchmark decreases after the data structure size is larger than 512. This is because anagram is a relatively short benchmark, so the data structures are large enough to record all the history without evicting almost any entries.

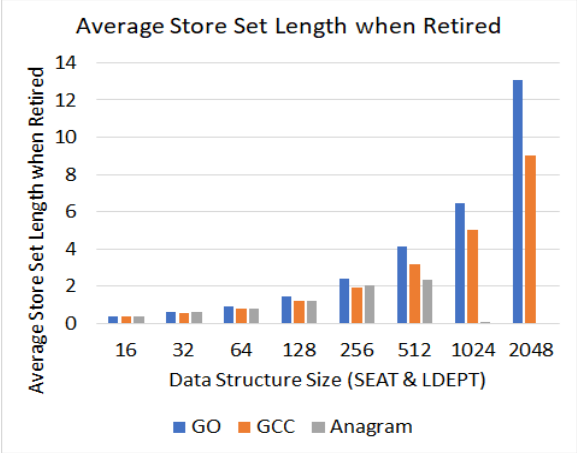

Figure 7: Average store set length when an entry of LDEPT is retired on go, gcc, and anagram.

5.4 SEAT Size

In previous sections, we changed the size of the SEAT and LDEPT together to see how these data structures can help reduce misspeculation and enable the store sets to remember longer history. In sections 5.4 and 5.5, we fix the size of one of these data structures to the default 16 entries and vary the size of the other data structure across experiments.

In Figure 8, the normalized IPCs look similar to Figure 5, but in truth the values are slightly lower. This is because the fixed size of the LDEPT limits the number of store sets that can exist in the system at any given time, which reduces the LDEPT's ability to record history.

In addition, increasing the size of the SEAT while holding the size of the LDEPT constant produced a small but constant decrease in IPCs. The most likely reason for this is that loads are replaced more often due to the size increment of the SEAT. When the SEAT is larger than the LDEPT, a load in the LDEPT may detect dependences on earlier stores when searching the SEAT to update its store sets. However, a longer history isn't always a good thing: if stores are not present in the RUU at the same time as the load, we don't gain anything from their presence in the store set. Stores with the same PC can show up with different target addresses, especially with a long interval. However, the LDEPT is updated by searching the SEAT by a load's target address. This means a load will learn ancient store addresses that are no longer relevant and can cause misspeculation.

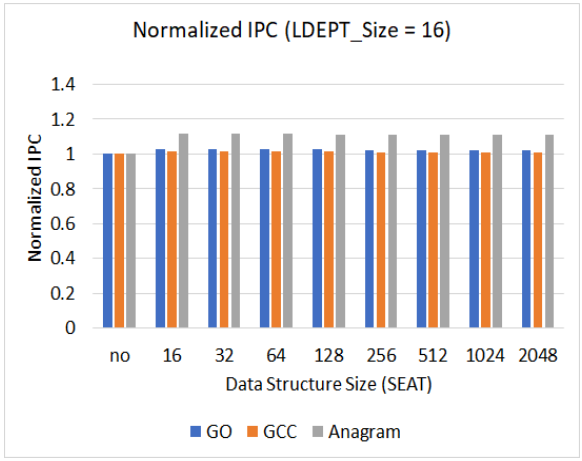

Figure 8: Normalized IPC on go, gcc, and anagram for various SEAT sizes.

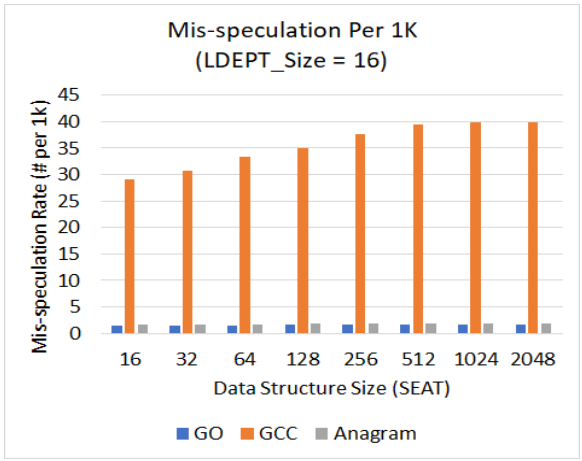

Figure 9: Average store set length when an entry of LDEPT is retired on go, gcc, and anagram for various SEAT sizes

As we can see in Figure 9, the misspeculation rate trends up as SEAT size increases. Recording too much history in the SEAT without enough space in LDEPT may

degrade performance. In Figure 10, the longer the store sets are when they are retired, the more predicted producing stores a load may see. However, these store sets cannot effectively reduce the misspeculation rate due to unbalanced data structure sizes—in other words, these are bad predictions.

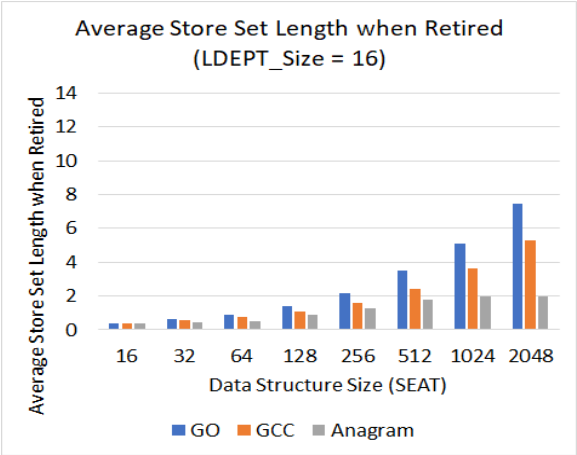

Figure 10: Average store set length when an entry of LDEPT is retired on go, gcc, and anagram for various SEAT sizes.

5.5 LDEPT Size

There is no point in increasing the SEAT size alone. In this section, we discuss whether increasing the LDEPT size alone will boost performance.

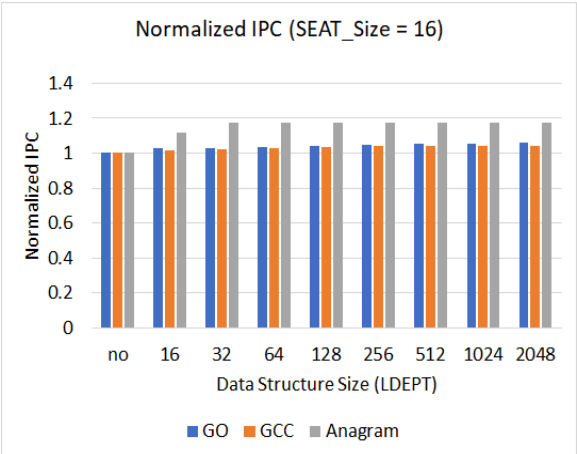

Figure 11: Normalized IPC on go, gcc, and anagram for various LDEPT sizes.

By comparing Figure 11 with Figure 5 and Figure 12 with Figure 6, we can conclude that the trends in increasing LDEPT size are very similar. Therefore increasing LDEPT size alone at least does not hurt the performance, and is likely driving the performance gain in the earlier examples. It's possible to get the benefit of enlarging both data structures by increasing the size of only the LDEPT.

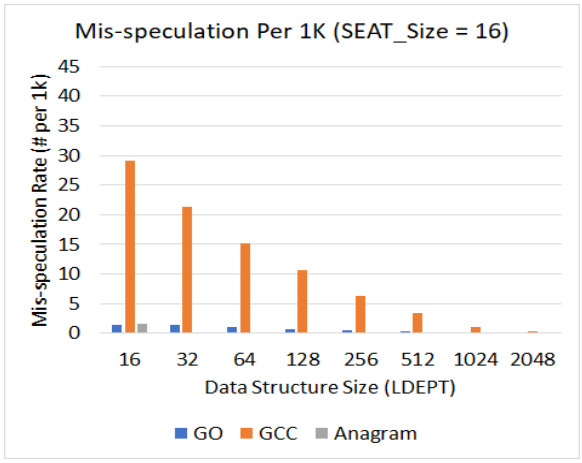

Figure 12: Normalized IPC on go, gcc, and anagram for various LDEPT sizes.

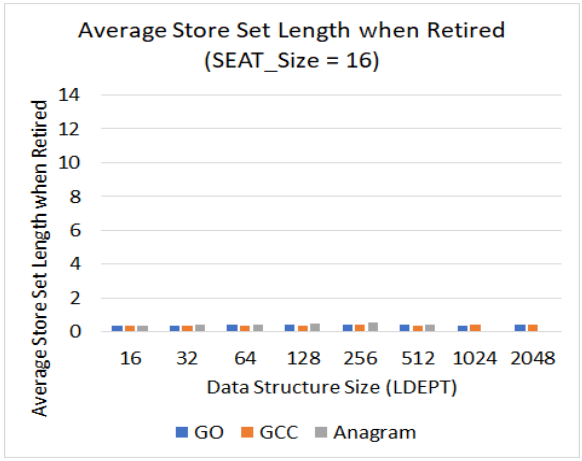

Figure 13: Average store set length when an entry of LDEPT is retired on go, gcc, and anagram for various LDEPT sizes.

We can see that the store sets are much shorter at retirement when we increase the size of the LDEPT than when we increase the size of the SEAT. An interesting property shown in figure 13 is that increasing the LDEPT size causes the average store set length for evicted entries in the LDEPT to decrease. Since the LDEPT is much larger than the SEAT, we think this happens because the LDEPT has enough capacity to record all the history and only a small number of entries will be evicted. Since the LDEPT is so large and we update the store

set only when the load associated with that entry is issued, it's reasonable that entries evicted by LRU replacement will contain smaller store sets (less history): these will be the loads that are accessed the least frequently. Compare this with figure 10.

5.6 Cache Latency

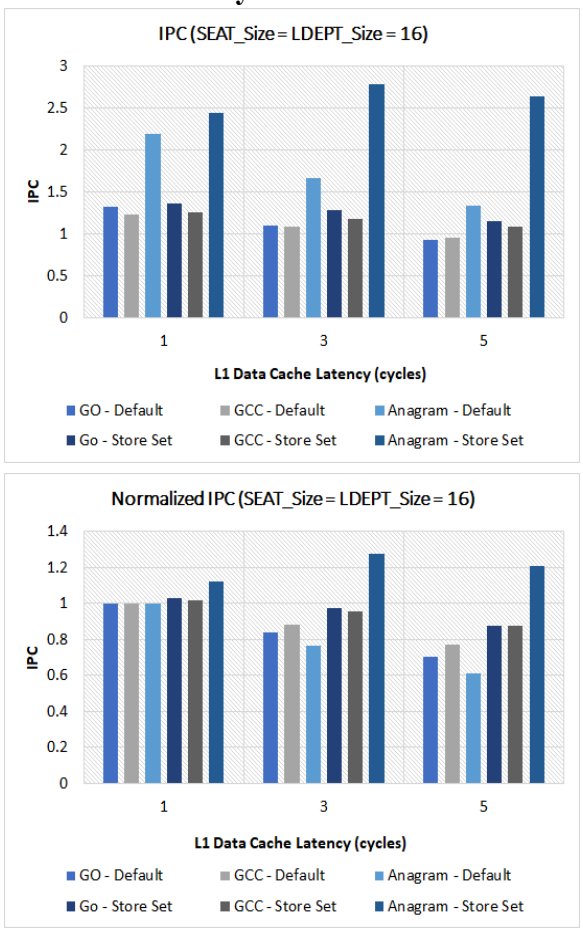

Figure 14: IPC and Normalized IPC for go, gcc, and anagram at various L1 data cache latencies. The normalization is benchmark-based; all values are normalized to the default case with L1 Data Cache Latency equals 1.

The cache access time is the main overhead of a load instruction, and hits are the most common case. By increasing the latency of the L1 data cache in SimpleScalar, we should be able to magnify the performance change that we observe. In Figure 14, we see that increasing the cache access latency hurts the performance of go and gcc. However, store sets significantly lessen the negative impact of the change. This is because, the longer the cache latency, the more it is worthwhile to schedule the load as early as possible—despite the chance to misspeculate and pay the penalty. In fact, the total misspeculation overhead is independent of the increased cache latency.

A special case is that anagram increases its IPC even when the cache latency increases. One possible reason why is that a longer data cache latency affects the execution order of the instructions depending on the loads in a way that we could not predict. It could also be that latencies are double-counted as discussed in section 4.4.

6. Conclusion

Chrysos and Emer claim that store sets have the potential to provide near-optimal memory dependence prediction and significantly increase the performance of load speculation schemes [1]. We evaluate this claim by implementing a simulation that finds the maximum theoretical performance of store sets without the cost-reducing limitations imposed by the original paper. As a result, we conclude that store sets generally improve performance, especially when cache access latencies are large. However, we don't see as much performance gain as the original paper claimed. Also, we explore the design space by varying the size of our data structures, and we concluded that the LDEPT should be larger than the SEAT to fully leverage the benefit from the store sets.

7. References

[1] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235), 1998, pp. 142-153, doi: 10.1109/ISCA.1998.694770.

[2] T. Austin, E. Larson and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling," in Computer, vol. 35, no. 2, pp. 59-67, Feb. 2002, doi: 10.1109/2.982917.

[3] S. Wang, "A Quick SimpleScalar Tutorial," *Computer Architecture and Organization - Shuai Wang*, 2016. [Online]. Available: https://cs.nju.edu.cn/swang/CA_16S/simplescalar_tutorial.pdf. [Accessed: 21-Nov-2021].