Memory Bypassing: Not Worth the Effort*

Gabriel H. Loh
Yale University
Dept. of Computer Science
gabriel.loh@yale.edu

Rahul Sami
Yale University
Dept. of Computer Science
rahul.sami@yale.edu

Daniel H. Friendly
Yale University
Dept. of Electrical Engineering
daniel.friendly@yale.edu

Abstract

Memory dependence prediction establishes a read after write dependence between a store and a load instruction. If the processor accurately predicts the data dependence between a store and a subsequent load, we can completely bypass *memory and forward the data directly from the store's producer to the load's consumer. Our simulation studies show that even in the case of processors with oracle dependence predictors, memory bypassing only provides a 2.3% IPC improvement over dependence prediction alone. Given the small potential gains in the ideal case and the hardware complexity required to implement memory bypassing, we argue that computer microarchitects should focus on memory dependence prediction and ignore memory bypassing.*

1. Introduction

A memory dependence typically presents an obstacle to greater performance. A load instruction must wait for the most recent, earlier store to the same address before issuing. At the same time, we want unrelated loads to issue as soon as their arguments are ready. *Memory dependence prediction* attempts to determine whether a load instruction should issue immediately after its arguments are ready, or whether the load instruction should wait due to an earlier store to the same address. Issuing the load too early leads to memory ordering violations, and forcing the load to wait may cause false memory dependences.

Researchers have studied how to accurately predict memory dependences to enable more aggressive out-of-order issue of memory instructions. Chrysos and Emer proposed using *Store Sets* to predict memory dependences [3]. They show that an out-of-order processor using store sets achieves performance levels that are very close to an ideal processor that is capable of perfect dependence prediction.

*This research was partially supported by NSF Grant CCR-9702281, NSF Grant CCR-9702980 and by the DoD University Research Initiative (URI) program administered by the Office of Naval Research under Grant N00014-01-1-0795.

Data that flow through memory typically go through four basic steps. First, an operate instruction (such as an ADD) produces the data value. Next, a store instruction places the data value into a memory location. The reason for storing the value may be for passing arguments to a subroutine, or it may be a spill due to high register pressure. Later, before the data value is needed again, a load instruction retrieves the value from memory and places it in a register. Finally, some consumer instruction uses the value (perhaps as an operand to another arithmetic instruction).

The question we asked is if the processor can accurately predict a memory dependence between a load and a store, why bother with sending the data values to the memory subsystem and back? By playing some tricks with register renamings, we could potentially make all consumers of the data read the value directly from the original producer instruction. To the extent that the memory dependence predictor can accurately determine the dependences from stores to loads, this *bypassing* of memory can potentially save many cycles.

In this paper, we explore the idea of bypassing memory dependences in load-store (RISC) architectures. We evaluate a store sets based version of this bypassing technique, and discover that the performance gains are unimpressive. We also explore the performance limits of memory bypassing by simulating different types of ideal processors. Our overall conclusion is that this kind of memory bypassing results in relatively small performance gains (and would probably be quite complex to implement in hardware as well).

The rest of this paper is organized as follows. In Section 2, we briefly review how store sets work and explain how to extend this to enable memory bypassing. In Section 3, we present the results of our performance studies to quantify the benefit of memory bypassing. We explain in Section 4 why memory bypassing does not provide any substantial performance gain. In Section 5, we attempt to improve performance by increasing the opportunities for bypassing memory dependences. We briefly review some related work in Section 6, and we conclude the paper with some final remarks in Section 7.

2. The Idea

Store sets are an effective technique for memory dependence prediction. Using store sets, a processor dynamically identifies memory dependences between store and load instructions. In this section, we explain how to use this information to completely shortcut or *bypass* memory.

2.1. Store Sets

A static load instruction's *store set* is the set of all store instructions that the load has ever been dependent on. Knowing a load's store set enables accurate memory ordering by preventing the load from issuing when any earlier stores belonging to the load's store set are pending. A processor can dynamically learn the store sets for each load instruction by initially allowing all loads to speculatively issue as soon as the load's arguments are ready. When the processor detects a memory ordering violation, the corresponding store is added to the load's store set.

The store set implementation presented in [3] uses a pair of tables to predict memory dependences. The first table learns the actual store sets. The second table, the *Last Fetched Store Table* (LFST) tracks the most recent store instructions currently in the instruction window that also belong to a store set. Each load instruction checks the LFST to determine if a store instruction currently in the instruction window is in its store set. If such a store exists, then the processor creates a dependence between the two instructions, forcing the load to wait for the store. For the purposes of this paper, it is sufficient to think of the store sets memory dependence predictor as a black box that takes a load instruction as an input, and returns a hardware identifier of an earlier store that the load depends on, or returns a null value if the load may issue as soon as its arguments are ready.

2.2. Memory Bypassing

Data values that are stored to and loaded from memory typically follow a produce-store-load-consume dependence chain. Memory bypassing converts this four instruction chain into a shorter producer-consumer relation.

An example of the produce-store-load-consume dependence chain is illustrated in Figure 1a. An ADD instruction α creates a new data value and stores the result in register R7, which has been renamed to physical register P18. At some later point in the instruction stream, the compiler decides that register R7 should be used for some other purpose, and uses a store instruction σ to place the result of the ADD into memory. Some number of cycles later, the program needs the result of the ADD again, and uses a load instruction λ to place the data from memory into register R3, which is mapped to physical register P23. Finally, another instruction β uses the value as an input operand.

The pair of store and load instructions to temporarily store a data value to memory may take several cycles. Furthermore, either the load or the store may be delayed for some additional cycles due to the fact that their addresses may not be immediately known. On the other hand, if we have an accurate prediction that the load λ depends on the store σ, we know that the value loaded into register P23 is identical to the value produced by the original ADD instruction α. Whether instruction β reads its operand from physical register P23 or P18, the final result will be the same. Reading the result directly from P18 avoids the long trip to and from memory.

The bypassed produce-consume dependence chain is really a dependence graph with a side branch to verify the dependence prediction, as shown in Figure 1b. As soon as the corresponding data dependences are ready, instructions α and β may issue. Some time later, the load and store must both issue and verify that they were indeed referencing the same address and that no other store to the same address occurred between σ and λ. If the memory references were to different locations in memory, then the bypassing would have forwarded the wrong value to instruction β. At this point, some sort of recovery is necessary to squash or reexecute instruction β, and its dependent instructions. Furthermore, some repair mechanism is needed since instruction β really should be reading its argument from P23, not P18.

Intuitively, it would seem that bypassing memory should provide large performance gains. Bypassing removes the latency of going to the store forwarding buffer and back. Furthermore, memory bypassing removes the computation of addresses from the critical path. To the extent that dependence predictions are accurate, address calculations are relegated to non-critical address verifications. Although we think that a hardware mechanism for efficient recovery is possible, we will omit details of an implementation since bypassing ultimately does not gain much in performance.

3. Experiments

We experimentally quantify the performance benefits of memory bypassing. In this section, we explain our simulation methodology, our simulated processor model, and present the results of our experiments.

3.1. Methodology

Our processor simulator is based on the SimpleScalar toolset, version 3.0 for the Alpha instruction set [1]. Specifically, our simulator is derived from *sim-outorder*, a cycle-accurate out-of-order processor simulator based on the register update unit (RUU) [15] which uses a unified reorder

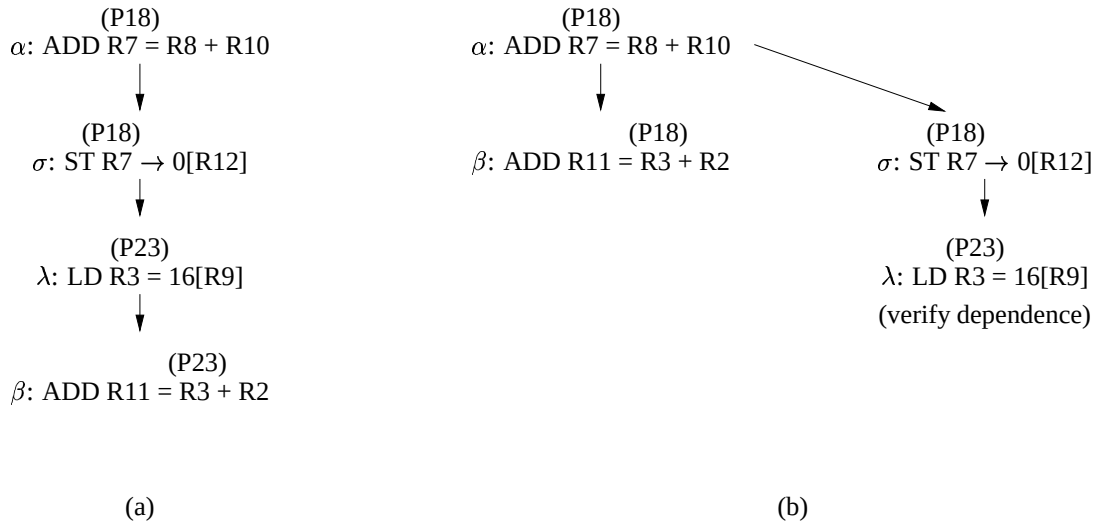

Figure 1. (a) A typical dependence chain from a producer instruction, through memory, to the consumer. (b) The dependence chain when memory is bypassed.

Decode Width	8
Issue Width	8
Commit Width	8
IFQ Size	32
RUU Size	128
LSQ Size	64
IL1 cache	8-way 256KB
DL1 cache	4-way 64KB
IL1 hit	1 cycle
DL1 hit	3 cycles
Store Forwarding	1 cycle
Unified L2	4-way 256KB
L2 hit	16 cycles
Memory Latency	60 cycles
Int/FP ALU	8
Int Mult	4

Table 1. Parameters for the simulated processor.

buffer and issue queue. We added support for simulating load speculation, store sets memory dependence prediction, and memory bypassing.

We used an aggressive processor model that is comparable to the configuration used in the original store sets study [3] and the 8-wide configuration from the Stack Value File study [7]. The processor parameters are listed in Table 1. We use a McFarling style hybrid branch predictor (gshare/PAs) to predict conditional branches [9].

The simulated benchmarks come from the SPEC2000 integer codes [16]. We use a mix of input sets from the test data set and the reduced run-length inputs from the University of Minnesota [6]. We skipped the initial start-up sections for each benchmark, and then simulated 200 million instructions. The data sets and number of skipped instructions are included in Table 2. The *mean* IPCs reported are always the harmonic mean across all benchmarks.

3.2. Bypassing With Store Sets

In general, a memory dependence predictor need only predict *when* a load may safely issue. The store sets predictor also predicts the actual dependence; that is, a store in the window is explicitly predicted as the parent of the load instruction. By tracking this dependence information, we can update the register alias table so all instructions that are data dependent on the load now read their argument directly from the store's parent.

We illustrate an example in Figure 2 using the same instructions as shown in Figure 1. In step (a), the processor renames the original data producing instruction α to write its result into physical register P18. Step (b) shows the store instruction σ that writes α's result to memory. In step (c), the store sets mechanism predicts a memory dependence from σ to the load instruction λ. The register alias table (RAT) is then updated so all consumers of the load's result (logical register R3) will read the value directly from the destination register of the original producer α. In step (d), the processor renames a consumer instruction β's arguments using the mappings in the RAT. Instruction β can now receive its argument directly from instruction α, thereby completely bypassing the memory references of σ and λ. Most of this

process is standard in out-of-order superscalar processors. To perform bypassing, the processor only needs to perform the additional work of updating the RAT when the dependence is predicted.

In a processor that performs memory bypassing, the predicted dependent store and load instructions still must execute. The store instruction must write its value to memory to enforce the proper program semantics. The load instruction must at least perform its address computation to verify that the dependence prediction was actually correct. In the case of a misspeculation, the recovery procedure can be quite involved. The simplest approach is to flush the pipeline, rollback the register alias table, and restart fetching from the mispredicted load instruction. This approach squashes many unrelated instructions that are not data-dependent on the faulting load. Selective reissue only squashes and reissues those instructions that are actually data-dependent on the misspeculated load [12]. This results in higher amounts of instruction-level parallelism, but may be complex to implement in hardware.

3.3. Experimental Results

We conducted simulation-based experiments to assess the performance gains derivable from memory bypassing. The baseline experiment is a processor that performs no memory speculation; loads may only issue when all earlier store addresses have been resolved. We then simulated a processor that speculates on load instructions using the store sets memory dependence predictor and another processor that uses store sets based memory bypassing. In both cases, we also use the set merging optimization described in [3], which we found to improve performance slightly. We simulated configurations using both squash recovery and selective reissue. With squash recovery, all instructions after a misspeculated load must reexecute, although they are kept in the window. With selective reissue, only the dependent instructions are forced to reissue.

The per-benchmark and harmonic mean IPCs are shown in Figure 3. Regardless of the misspeculation recovery model, allowing loads to speculatively issue when the store sets mechanism does not predict a memory dependence shows significant performance gains. Under the squash recovery model, the processor augmented with store sets shows a 31.0% increase in the mean IPC. Adding memory bypassing to this processor yields a total mean IPC increase of 31.7% over the non-speculating configuration. Adding the hardware complexity of memory bypassing hardly seems to be worth the effort. Even considering a selective reissue policy, the gains of bypassing with store sets (39.4%) compared to store sets alone (36.6%) are not very impressive.

One possible explanation for the poor performance of

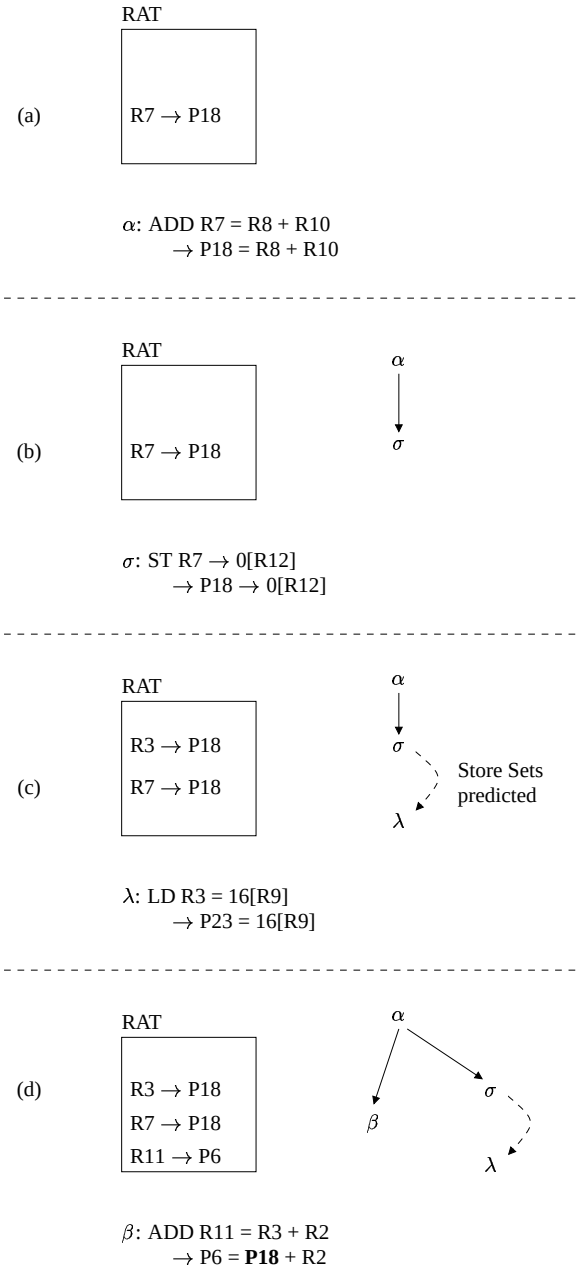

Figure 2. Using store sets to bypass memory: (a) the original value producing instruction, (b) the σ stores α's result to memory, (c) store sets predicts a dependence from σ to λ and we update the RAT so the load's children read their operand directly from the store's source, (d) the instruction β is a dependent of the bypassed load λ, but has been renamed to receive its value directly from α's destination register P18.

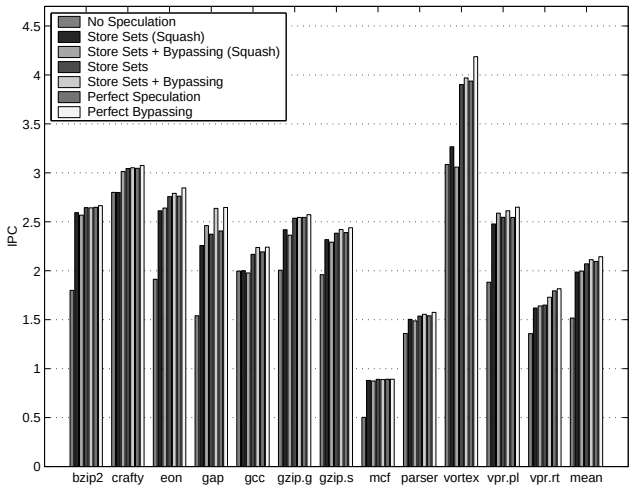

Figure 3. IPC for each benchmark and the harmonic mean IPC.

our store sets memory bypassing is that it is possible that the dependence prediction does a good job at preventing memory order violations, but doesn't necessarily predict the true dependences between stores and loads. A load may be dependent on the first of two earlier store instructions. If the store sets mechanism predicted the load to be dependent on the second store, then the load is forced to wait for the second store to issue before it can proceed. If the second store always issues after the first, then the load will never cause a memory ordering violation. On the other hand, bypassing data from the wrong store will always result in a misspeculation.

To bound how well memory bypassing could possibly perform, we simulated a processor with an oracle memory dependence predictor. These results are also shown in Figure 3. Even with perfect information about memory dependences, memory bypassing only improves the mean IPC by 2.3% over perfect dependence prediction alone. It is interesting to note that the store sets based memory bypassing (with selective reissue) achieves an IPC that is within 1.4% of perfect bypassing. This provides evidence that the store sets are accurately predicting dependent stores and loads. This is highly discouraging since the 2.3% improvement is a best case scenario. So why does memory bypassing provide such meager performance gains?

4. Explanation of Small Gains

In this section, we analyze the behavior of our benchmarks to explain why memory bypassing does not perform well. Based on the small performance increase in the ideal case of perfect memory bypassing, the natural hypotheses are (1) that there are relatively few opportunities for bypassing, and (2) the bypassed memory references are not on the critical path.

Subroutine calls present one possible situation that may provide opportunities for bypassing. Passing arguments to a function may require the caller to store the function arguments to the stack, and then subsequently the callee must load these values back into the registers. Typical register usage conventions allocate a few registers for passing function arguments, which avoids the back and forth trip to memory (under Digital Unix conventions, up to six arguments may be passed in R16 through R21 [14]). Most functions only take a few arguments.

Due to the fixed number of architecturally visible registers, the compiler may be forced to *spill* values to memory when there are no more available registers. These spills may represent more opportunities for memory bypassing. The spill is implemented by writing the contents of the spilled register to memory with a store instruction. At a later point in the program when the original value is once again needed, the value is loaded back into a (possibly different) register. If both instructions of this store-load pair are in-flight in the processor at the same time, then this presents a memory bypassing opportunity. However, assuming a reasonable register allocator, the spilled register is not likely to be used again very soon.

We would like to quantify how many possible opportunities exist for bypassing memory. To measure this, we simulated a processor configuration that performs naive speculation. That is, a load issues as soon as its address has been computed, subject to the availability of issue slots. Every time the processor detects a memory ordering violation, a store and a load to the same memory address exist in the instruction window which presents an opportunity for memory bypassing. The results are summarized in Table 2. We report the total number of load instructions, the total number of loads that resulted in a misspeculation, and the misspeculations as a percentage of the total number of loads. These misspeculations represent situations where the load is ready to issue, but it is being held up because the conflicting store is waiting for its data operand or the store has not yet computed its address. If the store is waiting for its data operand, then bypassing acts as a store value forwarding mechanism. If the store has not computed its address, then bypassing may save several cycles since the consumer of the load instruction does not have to wait for the address computation. The benchmark `gap` showed some of the largest IPC improvements, and our misspeculation counting shows that `gap` also has the largest number of bypassing opportunities. On the other hand, these statistics suggest that most of the benchmarks do not have very many opportunities for memory bypassing.

Part of the reason that there are so few bypassing op-

Benchmark	Input Set	Instructions Skipped	Number of Loads	Number of Misspeculations	Misspeculations (as % of all loads)
bzip2	test	200M	45,119,328	812,444	1.80
crafty	test	10M	62,433,982	454,349	0.73
eon	test (kajiya)	10M	60,081,228	3,588,252	5.97
gap	test	70M	62,961,510	7,292,950	11.58
gcc	test	200M	67,887,442	1,141,085	1.68
gzip.graphic	UMN	10M	48,086,571	2,529,345	5.26
gzip.source	UMN	10M	49,299,863	2,490,490	5.06
mcf	UMN	100M	58,855,822	275,950	0.47
parser	UMN	270M	57,772,158	652,342	1.13
vortex	test	10M	55,654,141	2,208,333	3.97
vpr.place	test	10M	56,082,583	2,136,052	3.81
vpr.route	test	54M	66,425,671	1,970,147	2.97

Table 2. Each load-store memory ordering violation is an opportunity for memory bypassing.

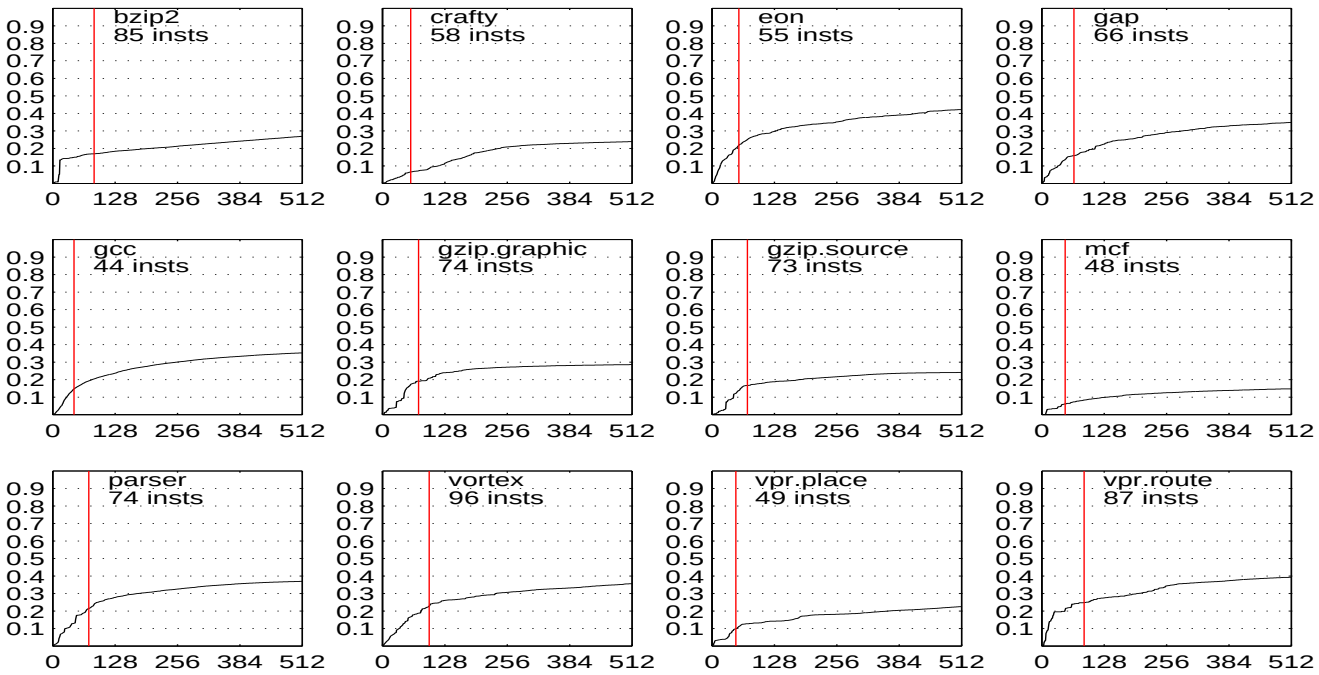

Figure 4. The cumulative frequency of the distance between a dependent store and load for each benchmark. The average instruction window occupancy is included below the benchmark name.

portunities may be due to the fact that the effective window size is too small. Store sets only consider stores and loads that are both in the instruction window at the same time. Although our instruction window has a capacity of 128 instructions, fetch stalls or bursts of high parallelism may cause the actual number of instructions in the window to be less than the maximum. To get a better idea of the characteristics of dependent stores and loads, we measured the distances between stores and loads. For every load, we count the number of instructions to the most recent store to the same address. Figure 4 shows the results of this experiment. The x-axis is the number of instructions that separate a load from the most recent store to the same address. The y-axis shows the cumulative frequency. For example, approximately 20% of all loads in `gcc` occur within 80 instructions of the previous store to the same address. The vertical line marks the average instruction window size for our processor configuration when no memory speculation is performed (not including instructions following a mispredicted branch).

The curves in our store-to-load frequency plots vary by benchmark. For some benchmarks, such as `mcf`, there are very few same address load-store pairs within the effective instruction window. Even extending to hundreds of instructions beyond the instruction window does not greatly increase the number of same address load-store pairs. The benchmark `crafty` does not exhibit very many same address load-store pairs within the effective instruction window, but the number of such pairs continues to increase as we consider more instructions. Some other benchmarks such as `parser` and `vortex` have more load-store pairs within the instruction window than the other benchmarks, but even for these benchmarks the bypassable loads comprise less than 25% of all load instructions. The benefit of bypassing these loads also depends on whether these loads are on the critical path. Effective hardware or software prefetching can potentially remove some of the loads from the critical path.

5. Enlarging the Effective Window

In the previous section, we observed that there are relatively few dependent store-load pairs within the instruction window. For some of the benchmarks, the number of dependent pairs increases if we consider a larger window of instructions. Perhaps if we enlarge the effective window for detecting dependent stores and loads, we may be able to increase the number of bypassing opportunities and boost overall performance.

One problem with using store sets is that the dependence between a store and load is broken when the store retires from the instruction window. Normally, the LFST entry corresponding to the store is replaced with a null value to indicate that no store instruction in that store set is currently in the window. An alternative approach would be to store the *value* written by that store into the LFST entry. A subsequent load instruction that accesses this LFST entry can then directly use this saved valued. This can be thought of as a simple form of load value prediction.

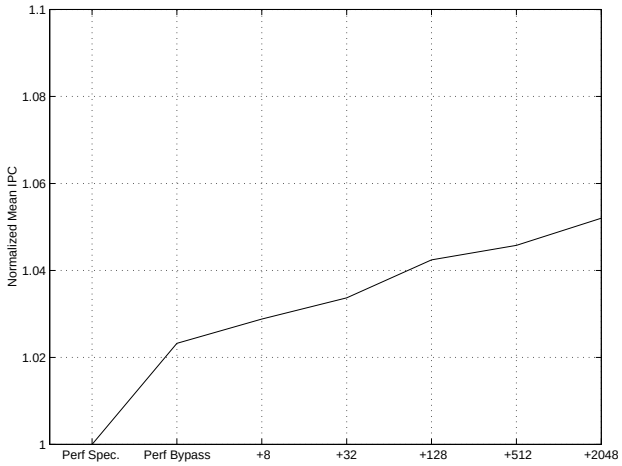

Figure 5. Relative performance of perfect bypassing augmented with a buffer of the last k retired store instructions.

We used another idealized processor model to determine the potential benefit of such an approach. We simulated a processor with perfect memory bypassing as before. Furthermore, we augment this processor with a FIFO buffer that tracks the last k stores to retire. If a dependent store-load pair exist in the window, then the processor uses memory bypassing as described in Section 3.2. If the store had already retired before a load to the same address was fetched, then the processor forwards the value of the store from the FIFO store buffer to the load. Again, the processor has oracle knowledge of the load's address.

We simulated this processor model for various sizes of the FIFO store buffer. The harmonic mean IPCs are shown in Figure 5. All IPC values are normalized with respect to the case of perfect speculation with no memory bypassing. Even in this idealized situation, the processor must track the last two thousand store instructions for the bypassing gains to achieve an additional 5% performance increase. Given that a 2K entry fully-associative buffer is probably too large to implement, and that these results rely on perfect memory dependence prediction, we conclude that any practical implementation of memory bypassing is not worth the hardware, the complexity, and the effort.

Another possibility that we considered was that the processor front-end was the bottleneck. If the fetch bandwidth was increased, perhaps there would be more opportunities

for memory bypassing. We do not believe this is the case because the results in Figure 5 effectively model a larger instruction window with respect to increasing the number of bypassing opportunities. We also simulated an unrealistic processor with a 1024-entry instruction window and very large branch predictors. This greatly increased the average number of instructions in the instruction window, but the overall impact of memory bypassing over perfect speculation was only a 3.1% increase in the harmonic mean IPC.

Out of all of the benchmarks for the 1024-entry window simulations, `vpr.route` (we will refer to this benchmark as just `vpr` for the rest of this section) exhibited the highest rate of bypassing loads: approximately one quarter of all loads were bypassed. Still, the IPC increase for `vpr` was only 3.2%. We profiled `vpr` to find out what functions comprised most of the dynamic instructions, and then we plotted the store-load dependence graphs with VCG, a graph visualization tool [13]. The graph for an invocation of the `node_to_heap` function is shown in Figure 6. We use rectangles for load instructions, diamond shapes for stores, and bypassed loads are indicated by a dependence arc from the parent store.

The section of the graph annotated with code is actually from the function `add_to_heap`, which the compiler chose to inline into `node_to_heap`. Calls to `node_to_heap` comprise 28.6% of the dynamic instructions in our sample window. The main loop of the function is from the `add_to_heap` function, which inserts a new item into a heap data structure. There are very few bypassing opportunities in the main loop because of how a heap insert works. A new node N is inserted at the bottom of a binary tree, and then "floats" upward so long as the parent node's key is greater than the new node's key. When nodes are swapped, N is written into the position of the parent node. On the following cycle, N is read out of the heap to compare to the new parent node. This read is the dependent load, shown by the one cycle bypass arcs in Figure 6 in the section of the graph for `add_to_heap`. Even if the processor can bypass all of these loads, the processor is still held up by the loads of the various parent nodes.

Instructions from the function `get_heap_head` comprise 39.9% of all dynamic instructions in `vpr` for our sample window. Figure 7 shows a section of the program where the function `try_route` (lightly shaded nodes) repeatedly calls `get_heap_head` (dark nodes)[1]. From the graph, we can see that some of the bypass paths span ten or more cycles. Unfortunately, there are also a large number of loads that we can not bypass. This suggests that even if we do bypass a load, then the critical path shifts to one of the many

[1]The reason `get_heap_head` requires so many instructions is that nodes in the heap may be invalid, and so `get_heap_head` actually has to find the node with the smallest key that is also valid, rather than simply returning the root node.

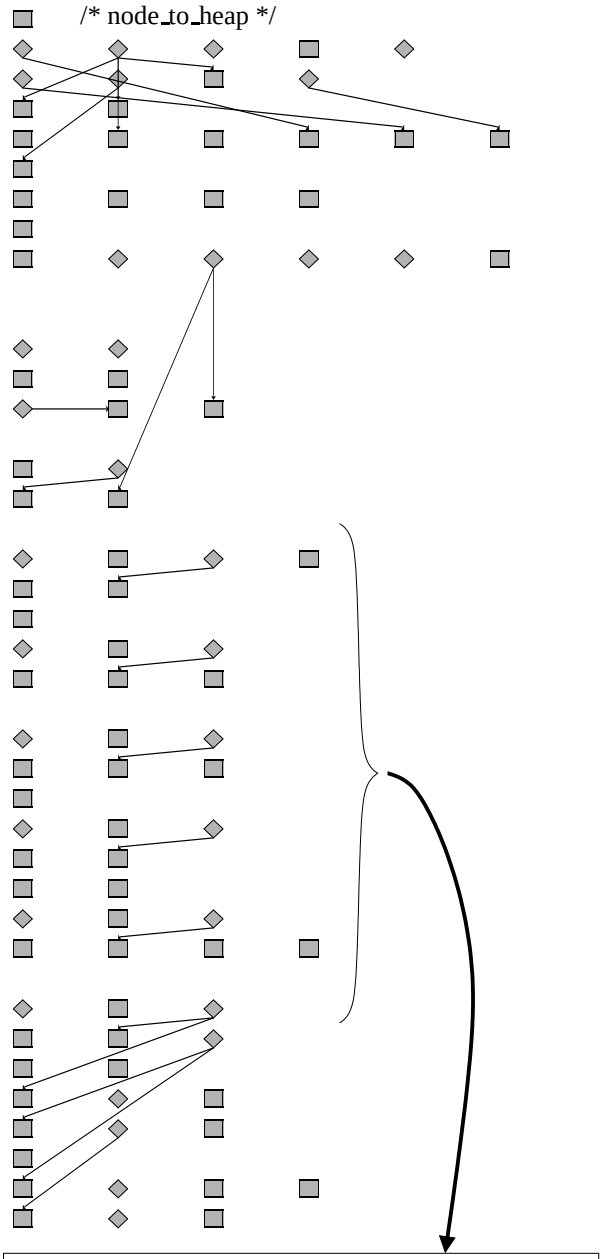

Figure 6. The bypass dependence graph of `node_to_heap`. Rectangles correspond to loads, diamonds correspond to stores, and the edges go from a store to a bypassed load.

other loads. In such a situation, bypassing may gain one or two cycles, but not the ten or more that correspond to the length of the bypass path. Another important characteristic of the bypass dependence arcs is that there are many stores that have a high fan-out. That is, a single store is the parent of multiple bypassed loads. For `vpr`, we found that on average, a store fans-out to 4.55 bypassed loads. To gain more than just a few cycles from bypassing, the processor must correctly bypass all these loads, and then the overall gain may still only be a small number of cycles due to all of the unbypassable loads previously discussed.

We also considered processor configurations where store-forwarding (not bypassing) requires multiple cycles. Our default configuration has a store-forwarding latency of a single cycle, but in a processor with a longer forwarding latency, the number of cycles saved by bypassing may increase. We simulated one processor with perfect speculation and one with perfect bypassing, and increased the store-forwarding latency to two cycles. The relative performance increase of perfect bypassing over perfect dependence prediction is only 2.4%. We further increased the forwarding latency to four cycles and increased the L1 data cache latency to five cycles, and the relative IPC improvement is still only a meager 3.8%.

6. Related Work

There are two main bodies of research related to our study. The first is in the prediction of memory dependences, and the second is in various forms of memory bypassing. Moshovos et al identified the importance of memory dependence prediction and proposed and evaluated techniques to implement dependence prediction in hardware [10]. Chrysos and Emer proposed store sets for memory dependence prediction which we used in this study [3]. The Alpha 21264 speculatively issues loads, but uses a simple table to track loads that cause misspeculations to prevent them from speculatively issuing in the future [5].

Several studies have used memory dependence to speed up loads from memory. Tyson and Austin introduced memory renaming [17], which has similarities to memory bypassing. Memory renaming forwards data to the dependents of a load by combining memory dependence prediction with load value prediction [8]. Jourdan et al explore several renaming-based techniques for move elimination, memory disambiguation and memory bypassing [4]. Moshovos and Sohi propose memory cloaking and bypassing to reduce the latency of memory instructions [11]. Calder and Reinmann compare a variety of aggressive load speculation techniques and examine the effects of combining different approaches [2]. They show that memory dependence prediction and value prediction together provide the greatest performance benefit, but further augmenting their processor

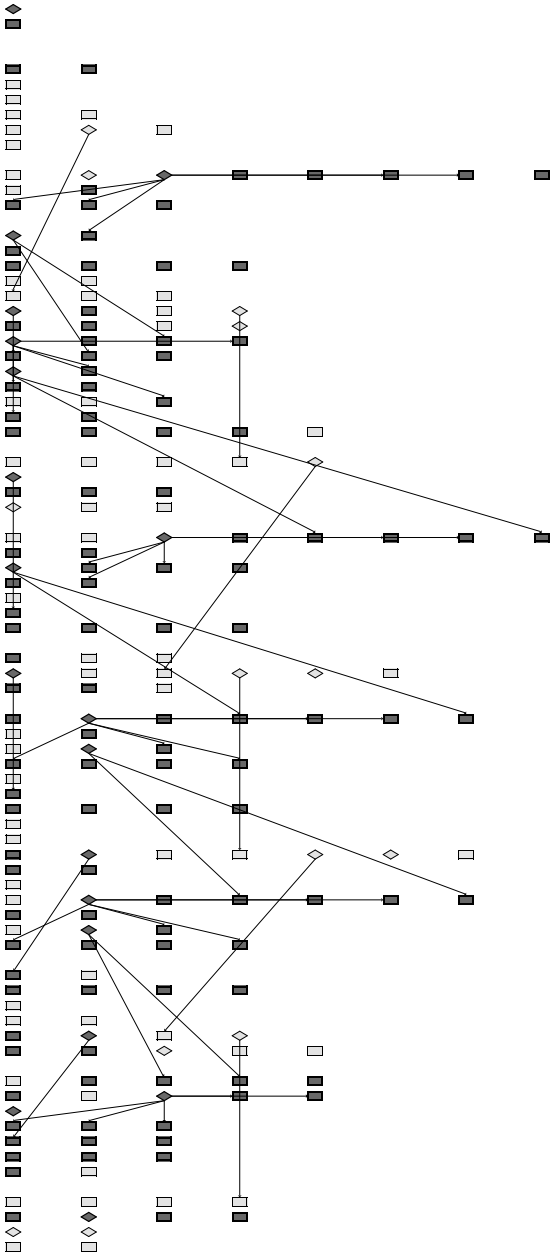

Figure 7. The bypass dependence graph of store and loads for the `get_heap_head` function (dark nodes), which is called by `try_route` (light nodes).

with memory renaming does not provide any significant additional benefit. Lee et al's Stack Value File (SVF) [7] computes the addresses of stack references early in the pipeline and redirects the stack traffic to a separate storage structure. Using register renaming, the SVF achieves effects similar to memory bypassing for stack references.

7. Conclusion

At face value, the idea of memory bypassing sounds like an excellent hardware optimization to remove long store-to-load dependence chains from the critical path. The main result of this study is that accurate memory dependence prediction can dramatically improve performance, but incorporating memory bypassing provides little additional benefit. Our analysis of memory dependence statistics suggests that this is because there are few opportunities for bypassing, and many of these opportunities are overlapping.

Our results are based on the Alpha instruction set architecture. However, an architecture with fewer logical registers, such as the x86 ISA, may yield different results. The limited number of registers leads to more frequent spilling and therefore potentially more bypassing opportunities.

References

[1] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, University of Wisconsin, June 1997.

[2] Brad Calder and Glenn Reinmann. A Comparative Survey of Load Speculation Architectures. (2), 2000.

[3] George Z. Chrysos and Joel S. Emer. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 142–153, Barcelona, Spain, June 1998.

[4] Stephan Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 216–225, Dallas, TX, USA, November 1998.

[5] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro Magazine*, 19(2):24–36, March–April 1999.

[6] A.J. KleinOsowski, John Flynn, Nancy Meares, and David J. Lilja. Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research. In *Proceedings of the International Conference on Computer Design, Workshop on Workload Characterization*, Austin, TX, USA, October 2000.

[7] Hsien-Hsin S. Lee, Mikhail Smelyanskiy, Chris J. Newburn, and Gary S. Tyson. Stack Value File: Custom Microarchitecture for the Stack. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 5–14, Monterrey, Mexico, January 2001.

[8] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value Locality and Load Value Prediction. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Cambridge, MA, USA, October 1996.

[9] Scott McFarling. Combining Branch Predictors. TN 36, Compaq Computer Corporation Western Research Laboratory, June 1993.

[10] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 181–193, Boulder, CO, USA, June 1997.

[11] Andreas Moshovos and Gurindar S. Sohi. Speculative Memory Cloaking and Bypassing. *International Journal of Parallel Programming*, October 1999.

[12] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace Processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 138–148, Research Triangle Park, NC, USA, December 1997.

[13] Georg Sander. Graph Layout Throught the VCG Tool. In *Proceedings of theDIMACS International Workshop on Graph Drawing*, pages 194–205, 1994.

[14] Richard Sites. *Alpha AXP Architecture Reference Manual*. Butterworth-Heinemann, 2 edition, October 1995.

[15] Gurindar S. Sohi. Instruction Issue Logic for High-Performance, Interruptable, Multiple Functional Unit, Pipelined Computers. *IEEE Transaction on Computers*, 39(3):349–359, March 1990.

[16] The Standard Performance Evaluation Corporation. WWW Site. `http://www.spec.org`.

[17] Gary S. Tyson and Todd M. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 218–227, Research Triangle Park, NC, USA, December 1997.