Journal of Instruction-Level Parallelism 1 (2000) 1-39 Submitted 5/99; published 5/00

A Comparative Survey of Load Speculation Architectures

Brad Calder CALDER@CS.UCSD.EDU
Glenn Reinman GREINMAN@CS.UCSD.EDU
Department of Computer Science and Engineering
University of California, San Diego

Abstract

Load latency remains a significant bottleneck in dynamically scheduled pipelined processors. Load speculation techniques have been proposed to reduce this latency. *Dependence Prediction* can be used to allow loads to be issued before all prior store addresses are known, and to predict exactly which store a load should wait upon. *Address Prediction* can be used to allow a load to bypass the calculation of its effective address and speculatively issue. *Value Prediction* can be used to bypass the load forward latency and avoid cache misses. *Memory Renaming* has been proposed to communicate stored values directly to aliased loads.

In this paper we examine in detail the interaction and performance tradeoffs of these four load speculation techniques in the presence of two mispeculation recovery architectures – reexecution and squash. We examine the performance of combining these techniques to create a load speculation chooser which provides performance improvement over using any one technique in isolation. We also examine the accuracy of these load speculation techniques for predicting data cache misses.

1. Introduction

Accurate determination of memory dependencies between store and load instructions is critical for performance on future superscalar processors. Processors with large execution windows to expose the ILP necessary to reach future generation performance goals will also expose more store/load communication and require precise load/store scheduling. In performing this scheduling, processors have to deal with aliasing between store and load instructions. One possible alternative is to require loads to wait for the completion of all previous stores before beginning execution. This avoids the problem of aliasing, but can result in many wasted cycles due to false dependencies.

Four approaches have been proposed for load speculation to reduce the impact of load instructions on processor performance – Dependence Prediction, Address Prediction, Value Prediction, and Memory Renaming. This paper examines which of these techniques are appropriate for future superscalar processors, how they interact, and how to combine them for improved performance.

Dependence prediction is used to predict aliases between load and store instructions. Dependence prediction will either predict that the load is independent of all prior stores, or it will predict which store the load is dependent upon. This allows a load to speculatively issue without waiting upon potentially independent stores after its effective address becomes available.

A load operation consists of two parts: the effective address calculation and the memory access. *Address prediction* can be used to predict the effective address calculation. This can eliminate the load's dependence on the effective address calculation and allow the load to more quickly detect store aliases. In addition, the predicted addresses can be used for data prefetching. *Value prediction* predicts the actual data that is to be brought in from memory, allowing instructions dependent on the load to speculatively execute with the predicted value.

The last form of load speculation we examine is *Memory Renaming*. Memory renaming involves predicting dependencies between loads and stores, and forwarding values directly from stores to loads using registers or a value cache, bypassing memory. Memory renaming relies on the observation that some loads typically alias the same store, even if the address or value accessed is not always the same. Moreover, these stores may not necessarily be in the current instruction window.

This paper provides a detailed analysis of the interaction between Dependence Prediction, Address Prediction, Value Prediction, and Memory Renaming. It is an extension of our prior comparison of these four load speculation architectures in [1]. In this paper we describe in more detail the implementation of each of these four architectures, related work, and provide additional results and analysis.

To examine the interaction of these four techniques we evaluate the performance of a chooser predictor, which selects between the four types of load speculation to achieve increased processor performance. We also examine the simultaneous use of multiple predictions for a load instruction, thereby predicting its address, dependence, and value at the same time.

Our baseline architecture and simulation methodology are described in Section 2. Section 3 describes using confidence counters and the misprediction recovery architectures for load speculation. Dependence prediction architectures and performance are described in Section 4. Address prediction architectures and results are described in Section 5. Value prediction architectures and results are described in Section 6. The Memory Renaming architecture and results are described in Section 7. We combine all four types of speculation and describe their performance in Section 8. Our findings are summarized in Section 10.

2. Methodology and Baseline Architecture

The simulator used in this study is derived from the SimpleScalar/Alpha 2.1 and 3.0 tool set [2], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive 16-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, branch misprediction, or load mispeculation.

To perform our evaluation, we collected results for the SPEC95 benchmarks for all the C programs and 2 FORTRAN programs. The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and FORTRAN compilers. We compiled the SPEC benchmark suite under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). Table 1 shows the data set we used in gathering results for each program, the

Program	Input	# instr fastfwd (M)	Base IPC	% ld exe	% st exe
compress	ref	0	1.93	26.7	9.5
gcc	1cp-decl	400	2.33	24.6	11.2
go	5stone21	2000	1.98	28.6	7.6
ijpeg	specmun	2000	4.90	17.7	5.8
li	ref	2000	3.48	28.2	18.0
m88ksim	ref	2000	3.96	22.1	10.9
perl	scrabbl	400	3.03	22.6	12.2
vortex	vortex	2000	4.28	26.5	13.7
su2cor	ref	2000	3.79	18.7	8.7
tomcatv	ref	2000	3.81	30.3	8.7

Table 1: Program statistics for the baseline architecture.

number of instructions fast forwarded through before starting our simulations (in millions), the baseline architecture IPC, and the percent of executed instructions that were stores or loads. We used the `-fastfwd` option in SimpleScalar/Alpha 3.0 to skip over the initial part of execution. We determined the amount of instructions to fast forward using [3]. Results are then reported for simulating each program for 100 million instructions.

2.1 Baseline Architecture

Our baseline simulation configuration models a future generation microarchitecture. We've selected the parameters to capture three underlying trends in microarchitecture design.

First, the model has an aggressive fetch stage, employing a variant of the collapsing buffer[4]. The fetch unit can deliver two basic blocks from the I-cache per fetch cycle, but no more than 8 instructions total. If future generation microarchitectures wish to exploit more ILP, they will have to employ aggressive fetch designs like this or one that is comparable, such as the trace cache [5].

Second, we've given the processor a large window of execution, by modeling large re-order buffers and load/store queues. Large windows of execution expose the ILP necessary to reach future generation performance targets; and at the same time they expose more store/load communication and thus benefit from more precise load/store scheduling. The out-of-order processor can issue 16 operations per cycle, and has a 512 entry re-order buffer with a 256 entry load/store buffer. Loads in the baseline architecture can only execute when all prior store addresses are known. To compensate for the added complexity of disambiguating loads and stores in a large execution window, we increased the store forward latency to 3 cycles.

Third, processor designs are including larger on-chip and off-chip caches. Larger caches are creating longer load latencies for hits in the L1 data cache. The Alpha 21264 processor has a 3 to 4 cycle first level data cache latency [6]. The processor we simulated has a 64K direct map instruction cache and a 128K 2-way associative data cache. Both caches have block sizes of 32 bytes. The data cache is write-back, write-allocate, and is non-blocking with four ports. The latency of the data cache is 4 cycles, and the cache is pipelined to allow up to 4 new requests each cycle. There is a unified 2nd level 1 Meg 4-way associative cache with 64 byte blocks, with a 12 cycle cache hit latency. A 2nd level cache miss has a

68 cycle miss penalty, making the round trip access to main memory 80 cycles. We model the bus latency to main memory with a 10 cycle bus occupancy per request. There is a 32 entry 8-way associative instruction TLB and a 64 entry 8-way associative data TLB, each with a 30 cycle miss penalty.

The branch predictor is a hybrid predictor with an 8-bit gshare that indexes into 16k predictors + 16k bimodal predictors [7]. There is an 8 cycle minimum mis-prediction penalty. The processor has 16 integer ALU units, 8-load/store units, 4-FP adders, 1-integer MULT/DIV, and 1-FP MULT/DIV. The latencies are: ALU 1 cycle, MULT 3 cycles, Integer DIV 12 cycles, FP Adder 2 cycles, FP Mult 4 cycles, and FP DIV 12 cycles. All functional units, except the divide units, are pipelined allowing a new instruction to initiate execution each cycle.

2.2 Issuing a Load

When executing a load or store instruction, the instruction is effectively split into two micro instructions inside the processor. One instruction calculates the effective address, and the other instruction performs the memory access once the effective address computation and any potential store alias dependencies have been resolved. In the baseline architecture, each store and load instruction must wait until its effective address calculation completes. In addition, each load and store must wait until all prior store addresses are calculated before it can issue. This is the default memory disambiguation for the baseline architecture we modeled.

In the baseline architecture, when a load *issues* it performs a lookup in the store buffer for a non-committed aliased store and it performs its data cache access in parallel. If a store alias is found, the load has a 3 cycle latency. If there is no store alias, and there is a data cache hit, the load has a 4 cycle latency because of the pipelined data cache. If there is a miss in the data cache, the miss will only be processed if no alias is found in the store buffer.

Figure 1 illustrates an example showing the dependencies used for the default memory disambiguation. This example will be used throughout the paper to show the benefits of each of the load speculation techniques. The example shows the execution of a multiply instruction, followed by two store instructions (ST2, and ST3), followed by the execution of two load instructions (LD4 and LD5), followed by a divide (DIV) and an add (ADD). The lines in Figure 1(a) represent true data dependencies between the instructions. Only the MUL, ST2, ST3, LD4, LD5, DIV and ADD are in the current instruction window. The first store, ST0, has already updated memory and passed through the processor. In this example, assume that LD4 and ST2 access the same address, so there exists a true dependency between them. Similarly, LD5 and ST0 access the same address. Figure 1(b) shows all the dependencies that the processor enforces between the different instructions. For the baseline memory disambiguation, the two load instructions must wait until the effective address of the two prior stores have been calculated before they can issue.

Table 2 shows the load latency statistics for the baseline architecture. The first column shows the percent of loads that suffer from stalls due to data cache misses. The next three columns show the percent of cycles a load spends, (1) waiting on its effective address calculation (ea), (2) waiting for prior store addresses to be calculated so the load can issue

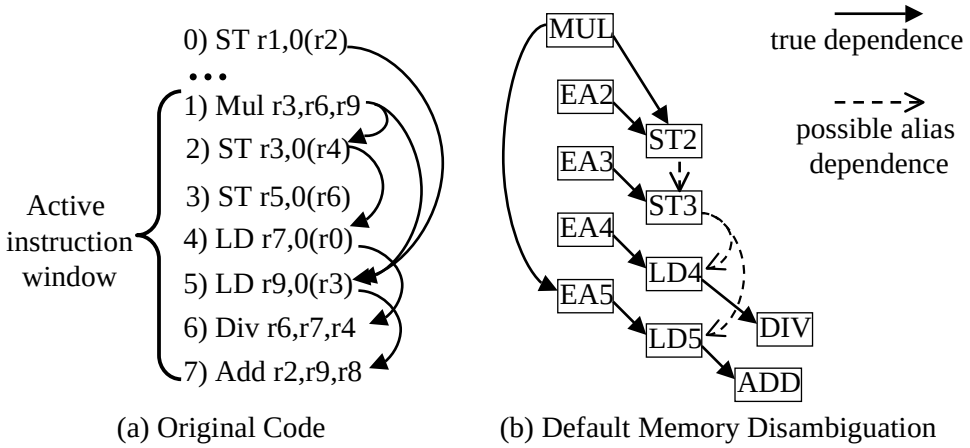

Figure 1: Code example used throughout this paper to show the benefits of load speculation. Instructions one through seven are in the processor's active instruction window. The solid arrows in (a) show the true data and alias dependencies. The arrows in (b) show the register dependencies and the dependencies enforced by the baseline memory disambiguation architecture due to potential aliases.

(dep), and (3) the latency for fetching the data (mem). The 2nd to last column shows the average number of instructions in the ROB during execution. The last column shows the percent of executed cycles the fetch unit stalled due to a lack of free ROB entries. Table 2 shows that on average for the SPEC C programs, each load spends 6.3 cycles waiting for its effective address. After the effective address is calculated, a load waits an additional 5.1 cycles on average for memory disambiguation. This is the number of cycles the load waits while all prior store addresses are calculated, ensuring the detection of any potential aliases. Finally, a load spends 4.4 cycles on average reading the loaded data value from the data cache for the C programs. The FORTRAN programs we examined waited considerably longer to read the loaded value from the data cache, around 40.5 cycles on average.

3. Speculating a Load

In this paper we examine speculatively issuing a load before all prior store addresses are known using dependence prediction, issuing the load using a predicted address using address prediction, and predicting the output value for a load using value prediction and memory renaming. For each of these techniques, information (dependence, address, or value) is predicted and later used by either the load instruction itself, or a dependent instruction, as in the case of a predicted value. Confidence prediction is used to guide when a prediction should be used. If the prediction is incorrect, misprediction recovery action must take place to correct the state of the pipeline.

Program	Dcache % Stalls	Load Delay (in cycles) ea	dep	mem	ROB occ	% cycles fetch delay
compress	10.6	15.3	11.0	4.7	190	4.0
gcc	2.0	6.7	3.9	4.1	103	1.6
go	0.6	6.1	3.1	4.1	100	0.5
ijpeg	2.9	6.1	4.6	4.8	141	2.4
li	5.8	4.5	4.3	4.0	110	0.3
m88ksim	0.1	2.1	2.3	4.1	66	0.0
perl	1.0	5.0	4.6	4.4	158	7.5
vortex	3.6	4.8	7.1	4.8	274	18.0
C average	3.3	6.3	5.1	4.4	143	4.3
su2cor	48.0	6.9	2.4	21.3	280	11.9
tomcatv	48.1	1.1	3.9	59.7	480	45.1
F average	48.0	4.0	3.1	40.5	380	28.5

Table 2: Load latency statistics for the baseline architecture. Dcache Stalls is the percent of loads that suffer from stalls due to cache misses. The next three columns show the total cycles a load spends waiting on its effective address calculation (ea), waiting for memory disambiguation (dep), and for memory access (mem). ROB shows the average number of instructions in ROB during execution. Last column shows the percent of executed cycles the fetch unit stalled because of no free ROB entries.

3.1 Producing and Consuming Predictions

Each of the four load speculation architectures we examine produce predictions which allow the load or its dependent instructions to speculatively execute. These predictions are produced early in the pipeline while the load is being fetched from the instruction cache and being decoded. When a load has a predicted dependency, address, or value, we label this load as *producing* a prediction [8]. Just because a load has a predicted dependency, address, or value, it does not mean that the prediction will be used. The prediction may be used later in the pipeline to either speculatively issue the load or its dependent instructions. When this occurs the prediction is *consumed* by the load or its dependent instructions. If the prediction is incorrect, and has been consumed then it will cause a misprediction recovery action. If the prediction was incorrect and not consumed, it will not cause any misprediction penalty. This relationship will be discussed in more detail in each of the four speculation sections.

3.2 Confidence Estimation

The address, value and rename load speculation techniques in this paper use a form of confidence estimation to decide when a prediction is a candidate for being consumed. We use confidence counters for each of these predictors. Confidence counters have been shown to be effective at reducing the miss rate of branches, while maintaining a high coverage of branch predictions [9, 10].

We use two different sets of confidence counters – a conservative one for squash recovery and a more forgiving one for reexecution recovery. There are four parts to the confidence counters. These are (1) saturation, (2) predict threshold, (3) misprediction penalty, and

(4) increment for correct prediction. We examined many different values for these four parameters, and chose two configurations – a 5-bit counter (31,30,15,1) for squash, and a 2-bit counter (3,2,1,1) for reexecution. The parameters are read as follows. For squash, a confidence counter can have a max value of 31, and the confidence indicates a prediction can be used when the counter is 30 or above. If the counter is below 30, the prediction is not used. If an incorrect prediction occurs, the saturating counter is decremented by 15. If the prediction is correct, the predictor is incremented by 1.

In modeling address, value and rename prediction, we update the predictor's values *speculatively*, and the values are repaired in the commit stage if there was an incorrect prediction. More importantly, we update the confidence counters in the write-back stage, and correct the confidence counter if the instruction is not committed. In our simulations, the prediction accuracy for some programs degraded due to the late update of the confidence counter. This is one reason for the high confidence threshold for squash recovery.

3.3 Load Mispeculation Recovery

In this paper we model squash and reexecution recovery for load mispeculation. The processor model we simulate uses a reorder buffer (ROB) and reservation stations to hold the state of the out-of-order processor.

3.3.1 Squash Recovery

When a data mispeculation occurs, *Squash* recovery flushes all the instructions out of the ROB after the mispeculated load instruction, and *refetches* the instructions from the cache starting at the next instruction after the mispeculated load. This is identical to the miss recovery approach used for branches.

3.3.2 Reexecution Architecture

A more aggressive recovery scheme would be to only *reexecute* instructions dependent upon the mispeculated load. We model reexecution recovery, which only reexecutes those instructions dependent (directly or indirectly) on the mispeculated load. This is accomplished by re-injecting the correctly loaded value onto the result bus. Instructions that had used the speculative value would detect the corrected value and be re-queued for instruction issue. This in turn may cause further reexecutions.

3.4 Load Speculation Architecture Costs

In the following sections we examine several different load speculation architectures. In gathering these results, we simulated several different hardware structure sizes and chose structures that were large enough to eliminate most of the aliasing effects. The goal of this study was to examine the interactions of these architectures without worrying about capacity misses.

In the next four sections we will go through each of the four different types of load speculation. For each predictor we will (1) describe the prior work for the predictor, (2) describe the hardware predictors we implemented for that type of speculation, and (3) examine the performance results for those predictors.

4. Dependence Prediction

Many current processors allow loads and stores to execute out-of-order by comparing the load's address to prior active store addresses. If the load address is independent, the load can issue out of order. Delaying this comparison until all prior active store addresses have completed can create long load latencies. Dependence prediction has been proposed to remove this latency by predicting if loads are independent of prior stores, or by predicting which store a load is dependent upon.

There are several varieties of dependence prediction. Kourosh et al. [11] proposed blindly speculating loads (always predicting that there are no store aliases), applying this prediction in the presence of memory consistency. They showed that the common case is for a load to be independent of prior stores.

Independence prediction can be used to predict that a load address does not depend upon any prior active store addresses. When a load is independence predicted, it can perform its cache access as soon as its effective address is available. Existing architectures, such as the DEC Alpha 21264, provide a simple but very accurate form of independence prediction. The 21264 uses a *Wait Table* to record the load instructions that have been found to be dependent upon a prior store [6]. If a load is found to be dependent upon a store, then its corresponding bit in the load table indexed by the instruction PC is set. When the load executes, if the bit is not set then the load will speculatively issue, otherwise it will wait for all prior store addresses to complete before issuing.

The second type of dependence prediction predicts the exact store (if one exists) that the load is dependent upon [12, 13]. Chrysos and Emer introduced Store Sets [14] which dynamically clusters loads and stores which have aliased the same memory addresses in the past. Their implementation allows multiple loads and stores to be clustered together to guide prediction. They avoid memory order violations by enforcing in-order issue of loads and stores within the same store set.

Figure 2 shows the benefit of dependence prediction on our prior code example described in section 2.2. Dependence prediction can correctly identify that LD4 is aliased with ST2, and that no prior uncommitted store is aliased with LD5. LD4 can issue once its effective address has completed and ST2 has issued. LD5 can issue as soon as its effective address is calculated.

4.1 Dependence Prediction Architectures

In the architecture we modeled, when a load is predicted to be independent of all prior stores, the load will issue as soon as its effective address has been calculated.

When a load is predicted to be dependent upon a particular store, the load will issue as soon as both the store issues and the load's effective address is calculated. A store issues only after its effective address and input operand are calculated. When a load is predicted in this manner, the value of the aliasing store is not directly communicated to the load, but instead, the load goes through its usual pipeline steps - checking the store buffer for an alias while looking up its address in the data cache in parallel (as described in Section 2.2).

Not every dependence prediction will be used. The dependence prediction is performed early in the pipeline during the fetch and decode stage. By the time the effective address for the load has been calculated, all prior store addresses might be known. In this case, a

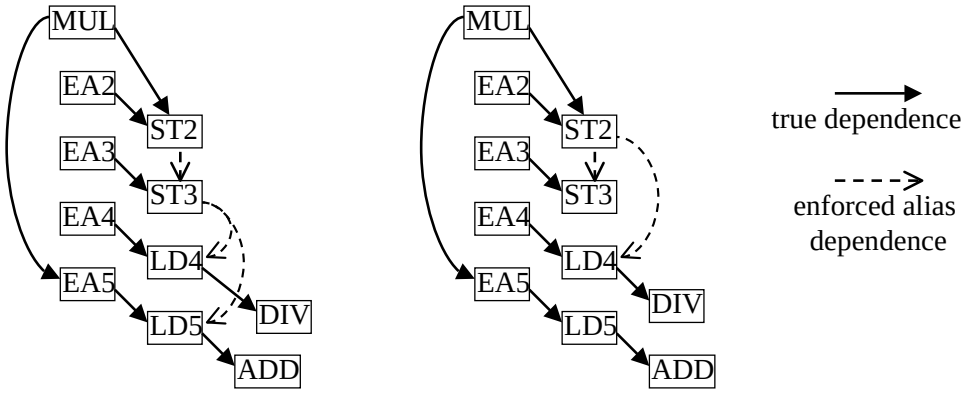

Figure 2: The benefit from using dependence prediction. Dependence prediction is shown in (b) to correctly predict the alias dependence between ST2 and LD4, and to predict that LD5 has no store alias.

dependence prediction for the load will not be used, and no misprediction penalty would occur if the prediction was wrong.

A dependence misprediction occurs when a prior store address is found to be the most recent alias of a load, after the load has issued. We modeled an aggressive miss-handling architecture to handle dependence mispredictions. Each time a store address completes, all the issued loads that occur after the store in the instruction window have their addresses checked for an alias. If an alias is found, misprediction recovery action is taken for the load, and the load re-issues. This miss recovery has an advantage in that as soon as a mispredict occurs, the load speculatively re-issues, even though there might still exist some unresolved prior store addresses before the load. Note that this could cause the load to be mispredicted several times, until the load finally finds the correct store dependency. Our results showed that effects from multiple mispredictions are much smaller than the benefits.

We will now describe the different dependence prediction architectures compared in this paper.

4.1.1 Blind

Blind prediction is an aggressive form of prediction that keeps predicting independence for a load until it gets it right. After the load's effective address has been calculated, the load speculatively issues, searching the store buffer for known aliases and performing its data cache lookup. If it finds a store alias, the load uses the value to be written by the store. Later, if a prior store's address resolves and it is a more recent alias, misprediction recovery action is taken for the load and the load re-issues, predicting that this recent store is its dependency. As described earlier, this may occur several times until the load finds its correct dependency, if one exists.

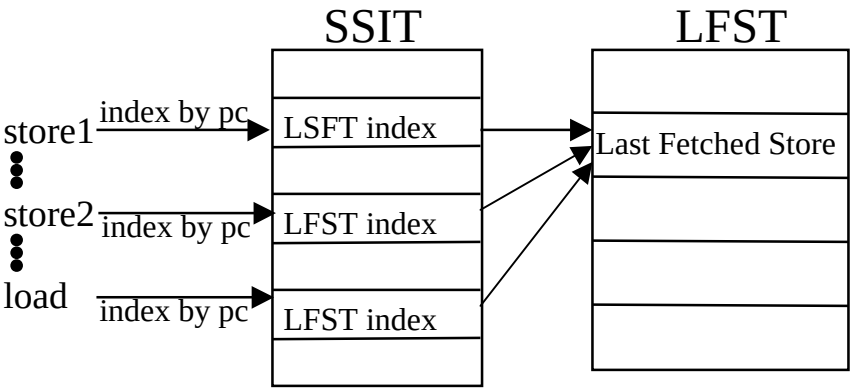

Figure 3: Structure of the Store Sets predictor. The SSIT maps memory operations to store set ids. The LFST contains the last fetched store instruction for a particular store set.

4.1.2 Wait

A *Wait* dependence predictor [6] has a table with one prediction bit for each instruction in the instruction cache. The predictor will speculatively issue a load if the wait bit associated with that load instruction is not set (turned off). The load speculatively issues as soon as its effective address is calculated. On a misprediction, the wait bit is set to avoid mispredictions in the future. If the wait bit is set, the load waits until all prior store addresses have been calculated before issuing. To prevent the predictor from being too conservative, all wait predictor bits in the instruction cache are cleared every 100,000 cycles. Moreover, on an instruction cache miss, the wait bits are cleared for the instructions in the incoming cache line.

4.1.3 Store Sets

Store sets [14] attempt to chain together memory operations that alias the same location. Memory operations are assigned store set id numbers, and those operations which are found to alias the same memory location are given common id's. The store set architecture contains two tables shown in Figure 3. A cache called the Store Set Id Table (SSIT) [14] is used to keep track of store set id numbers for each load and store instruction. When a store or load is fetched, their PC is used as an index into the SSIT and an id is returned. This id is then used to index into a table called the Last Found Store Table (LFST) [14], which tracks the last store operation to issue with that particular store set id. The LFST returns a store identifier indicating which uncommitted store instruction may be aliasing the memory address the load is referencing. If a store dependence is found, the load instruction will delay its issue until that store issues and its own effective address is calculated. If no dependence is found, the load will issue as soon as its effective address has been calculated.

If a load is speculatively issued, and is found to be mispredicted because of the existence of an unknown prior store alias, the load and store will be made members of the same store set. This occurs by storing both the store and load in the SSIT, and having them point to

	Dependence Predictor						
	Blind	Wait		Store Sets			
				Indep		Dep	
Program	% mr	% ld	% mr	% ld	% mr	% ld	% mr
compress	9.0	82.7	0.0	77.9	0.0	22.1	0.0
gcc	4.2	89.9	0.2	82.9	0.2	17.1	0.1
go	3.5	85.3	0.2	83.4	0.1	16.6	0.0
ijpeg	6.3	84.1	0.0	77.6	0.0	22.4	0.0
li	14.4	67.7	0.1	47.6	0.0	52.4	0.0
m88ksim	4.9	91.7	0.1	82.4	0.2	17.6	0.0
perl	5.2	84.1	0.0	75.7	0.0	24.3	0.0
vortex	2.2	95.6	0.0	60.2	0.0	39.8	0.0
C average	6.2	85.1	0.1	73.4	0.1	26.5	0.0
su2cor	4.8	91.9	0.0	91.9	0.0	8.1	0.0
tomcatv	1.4	98.6	0.0	98.6	0.0	1.4	0.0
F average	3.1	95.2	0.0	95.2	0.0	4.8	0.0

Table 3: Prediction statistics for dependence prediction.

the same entry (store set id) in the LSFT. See [14] for a complete description of how store set ids are allocated.

In our simulation, stores do not use dependence prediction to speculatively issue, only loads. A store must wait until all prior store addresses have been calculated before it can issue. Loads with a valid pointer to a store in the LFST entry can speculatively issue once the store has issued. If the load does not have a valid LSFT entry, then the load is predicted as being independent of all prior store addresses. We use a 4K entry direct mapped SSIT and a 256 entry direct mapped LFST. To prevent store sets from growing too large, and from establishing false dependencies, we flush the store set data structures every 1 million cycles as described in [14].

4.1.4 Perfect

A *perfect* dependence predictor is one that issues a load only after all prior aliasing stores have issued. No recovery mechanism is required, and false dependencies are avoided. This predictor exposes the maximum possible gain obtainable from dependence prediction. It relies on oracle knowledge of all prior store addresses and the current load address.

4.2 Performance of Dependence Predictors

Figures 4 and 5 show the percent speedup obtained for Blind, Wait, Store Sets, and Perfect dependence prediction over the baseline architecture for squash and reexecution recovery respectively. Results show that the Store Sets configuration achieves performance close to Perfect. It also shows that aggressive Blind speculation with reexecution can achieve performance close to Store Sets. For squash recovery, the wait bits provide a simple and efficient solution to dependence prediction with a speedup of 7% on average.

Table 3 shows the percent of loads predicted and the misprediction rate for each of the predictors. Store Sets prediction rates are broken up into the loads that are predicted as independent of prior stores, and the loads that are predicted to be dependent upon a prior store. The Wait results show the percent of loads that were predicted as being independent

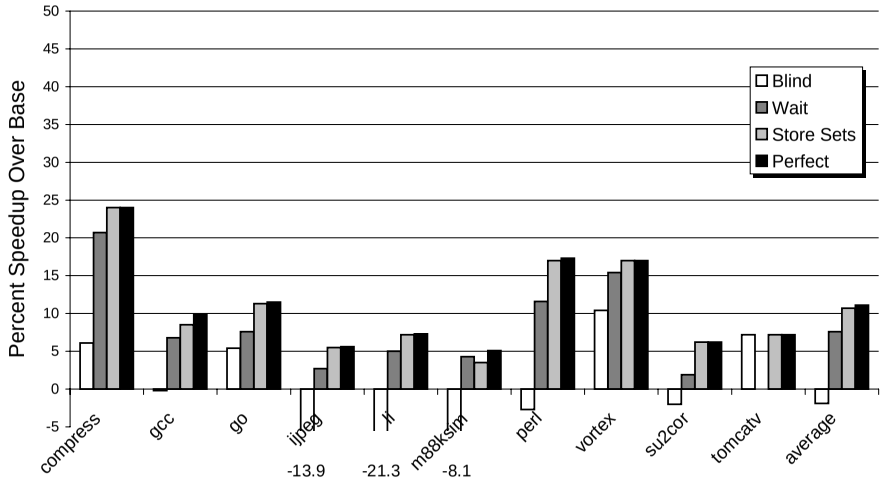

Figure 4: Percent speedup over baseline architecture for dependence prediction with squash recovery.

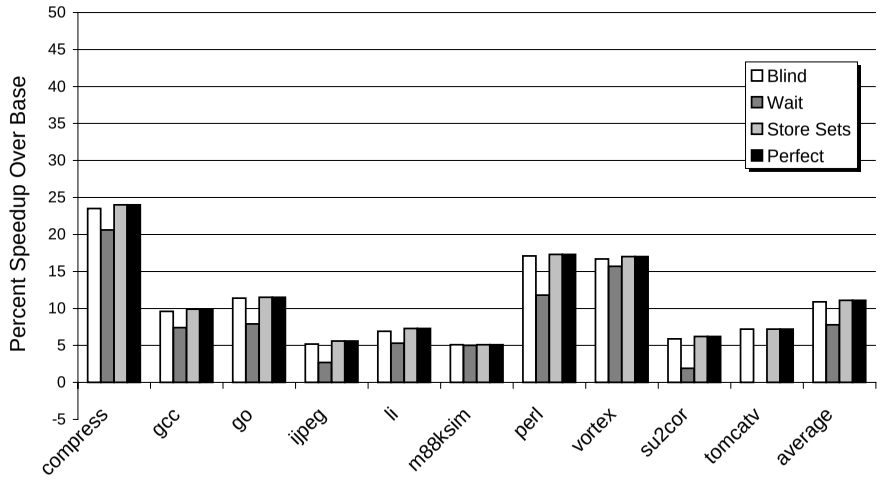

Figure 5: Percent speedup over the baseline architecture for dependence prediction with reexecution recovery.

of all prior store addresses. A misprediction occurs when a load speculatively issues without finding a prior store alias, because the store's effective address has not yet been calculated.

5. Address Prediction

All loads have to wait until their effective addresses are calculated before they can issue. If the load is on the critical path, and the address can be accurately predicted, then it can

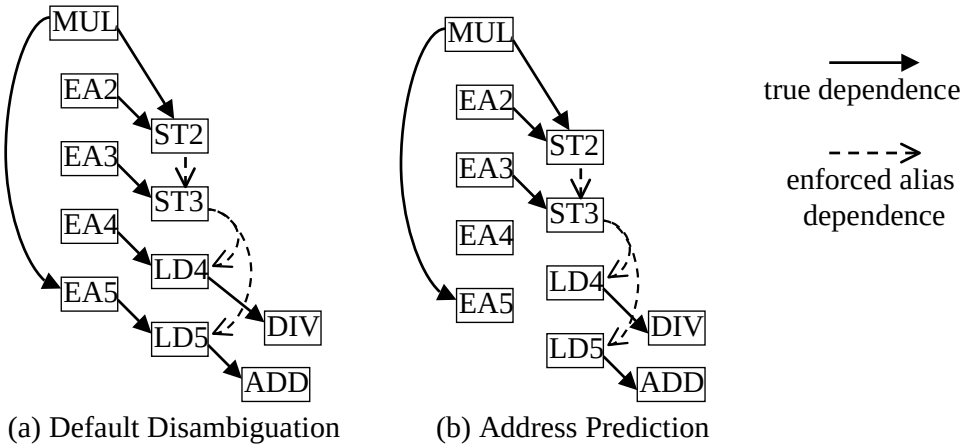

Figure 6: The benefit of address prediction. Figure (b) shows that when using address prediction LD5 does not have to wait on its effective address calculation, which is waiting on the multiply. The load only has to wait for the store addresses to be calculated before it can issue, so it can be sure that the predicted address does not have a store alias.

be beneficial to speculate the value of the address and load the data as soon as possible, or even prefetch the data.

Address prediction predicts the effective address for a load. The load then has only to wait on potential store aliases before issuing. When a load instruction uses address prediction, its effective address (EA) calculation still executes normally. Once the effective address calculation finishes, it checks this address against the predicted address to determine if the load's address was correctly predicted. If the address was incorrectly predicted, the load is re-issued with the correct address.

Figure 6 shows the benefit of address prediction on our prior code example described in section 2.2. Using address prediction can allow LD5 to speculatively issue as soon as the effective addresses are calculated for the two store instructions. This could allow LD5 to overlap with the multiply operation, which its effective address is dependent upon.

Several predictors have been proposed for address prediction, specifically to be used to reduce the latency of load instructions via prefetching [15, 16, 17]. Gonzalez and Gonzalez [18] observed that source operands of load and store operations could be predicted with considerable accuracy. They used a stride address predictor to speculatively issue both loads and stores, and to guide data prefetching.

Black et al. [19] proposed a hybrid load effective address predictor to reduce load latency. Their hybrid predictor consists of a last address predictor, a stride predictor, and a global dynamic predictor. They implemented a classification scheme which controlled both the selection of which predictor to use for a particular load, and the selection of which predictor(s) to update. This classification is influenced both by confidence counters at each predictor, and by a fixed ordering of predictors - based on spatial efficiency.

5.1 Address Prediction Techniques Examined

In our simulations, the following predictors have their values (addresses) and strides updated speculatively, and repaired in the commit stage if an incorrect update was performed. However, the confidence counters, which are used to guide when to use the prediction information, are updated in the writeback stage, once the outcome of the prediction is known.

5.1.1 Last Value (Address) Prediction

A last value predictor (*LVP*) [20] preserves the last value seen for a particular load (in this case the address of the memory reference), and speculates that the load will re-use the same memory location during its next execution. We implement LVP predictors using a direct mapped, tagged cache with 4K entries. Each entry contains the tag, the predicted value, and a confidence counter.

5.1.2 Stride

A *stride* predictor [15, 17, 21] keeps track of not only the last address referenced by a load, but also the difference between the last address of the load and the last address before that. This difference is called the stride. The predictor speculates that the new address seen by the load will be the sum of the last value seen and the stride. We chose to use the two-delta stride predictor [17, 21], which only replaces the predicted stride with a new stride if that new stride has been seen twice in a row. Our implementation uses a direct mapped, tagged cache with 4K entries. Each entry contains a tag, the predicted value, the predicted stride, the last stride seen, and a confidence counter.

5.1.3 Context

A *context* predictor [21, 22, 23] bases its prediction on the last several values seen. We chose to look at the last 4 values seen by a load. A direct mapped tagged cache of 4K entries, called the Value History Table (VHT), contains these last 4 values per entry. Another cache, called the Value Pattern Table (VPT) of size 16K entries, contains actual values (addresses in this case) to be predicted. A load's PC is used to index into the VHT, which holds the past history of the load. The 4 history values in this entry are combined to produce an index into the VPT. This entry in the VPT contains the value to be predicted. To compute the hashing function, each value is folded onto itself using an XOR to include all of its bits. The result is a value Id that is equal in size (in terms of bits) to the VPT table index. These four value Id's are then combined by shifting each value by twice its position in the value stream and XORing these values together. We did not include any PC bits into this hashing function. In [8], we showed that including too many of the PC bits in the hashing function resulted in a significant degradation in the ability to provide predictions. Not including PC bits into the hashing function allows separate static load instructions to share entries in the VPT. An example of this occurs between two different load instructions that are traversing the same pointer list. One load instruction will initialize the VPT with its values, and the other load instruction can achieve 100% value prediction accuracy when traversing that same list.

Confidence counters are also used in the VHT to guide when to use the prediction. Unlike stride predictors, context predictors do not perform well on values that have not been seen before, although there is benefit from correlated static loads as described above.

5.1.4 Hybrid

Our hybrid predictor is similar to that proposed in [23] and [19]. It is composed of one context predictor and one stride predictor, which are of the sizes described above. Prediction is guided by the confidence counters. If both predictors hit (the confidence is above their predict threshold), then the value to be speculated is chosen from the predictor with the higher confidence. If both have the same confidence, a global *mediator* counter of correct predictions is consulted. Whichever predictor has the greater history of correct predictions is declared the winner. Preference is given to stride prediction in the case of a tie. The mediator counter is cleared every 100,000 cycles. The hybrid predictor combines the ability of the context predictor to recognize repeated values without a fixed stride, and the ability of the stride predictor to predict values that have not been seen, but that are a fixed stride apart.

5.1.5 Perfect Confidence

We simulated the hybrid predictor with perfect confidence prediction. The Perfect predictor is the same as the hybrid predictor, except it only produces a prediction when the prediction is correct, and it chooses not to predict when the prediction is going to be incorrect.

5.2 Performance of Address Predictors

Figures 7 and 8 show the percent speedup obtained for Last Address Prediction, Stride Prediction, Context Prediction, Hybrid, and Perfect Confidence prediction. The results show stride prediction performs well on the Fortran programs, since stride prediction should accurately predict array traversals. In comparison, context prediction is better suited for predicting pointer addresses and provides decent speedups for the C programs.

Table 4 shows the percent of loads producing predictions above the confidence threshold and the miss rates for each of the four predictors. Results are shown using the squash confidence counters, and the last column is the percent of loads that could be accurately predicted if one had perfect confidence information. The confidence mechanism we use provides highly accurate predictions, as can be seen by the miss rates for the various predictors. However, the coverage provided is conservative as shown by the potential coverage of perfect confidence. This table corresponds to the speedups shown in Figure 7. As can be seen, context prediction provides higher coverage than stride prediction for `perl`, and this is reflected in the IPC results for this benchmark. `Vortex` has slightly less coverage from context prediction than stride prediction, but still shows more speedup with context prediction because this technique covers different loads that have more of an impact on IPC. When the hybrid predictor is used with `vortex` we can see that the combination of these two approaches captures more loads than either approach alone.

Table 5 shows the percent of executed loads that were correctly predicted by each type of predictor. Each column represents the percent of loads correctly predicted by all of the predictors listed in the column header. Each executed load for a program will

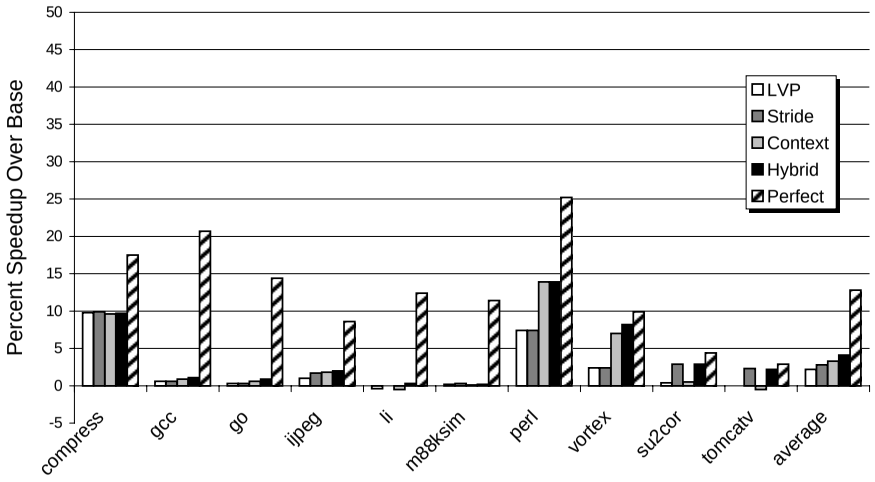

Figure 7: Percent speedup over the baseline architecture for address prediction with squash recovery.

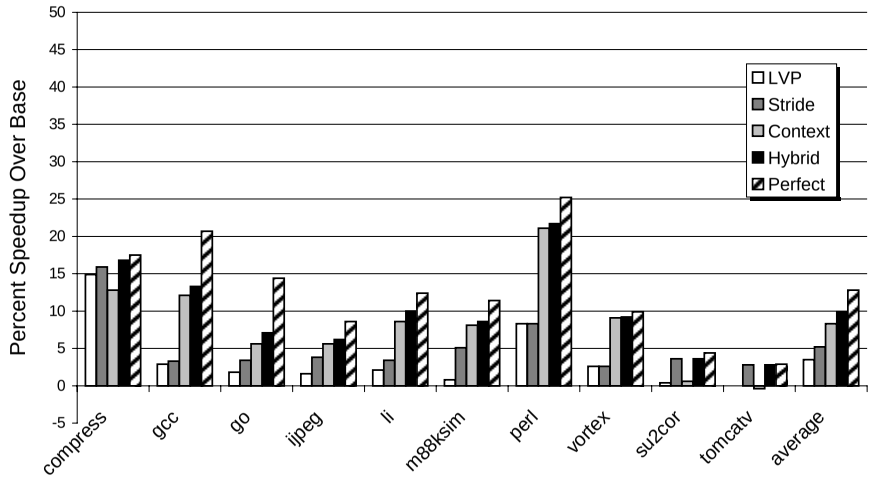

Figure 8: Percent speedup over the baseline architecture for address prediction with reexecution recovery.

only be counted in one column (all the columns for a program will sum up to 100%). The results show that on average for the C programs, 8.3% (1.8%+6.5%) of the loads are correctly predicted by stride prediction, and context prediction either chooses not to predict or incorrectly predicts them. However, 11.8% of the load addresses are correctly predicted by context, but not by stride. The stride predictor has more coverage at predicting the FORTRAN addresses however, where 53.1% of the load addresses are correctly predicted by stride prediction, and are either not predicted or incorrectly predicted by context prediction.

	Address Predictor - Using (31,30,15,1) confidence								
	Lvp		Stride		Context		Hybrid		Perf
Program	% ld	% mr	% ld	% mr	% ld	% mr	% ld	% mr	% ld
compress	71.4	0.0	71.5	0.0	72.7	0.1	73.4	0.1	85.9
gcc	16.6	0.4	17.7	0.4	15.3	0.6	19.4	0.5	62.1
go	14.2	0.2	14.6	0.3	11.9	0.8	15.8	0.4	58.7
ijpeg	17.8	0.0	20.3	0.0	39.5	0.3	41.1	0.3	78.2
li	20.8	0.1	23.0	0.1	21.7	0.4	26.3	0.2	66.7
m88ksim	26.1	0.2	26.1	0.1	34.1	0.8	41.3	0.7	79.7
perl	40.3	0.0	40.8	0.1	51.1	0.4	57.4	0.4	80.7
vortex	33.9	0.0	33.9	0.0	30.0	0.2	36.3	0.0	67.0
C average	30.1	0.1	31.0	0.1	34.5	0.5	38.9	0.3	72.4
su2cor	26.8	0.0	85.0	0.1	30.2	0.3	85.2	0.1	89.9
tomcatv	1.5	0.0	91.3	0.6	34.5	0.8	91.4	0.6	99.5
F average	14.1	0.0	88.2	0.3	32.4	0.5	88.3	0.3	94.7

Table 4: Address prediction statistics for Last Value, Stride, Context, Hybrid, and Perfect Confidence Prediction.

	Using (3,2,1,1) confidence								
Program	l	s	c	ls	lc	sc	lsc	miss	np
compress	0.0	1.6	1.3	5.1	0.0	0.3	75.7	1.0	15.0
gcc	0.3	2.0	10.6	6.5	0.1	3.0	25.6	1.0	51.2
go	0.4	2.5	4.9	8.6	0.1	1.4	28.9	0.9	52.7
ijpeg	0.4	3.3	14.1	13.6	0.2	15.1	27.9	2.0	23.9
li	0.1	3.1	12.6	5.7	0.0	0.5	28.0	1.1	49.1
m88ksim	0.0	0.7	10.1	5.7	0.0	16.4	36.9	1.6	28.5
perl	0.0	0.9	24.7	4.0	0.0	0.3	44.0	0.8	25.4
vortex	0.0	0.1	16.1	3.2	0.0	0.0	34.4	1.2	45.0
C average	0.1	1.8	11.8	6.5	0.0	4.6	37.7	1.2	36.4
su2cor	0.0	56.6	0.1	0.1	0.0	6.1	26.7	0.0	10.5
tomcatv	0.0	49.7	0.1	0.0	0.0	48.2	1.5	0.0	0.5
F average	0.0	53.1	0.1	0.1	0.0	27.1	14.1	0.0	5.5

Table 5: Breakdown of correct address predictions. L = Last Value, S = Stride, C = Context, NP = not predicted, Miss = all predictors mispredicted these loads. Each column represents the disjoint percentage of executed loads that were correctly predicted by the combination of predictors in each column header (and not by any other combination). For example, the column labelled *ls* corresponds to the percent of loads that were correctly predicted by both last value and stride prediction, but not context prediction. The *s* column represents loads that were correctly predicted by stride prediction, but not by either context or last value prediction.

Table 5 shows results using the reexecution confidence counters and corresponds to data shown in Figure 8. `Perl` and `vortex` are examples of two C programs which have higher context predictor coverage, and have a corresponding higher IPC speedup. `Su2cor` and `tomcatv` show higher stride predictor coverage, and show little improvement from context prediction. Both of these benchmarks provide speedups using stride prediction that are close to speedups attainable with perfect confidence.

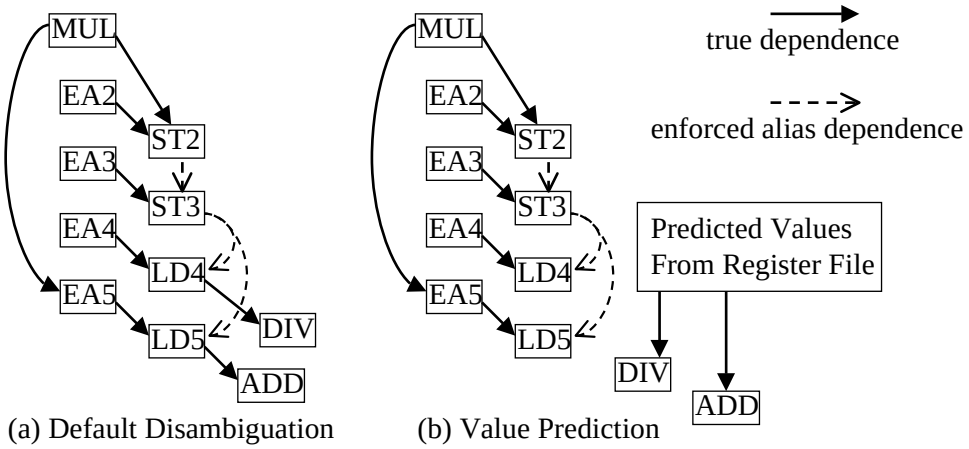

Figure 9: The benefit from using value prediction. Value predicting the result register for the two load instructions allows the divide and add to speculatively execute in parallel with the two loads.

6. Value Prediction

Value prediction predicts the actual data value that is to be brought in from memory, allowing instructions dependent on the load to speculatively execute with the predicted value. If the prediction is correct, this breaks true data dependencies since the value is produced without having to wait on the load instruction.

When a load instruction produces a value prediction, the value is inserted into the instruction's allocated physical register. This value will then be seen and consumed (used) by subsequent instructions. The predicted load instruction still takes its normal path of execution for a non-speculative load. When the predicted load's real value becomes available, it is checked against the predicted value for mispeculation.

Figure 9 shows the benefit of value prediction on our prior code example described in section 2.2. Using value prediction allows the divide and add to speculatively execute without having to wait on the load instructions to provide the values. The predictions are then checked when the load instructions complete.

6.1 Value Prediction Pipeline

In the fetch stage of the processor, the value prediction table is accessed for the range of PCs being fetched. The value table lookup could potentially take multiple cycles, and needs to complete by the time the instruction enters the register rename stage. Efficient techniques, like those proposed in [24], are needed to handle multiple value predictions, but modeling this was beyond the scope of this paper.

After a load instruction is decoded, it then enters the register renaming stage, where the instruction is allocated a physical register. The allocated physical register is normally used to hold the result value for an instruction, but we also use it to temporarily hold the predicted value for an instruction. If a value prediction was found in the table, the

predicted value is stored into the physical register, to be potentially consumed by other instructions. Once the instruction finishes executing and the real result value is available, it will overwrite any predicted value in its physical register. In this design, the register map contains an additional bit indicating whether the physical register contains a real value or a predicted value. A value prediction is only produced and stored in the register file if the prediction's confidence is above the specified threshold as described in section 3.2. Otherwise, the predicted value is discarded.

During the issue stage of the processor, instructions are issued to reservation stations when a free reservation station for a functional unit becomes available. Instructions are then executed from the reservation stations when their operands become available. When an instruction is inserted into a reservation station, the instruction reads its available inputs from the register file, storing the prediction bit from the register map along with the register value in the reservation station.

Instructions are scheduled to execute from the reservation station, when there is an idle functional unit. The input operands of an instruction in the reservation station can be in one of the following states (1) ready, (2) pending, or (3) pending with value prediction. When choosing which instruction to execute, priority is given to those instructions whose operands are ready. If no instruction in the reservation station is ready to execute and there is an idle functional unit, we try to find an instruction to schedule using a predicted value. An instruction is a candidate for execution with value predicted operands if all input operands are either ready or have predicted values.

When an instruction consumes a predicted value, this means that the instruction producing this value has not completed. We keep a *use bit* associated with the instruction producing the predicted value. In the reservation station, the consumer instruction already keeps track of the reorder buffer entry of the instruction producing the result value. When an instruction uses a predicted value, it sets the use bit for the producer of the value to indicate that an instruction has used its prediction. Once the producer finishes execution, it checks the use bit to see if its prediction has been consumed. If so, it will need to compare the predicted value to the real value loaded to see if misprediction recovery is necessary. This means that when a load is mispredicted, recovery will only be necessary if that prediction is actually consumed by another instruction.

When an instruction that has consumed a value prediction finishes executing, it stores its computed value in its physical register. The value is also broadcast to all reservation stations. This allows results from speculative values to propagate through the pipeline, potentially allowing the execution of instructions well down the dependency chain to execute in parallel with the load instruction.

6.2 Value Prediction Techniques Examined

Several architectures have been proposed for value prediction including last value prediction [20, 25], stride prediction [26, 27], context predictors [21], and hybrid approaches [23].

We simulate the five prediction architectures described in Section 5.1 for value prediction:

- *Last Value Prediction* - prediction based on last data value seen for a particular load.

	Value Predictor - Using (31,30,15,1) confidence								
	Lvp		Stride		Context		Hybrid		Perf
Program	% ld	% mr	% ld	% mr	% ld	% mr	% ld	% mr	% ld
compress	44.1	0.0	65.1	0.0	46.1	0.2	67.8	0.0	75.3
gcc	16.2	0.4	16.2	0.4	14.9	0.7	18.6	0.5	61.5
go	8.9	0.5	9.0	0.5	7.0	1.3	10.5	0.7	56.2
ijpeg	10.9	1.2	11.5	1.2	21.9	0.8	24.5	0.8	57.5
li	23.4	0.3	26.2	0.3	22.2	0.6	28.8	0.4	75.9
m88ksim	26.9	0.6	27.7	0.6	24.9	1.3	34.4	0.7	77.6
perl	45.8	0.0	48.2	0.0	46.8	0.3	57.7	0.1	78.3
vortex	38.6	0.0	38.9	0.0	33.8	0.2	43.2	0.0	70.0
C average	26.9	0.4	30.4	0.4	27.2	0.7	35.7	0.4	69.0
su2cor	44.0	0.0	44.6	0.0	46.0	0.2	49.0	0.2	53.4
tomcatv	1.5	0.0	1.5	0.0	29.6	0.4	29.7	0.4	44.2
F average	22.8	0.0	23.1	0.0	37.8	0.3	39.4	0.3	48.8

Table 6: Value prediction coverage and misprediction statistics for Last Value, Stride, Context, Hybrid Prediction, and Perfect Confidence for squash recovery.

- *Stride Prediction* - prediction based on last data value seen and current stride for a particular load.
- *Context Prediction* - prediction based on last 4 data values seen for a particular load.
- *Hybrid Prediction* - choose between context and stride predictors.
- *Perfect Confidence* - use the hybrid predictor, but only use the prediction if correct. This models having perfect confidence information when value predicting.

These predictors are identical in function and size to the ones described in the previous section on address prediction, but instead of predicting addresses, these predictors speculate on the the actual contents of the memory location.

6.3 Performance of Value Predictors

The overall potential benefit from value prediction is demonstrated by the average load delay cycles seen in Table 2. By correctly predicting the value for the load, instructions dependent upon the load can avoid stalling during the time the load spends calculating its effective address, waiting on memory disambiguation, and performing its memory access.

Figures 10 and 11 show the percent speedup obtained for Last Value Prediction, Stride Prediction, Context Prediction, Hybrid, and Perfect Confidence prediction.

We are able to achieve speedups for value prediction using squash recovery for all programs, by only consuming predictions when the prediction has a high degree of confidence, using the 5 bit counter described in Section 3.2. The results show that squash recovery is able to achieve close to a 12% speedup on average and 23% speedup for reexecution recovery. A 30% reduction in execution time could be achieved with perfect confidence. In [8, 28], history based confidence estimation was proposed to improve the confidence used by value prediction in the hope of attaining performance closer to that of perfect confidence

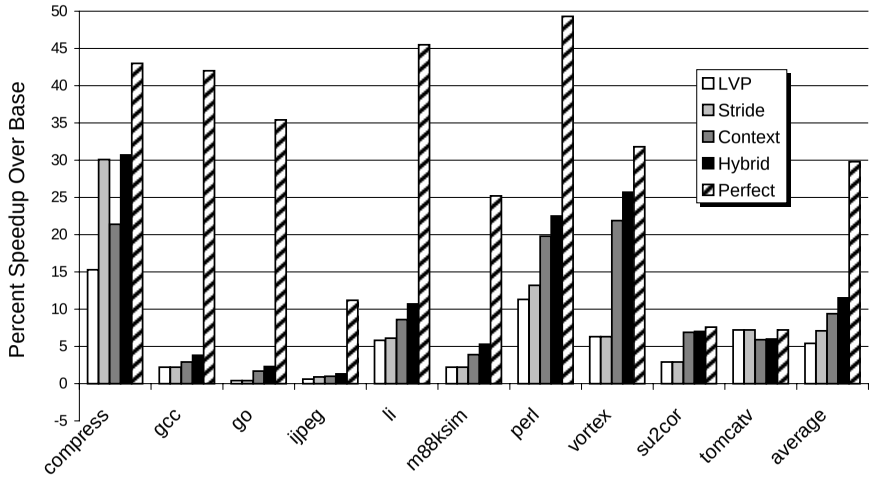

Figure 10: Percent speedup over the baseline architecture for Value Prediction with Squash recovery.

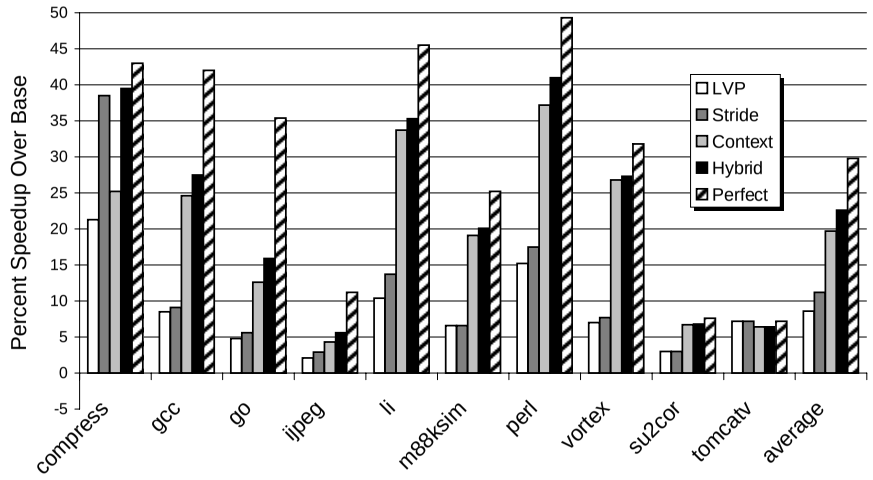

Figure 11: Percent speedup over the baseline architecture for Value Prediction with Reexecution recovery.

Using (3,2,1,1) confidence									
Program	l	s	c	ls	lc	sc	lsc	miss	np
compress	0.0	17.7	2.9	6.9	0.0	4.3	42.5	0.6	25.2
gcc	0.2	0.5	11.1	8.0	0.1	0.4	26.8	1.7	51.4
go	0.4	0.6	8.1	10.3	0.1	0.4	21.5	1.7	57.3
ijpeg	0.4	1.0	13.8	9.9	0.1	0.7	24.8	2.0	47.7
li	0.0	4.3	19.4	4.2	0.0	2.7	33.8	1.6	34.0
m88ksim	0.1	1.9	13.4	5.9	0.1	2.4	44.2	1.9	30.1
perl	0.0	2.6	15.6	9.3	0.0	0.5	41.9	0.4	29.6
vortex	0.0	0.1	13.9	6.7	0.0	0.4	37.4	0.7	40.8
C average	0.1	3.6	12.3	7.7	0.0	1.5	34.1	1.3	39.5
su2cor	0.0	1.2	5.5	2.0	0.0	0.1	42.9	0.2	48.1
tomcatv	0.0	0.0	40.9	0.1	0.0	0.0	1.5	0.1	57.5
F average	0.0	0.6	23.2	1.0	0.0	0.0	22.2	0.1	52.8

Table 7: Breakdown of correct value predictions using reexecution recovery. L = Last Value, S = Stride, C = Context, NP = not predicted, Miss = all predictors mispredicted these loads. Each column represents the disjoint percentage of loads that were correctly predicted by the combination of predictors in each column header (and not by any other combination). For example, the column labelled *ls* corresponds to the percent of loads that were correctly predicted by both last value and stride prediction, but not context prediction. The *s* column represents loads that were correctly predicted by stride prediction, but not by either context or last value prediction.

	% DL1 misses correctly predicted by value prediction								
	Squash				Reexecute				
Program	lvp	str	ctx	hyb	lvp	str	ctx	hyb	perf
compress	0.1	0.1	0.0	0.1	0.9	0.9	1.0	1.9	5.0
gcc	6.8	7.0	7.3	10.1	21.0	21.8	29.5	34.8	50.7
go	2.8	2.8	1.6	3.3	18.0	18.2	14.3	22.8	38.9
ijpeg	0.1	0.1	1.4	1.5	48.3	48.5	13.7	37.0	62.6
li	36.5	36.5	23.3	43.0	43.7	46.3	58.2	61.6	76.3
m88ksim	0.7	0.7	0.6	1.0	8.0	23.7	5.4	24.8	46.6
perl	1.4	1.4	1.4	1.9	8.1	12.1	5.9	11.9	27.2
vortex	20.7	20.7	22.2	33.4	21.3	21.0	23.1	34.5	39.6
C average	8.6	8.7	7.2	11.8	21.2	24.1	18.9	28.7	43.4
su2cor	52.6	54.1	52.0	54.5	53.3	55.9	52.4	55.7	57.1
tomcatv	0.0	0.0	13.5	13.5	0.2	0.2	16.3	16.4	19.9
F average	26.3	27.1	32.7	34.0	26.8	28.1	34.4	36.0	38.5

Table 8: Percent of time that a predictor successfully predicts a load that is stalled by a dl1 cache miss.

prediction. The need for improved confidence estimation is especially visible in Figure 10, where we trade a significant amount of coverage for prediction accuracy.

Table 6 shows the percent of loads producing predicted values above the confidence threshold and the miss rates for each of the four value predictors for the squash recovery architecture, as well as the percent of loads correctly predicted for perfect confidence prediction. The results show that the hybrid predictor both increases the load coverage over either stride or context prediction alone and decreases the miss rate. The data in this table corresponds to speedups shown in Figure 10. As with address prediction, load coverage and IPC speedup must be examined together to reveal the efficacy of a predictor. Although context prediction and stride prediction share similar coverage for `vortex` and `perl`, it can be seen that context prediction has more of an impact on IPC for these two benchmarks. However, the hybrid predictor is able to provide more IPC speedup than either technique alone. The loads covered by context prediction are more often found on the critical path, and so impact IPC more than the loads covered by stride prediction.

Table 7 shows the percent of executed loads that were correctly predicted by each type of predictor. Each column represents the percent of loads correctly predicted by the predictors listed in the column header. The results show that 10.3% of loads are covered by stride prediction and not by context for the SPEC C programs, but slightly more (12.3%) are only covered by context prediction. This effect is even more dramatic for the FORTRAN programs we examined where 23.2% of loads are covered only by context prediction. The data in this table corresponds to speedups shown in Figure 11. Table 7 reveals the disparity in coverage between these two techniques. `Compress` shows greater speedup with stride prediction than with context prediction in Figure 11. This can be explained by examining the coverage of these predictors as seen in Table 7. Most of the correctly predicted loads for `compress` could be covered with stride prediction, whereas only 3% of loads could only be correctly predicted by context prediction.

The percent of loads that suffer from stalls due to data cache misses is shown in Table 2. Table 8 then shows how effective value prediction is at predicting loads that suffer from data cache misses. The table shows the percent of first level data cache misses that were correctly predicted by the different value predictors. The cache architecture we model is 128K, 2-way associative, with 64 byte lines. The results show that a large number of cache misses can be correctly predicted by value prediction. Even last value prediction accurately predicts 8% to 22% of the data cache misses on average for the SPEC C programs.

7. Memory Renaming

Memory renaming is the process of finding dependencies between store and load instructions, and communicating a predicted value from the store to the load. Research by Moshovos et. al. [29] and Tyson and Austin [30] found that memory communication between store and load instructions can be accurately predicted in hardware.

Memory renaming keeps track of store/load dependencies in order to directly communicate a predicted value from a store to a load, bypassing memory. The approach uses a store cache to keep track of recently executed stores. When a load is found to be aliased with a store in the cache, a relationship is recorded. When the load is executed again, if the store has been resolved, the load will value predict the source value of the store from a

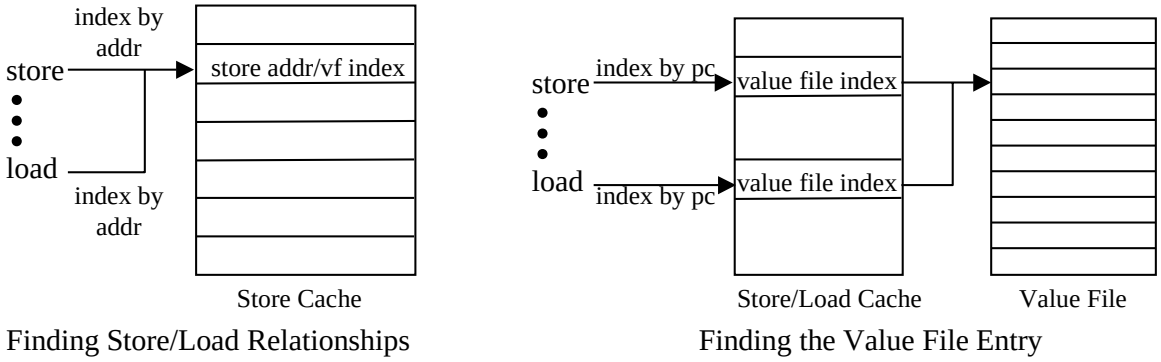

Figure 12: The structure of the memory renaming architecture [30]. The Store Cache is used to find the relationships between store and load instructions. The Store/Load cache is used to keep track of which Value File entries to use for store and load instructions. Store instructions use the value file entry to store their last value or a pointer to the instruction producing the value. Load instructions used the value file entry to predict the value to use for the load.

value file. If the store has not been resolved, the value will be forwarded to the load when it becomes ready.

Memory renaming is similar to dependence prediction, since it is used to predict dependencies as well as values. The difference is dependence prediction speculates based on the dependency, whereas memory renaming speculates based on the predicted value communicated to the load. With dependence prediction, the load still performs the store buffer and cache access to get the value for the load. A misprediction occurs when there is a store/load dependency and the load issued before that store was ready to issue. In contrast, memory renaming predicts a dependency or value, and its dependent instructions can start using the value as soon as it becomes available. A misprediction occurs only if the value is incorrect.

Another difference is that memory renaming keeps track of dependencies between store and load instructions over a larger window of instructions than store sets [14] used for dependency prediction. Memory renaming keeps track of prior store addresses in a store cache, and this cache can include stores that are no longer in the current instruction window. The store sets architecture does not need a store cache, since it is only concerned about predicting dependencies between stores and loads in the processor's active instruction window.

7.1 Memory Renaming

We chose to study the memory renaming approach of Tyson and

Austin [30] shown in Figure 12. For effective memory communication, the architecture has (1) a Store Cache to cache stores recently seen (4K entries direct mapped), (2) a Store/Load Cache to hold the dependencies found (4K entries direct mapped), (3) a Value

File for rename/value prediction (1K entries), and (4) a confidence mechanism (not shown in the figure) to determine when to use the prediction.

When a store instruction is decoded, it indexes into the Store/Load Cache with the store PC to find its Value File entry. If there is a miss, the store is allocated the least recently used Value File entry and it updates its new Store/Load Cache entry to point to this Value File entry. The store then updates the Value File entry with the current value of the store or a pointer to the instruction producing the value for the store. When the effective address for the store becomes available, the store indexes into the Store Cache with its address and updates the entry to point to its current Value File index.

When a load instruction is fetched/decoded, it uses its PC to index into the Store/Load Cache to find its Value File entry. If there is a hit, the Value File entry is then used for *predicting* the value for the load instruction. After the load's effective address is known, the load indexes into the Store Cache with its address to find an alias. If an alias is found, the load updates its Store/Load Cache entry to have the same Value File index as the aliased store. If an alias is not found, then the load updates its Store/Load entry to point to the Value File index corresponding to indexing the Value File with the load's PC. This is used to provide last value prediction the next time the load is executed. If there was no store alias, then the load updates its Value File entry with the last value used by the load to provide last value prediction. For further details, please see the complete description of the memory renaming architecture in [30].

7.2 Consuming Predicted Values

The memory renaming architecture only needs to pay the cost of misprediction if a predicted value was actually used (consumed) by a dependent instruction. Each load that hits in the store/load cache will produce a predicted value/tag, but dependent instructions on that load will only consume the predicted value once they are in the reservation station and there are idle functional units.

For a predicted load instruction, the value file provides either (1) a predicted value or (2) a physical tag pointing to the instruction producing the value. When performing memory renaming, a load producing a predicted tag from the Value File will be split into two separate instructions – spec-move and the original load. Both of these instructions will have the same register mapping and same physical register destination. The spec-move will be hooked up to the the instruction producing the value, and acts as a register move. The spec-move is only used when a load is predicting a value communicated by a store, and the instruction producing the input to the store has not yet completed execution. The spec-move is not used when the load predicts a value from the Value File. The value or physical tag for the spec-move is stored in the result register for the load instruction after the prediction is produced from the Value File.

The register file is modified to contain a speculative bit (spec bit) and a value bit. The spec bit indicates if the register file entry contains a real value or a speculative value/tag. The value bit is used to indicate if the speculative data stored in the register is either a speculative value or a physical tag to the spec-move. In addition, we modified the reservation station to contain the spec bit and value bit, to indicate the type of value stored for both input operands in the reservation station.

Consider an instruction Y, dependent upon a load Z, dispatched to a reservation station. If the load instruction has completed, then no speculation will need to occur. If the load has not completed and it has been predicted, the load destination register's spec bit will be true and instruction Y will read the speculative value or tag from the register file. If a physical tag to instruction X is read from the register file, then instruction X will be the spec-move, which will produce the speculative value for the load. In addition to this, the reservation station still holds a pointer to the original load instruction Z that is producing the real value for instruction Y for the given operand. If the spec-move X completes before load Z, then Y's reservation station will have the speculative value stored as one of its input operands and the value-bit will be changed from tag to value. If at any time the load Z finishes executing before instruction Y starts executing, then the load will update the correct operand value for instruction Y, the spec bit will be set to false, and the ready bit for that operand will be set to true.

When deciding which instruction to execute next for a functional unit, the reservation stations are first searched for instructions with ready, non-speculative operands. If no ready instructions can be found, the architecture will choose to predict instructions whose remaining operands have their value bit set to *value*. Note that in this architecture, a predicted load instruction only causes a misprediction penalty if a dependent instruction actually used the predicted value. If a dependent instruction does not use the predicted value, then there is no mispeculation penalty.

Figure 13 shows the benefit of memory renaming on our prior code example described in section 2.2. Using dependence prediction can correctly communicate the dependence from ST2 to LD4, which results in a spec-move added to the instruction stream to move the predicted value produced by the multiply into the destination register for LD4. The divide will then see this speculative value broadcast over the results bus and use it to speculatively execute while LD4 is performing its memory lookup. In addition, the value stored by ST0 is communicated via the value table to LD5 resulting in a value prediction consumed by the ADD.

7.3 Performance of Memory Renaming

Table 9 shows the speedup results for the memory renaming architecture, and memory renaming with perfect confidence prediction. The first four columns show the percent speedup for squash over the baseline IPC, the percent of loads producing predictions for squash, the misprediction rate for squash, and the percent of load data cache misses correctly predicted using memory renaming. The next two columns show the speedup for reexecution and the percent of data cache misses correctly predicted using renaming. The final three columns show the speedup, load coverage, and data cache misses correctly predicted for perfect confidence.

One interesting performance aspect we saw occur occasionally with renaming is that sometimes an incorrect store dependence is predicted for a load instruction, but the value communicated from the store to the load was correct. In this case, a misprediction will not happen, since the real result value of the load is compared to the predicted value to determine a misprediction.

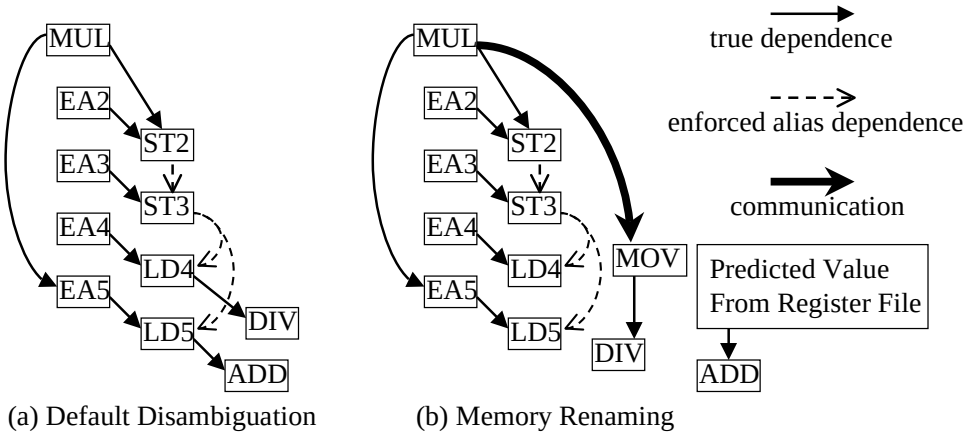

Figure 13: The benefits of memory renaming. Memory renaming can correctly communicate the result value of the multiply to the divide as a value prediction, and provide a prediction from ST0 to the add via the value table.

	Memory Renaming								
	Squash				Reexecute		Perfect		
Program	% Speedup	% lds	% MR	% DL1	% Speedup	% DL1	% Speedup	% lds	% DL1
compress	9.3	39.0	0.0	0.1	9.6	0.4	11.0	46.6	1.6
gcc	3.0	18.1	0.9	4.6	8.1	18.2	12.6	41.8	26.3
go	3.8	15.6	0.9	2.2	9.5	14.4	18.0	38.7	23.4
ijpeg	1.3	14.2	0.7	0.0	2.6	48.0	4.9	39.9	48.5
li	4.7	29.1	0.4	35.0	10.5	44.8	12.8	43.6	51.5
m88ksim	5.6	37.5	0.6	0.6	10.6	56.0	11.7	61.1	64.7
perl	13.6	41.4	0.1	0.8	15.1	17.5	20.3	53.4	27.8
vortex	9.6	34.6	0.1	28.1	10.7	29.6	14.0	46.6	31.8
C average	6.3	28.7	0.5	8.9	9.6	28.6	13.2	46.5	34.4
su2cor	5.2	45.2	0.1	51.1	4.9	51.5	5.1	46.3	52.5
tomcatv	-0.0	0.0	0.2	0.0	0.0	0.0	0.0	0.3	1.0
F average	2.6	22.6	0.1	25.6	2.5	25.8	2.6	23.3	26.7

Table 9: IPC Speedup and Prediction statistics for original rename predictor and merging rename predictor for Squash and Reexecution recovery.

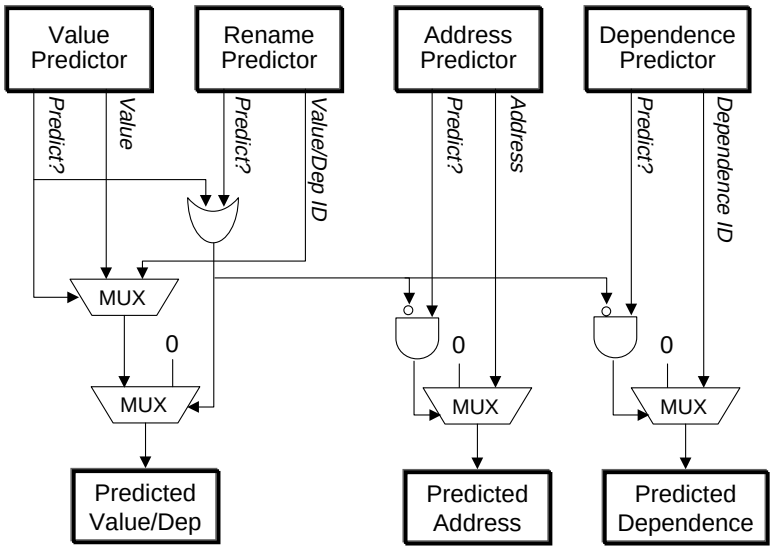

Figure 14: Structure of the Load-Spec-Chooser. The boxes across the top represent instances of the four basic types of load speculation predictors examined in this paper. The boxes along the bottom represent the predictions of the Load-Spec-Chooser.

Many of the renaming mispredictions arise from dependencies that are found to a store, but an incorrect instance of that store is communicated to the load. This can happen with loop carried dependencies as described in [30].

8. Interaction Between Load Speculators

In this section we examine the performance and interaction of combining all four types of load speculation. To combine these four predictors we implemented a number of different *choosers*. All four predictors report their confidence at predicting a particular load to the chooser. The chooser then selects which of the predictions to follow according to a set of heuristics. The chooser we found to perform the best, called the Load-Spec-Chooser, used a fixed ordering among the different predictors. In this section we examine the performance of using all combinations of these 4 predictors using the Load-Spec-Chooser.

The Load-Spec-Chooser uses the following ordering to determine which speculation to apply. Priority is given to (1) value prediction, then (2) memory renaming, and finally to (3) both dependence prediction and address prediction at the same time. We apply both address and dependence prediction together (if the predictors choose to predict), since they are used to speculate different dependencies (address and alias) for the load.

For the Load-Spec-Chooser shown in Figure 14, each predictor performs its lookup in parallel and returns a decision to predict or not. If the value predictor chooses to predict, value prediction is used for the load. If not, and the memory rename predictor chooses to predict, memory renaming is used. If neither value or rename prediction chooses to

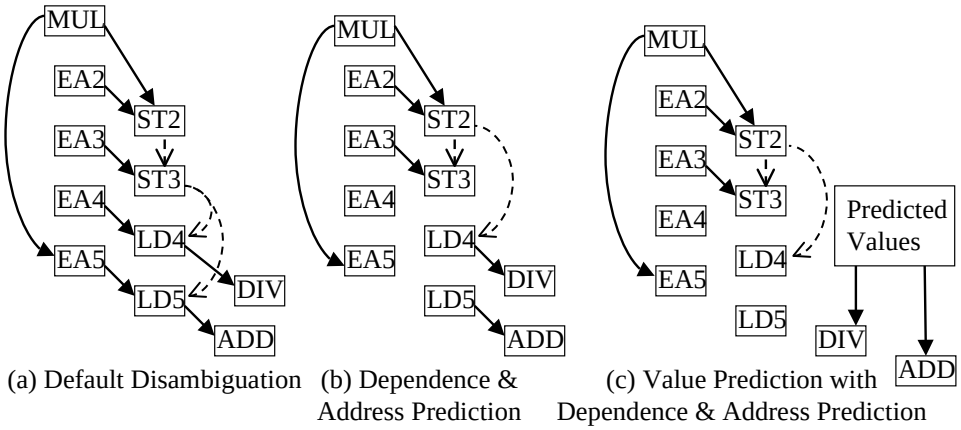

Figure 15: The advantage of check-load speculation. Part (b) shows the benefit of combining address prediction with dependence prediction. Part (c) shows the dependencies removed by performing dependence and address prediction on a load that has been value predicted.

predict, then either or both dependence and address prediction are used to speculate the load depending upon their decision to predict.

We also provide results for a 2nd type of chooser called the Check-Load-Chooser. During value and rename prediction the normal load has to go through the baseline hardware memory disambiguation before the load can issue. This can create a long misprediction penalty for load value prediction and memory renaming. We call a load that has been value or rename predicted a *check-load*. Since dependence and address prediction are very accurate, these could potentially be used to speculate the check-load to decrease the value and rename miss penalty. If the processor already has dependence and/or address prediction, no major changes would be needed to allow check-load instructions to benefit from this prediction.

Check-load speculation can be beneficial when the load dependence or address prediction is correct and the value prediction is incorrect, since it decreases the miss penalty. If the check-load prediction and the value prediction are correct, then there is no benefit and no harm caused by check-load prediction. If the check-load's address or dependency is mispredicted and the wrong value is loaded this can cause the unfortunate effect of turning a correct value or rename prediction into an incorrect prediction. Because of this, check-load prediction should only be used if dependence and/or address prediction is very accurate.

Figure 15 shows the benefit of using address prediction with dependence prediction, and using check-load speculation with value prediction of our prior code example described in section 2.2. Providing both dependence and address prediction allows LD5 to speculatively issue right away. In (c), even if a predicted value is provided for the load, dependence and address prediction can be used to process the check-load to sooner detect a mispredicted value.

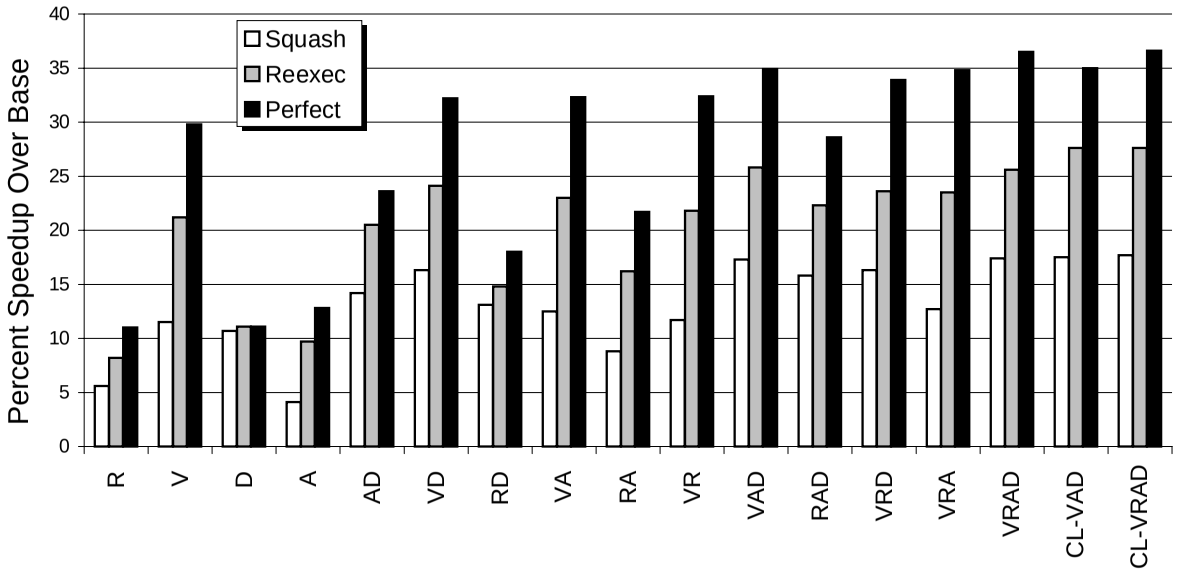

Figure 16: Average speedup results for Squash and Reexecution recovery for all combinations of the predictors using the Load-Spec-Chooser to decide which predictor to use. D = Store Set Dependence Prediction, V = Hybrid Value Prediction, A = Hybrid Address Prediction, R = Original Memory Renaming, and CL = Check-Load Prediction

8.1 Chooser Performance

To examine the interaction of the different predictors we ran all possible combinations of the predictors, using the Load-Spec-Chooser. In performing this comparison, we chose the four best predictors (store sets, hybrid address, hybrid value, and memory renaming) from the previous sections in this paper, and used the table sizes described in those sections.

Figure 16 shows the average speedup achieved for all possible different combinations of predictors for squash and reexecution recovery, as well as speedups with perfect confidence predictors. The X-axis represents the combination of predictors used. For example, the VDA bars show the speedup due to an architecture with a hybrid Value predictor, hybrid Address predictor, and a store set Dependence predictor and using the Load-Spec-Chooser to predict each load. The results show that for each type of recovery, value prediction has the best individual performance. With squash recovery, it has just slightly better performance than dependence prediction. The best performance achieved from combining two predictors is from using both store sets and value prediction, followed next in performance by the combination of store sets with address prediction.

The last two sets of bars in Figure 16 show the performance obtained using check-load (CL) prediction with value, dependence, and address prediction, and with all four predictors. The results show that check-load prediction achieves only a slight improvement in performance for reexecution recovery.

Using (3,2,1,1) confidence									
Program	d	da	vd	rd	vda	rda	rvd	rvda	other
compress	16.0	6.2	0.4	0.3	34.0	3.3	0.1	39.7	0.0
gcc	36.6	7.4	5.6	8.5	15.6	2.2	3.3	20.6	0.1
go	40.5	8.8	3.3	7.1	15.3	3.5	3.5	17.9	0.1
ijpeg	25.3	14.1	1.4	1.7	27.2	10.7	1.3	18.3	0.0
li	23.1	5.9	12.7	7.4	19.8	1.4	8.4	21.2	0.0
m88ksim	13.0	5.5	3.4	12.4	23.0	3.5	4.1	35.0	0.2
perl	14.2	5.9	1.1	8.8	29.6	2.7	4.2	33.5	0.0
vortex	21.8	4.4	5.9	14.4	26.9	2.1	5.0	19.4	0.0
C average	23.8	7.3	4.2	7.6	23.9	3.7	3.7	25.7	0.1
su2cor	9.9	35.2	0.1	0.3	8.8	3.0	0.3	42.4	0.0
tomcatv	0.7	56.9	0.1	0.0	42.3	0.0	0.0	0.0	0.0
F average	5.3	46.0	0.1	0.1	25.6	1.5	0.2	21.2	0.0

Table 10: Breakdown of correct predictions. R = Memory Renaming, D = Store Set Dependence Prediction, A = Hybrid Address Prediction, V = Hybrid Value Prediction, NP = loads that were not predicted by any of the predictors, Miss = all predictors mispredicted these loads, Other = the remaining contributions of loads for the columns not shown. Each column represents the disjoint percentage of loads that were correctly predicted by the combination of predictors in the column header. For example, the column labelled *VD* corresponds to the percent of loads that were correctly predicted by both value and dependence prediction, but not by either rename or address prediction. The "other" column contains predictor combinations which were responsible for close to 0% of the load coverage. For example, since almost all loads could be predicted with dependence prediction, the "V", "A", and "D" columns were close to 0% and were combined into the Other column, along with other combinations close to 0%.

	Data Cache Hits						Data Cache Misses					
	Rename		Value		Address		Rename		Value		Address	
Program	Cor	Inc	Cor	Inc	Cor	Inc	Cor	Inc	Cor	Inc	Cor	Inc
compress	45.2	3.5	79.1	1.8	89.7	2.7	0.4	0.3	1.4	5.1	14.5	14.6
gcc	35.2	7.2	45.2	10.8	45.5	10.4	18.3	7.8	33.4	18.2	12.6	15.0
go	30.8	6.5	38.9	9.9	42.8	7.3	13.8	10.0	20.9	14.3	9.9	12.2
ijpeg	26.5	7.7	52.9	13.8	64.8	14.6	47.4	45.0	46.5	41.2	55.5	43.1
li	43.5	5.4	61.0	11.9	53.3	11.0	44.1	5.2	55.7	13.8	62.6	7.7
m88ksim	53.8	6.2	67.2	10.6	70.4	9.6	57.2	18.4	25.9	44.9	21.8	53.1
perl	45.9	4.4	65.8	8.7	72.2	4.4	9.9	9.7	10.1	26.3	6.3	21.6
vortex	38.0	5.1	58.7	9.5	56.9	8.7	30.0	0.8	21.3	15.1	1.9	3.0
C average	39.9	5.7	58.6	9.6	62.0	8.6	27.7	12.1	26.9	22.4	23.1	21.3
su2cor	25.8	0.2	32.0	2.5	98.3	1.1	51.7	0.3	56.0	0.7	97.8	1.5
tomcatv	0.0	0.0	60.0	4.2	99.4	0.6	0.0	0.5	16.5	8.5	98.4	1.6
F average	12.9	0.1	46.0	3.3	98.9	0.9	25.9	0.4	36.3	4.6	98.1	1.6

Table 11: Percent of correct and incorrect Rename, Value, Dependence, and Address predictions for Data Cache hits and misses.

Table 10 shows the contribution of different predictor combinations. The numbers represent the percentage of executed loads that were correctly predicted by the combination of predictors in the column header. For a given program all of the columns add up to represent 100% of the executed loads. To save space we do not show show the contribution columns that were essentially all zero. Their contribution is instead accumulated in the last column (Other) in the table.

In Figure 16, after combining value prediction with dependence and address prediction, little performance improvement is seen by adding in memory renaming. One reason for this can be seen in Table 10, which shows that value prediction correctly predicts 27.7% of the loads, which memory renaming either mispredicts or chooses not to predict. In contrast, memory renaming only predicts 9.3% of the loads, which value prediction does not predict.

8.2 Interaction with Data Cache

An interesting statistic for all these techniques, that has not been reported in prior research, is the interaction of Data Cache hits and misses with load speculation. Table 11 shows the breakdown of percent correct and incorrect predictions for each type of predictor for data cache hits and misses. The best program performance would be expected for the predictors that were able to correctly predict the most loads that resulted in data cache misses. The results show that address, value and rename prediction can be very accurate for loads that are stalled due to data cache misses for some programs.

9. Space and Performance Tradeoffs

Finally, we provide space/performance analysis for the predictors we examined in our study. Figure 17 shows the average performance of the four predictors we examined for a variety of predictor sizes. The size in bytes includes the bits needed to hold both the tags and the data for the predictors.

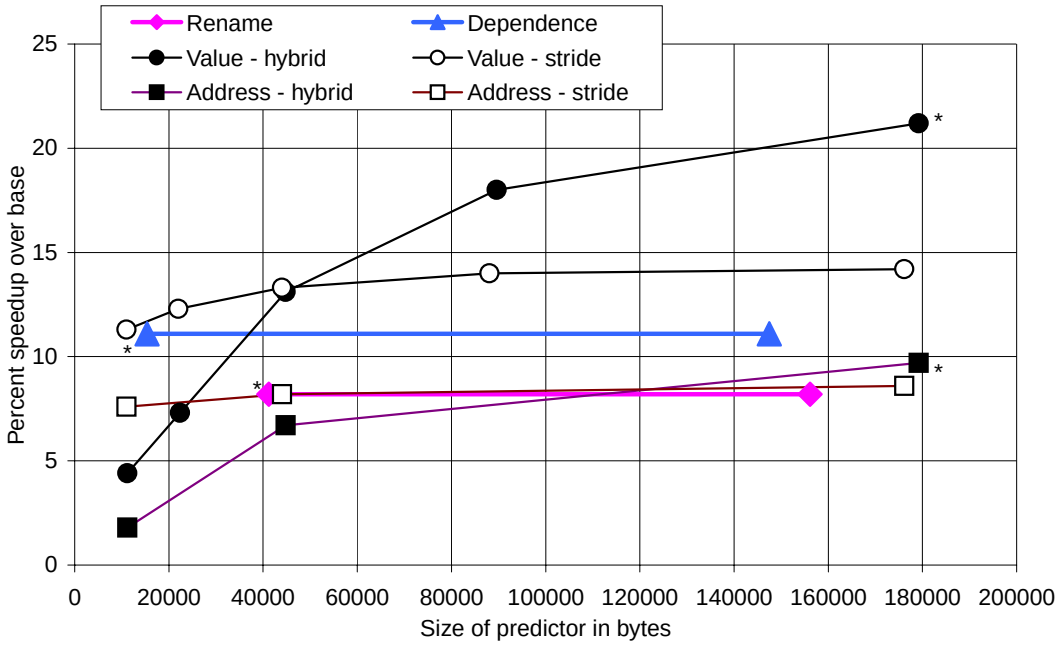

Figure 17: Analysis of the tradeoffs between performance and hardware cost for different predictor sizes. The x-axis shows the size of the various predictor structures used in bytes and the y-axis shows the average percent speedup in IPC over the base configuration for all benchmarks that we examined. Six different predictors are shown: dependence prediction using store sets, rename prediction, value prediction with a stride predictor, value prediction with a hybrid predictor, address prediction with a stride predictor, and address prediction with a hybrid predictor. Each predictor is shown for a variety of different sizes (connected with solid lines). Graph points with asterisks mark the size of the predictor used in the remainder of the study. For example, the final point on the hybrid value predictor curve is marked, as it corresponds to the predictor size used in our study.

The circular data points represent different sizes of stride (hollow circle) and hybrid (solid circle) value predictors. The first solid circle corresponds to a hybrid value predictor with a 256 entry stride value predictor and a context value predictor with a 256 entry VHT and 1K entry VPT. The first hollow circle corresponds to a 1K entry stride value predictor. These two structures are roughly the same size, but the larger stride value predictor is able to achieve better performance. However as the final set of circular points show, a 16K entry stride predictor is clearly outperformed by a hybrid predictor of roughly the same size (4K stride, 4K VHT, 16K VPT). We conclude that a small stride predictor is able to achieve reasonable performance gains, but that a hybrid predictor can ultimately outperform a stride predictor given a greater hardware budget. In this graph, this transition occurs around the third set of circular data points (a 4K entry stride predictor compared to a hybrid predictor with a 1K entry stride, a 1K entry VHT, and a 4K entry VPT).

Similarly, the square data points represent different sizes of stride (hollow square) and hybrid (solid square) address predictors. The performance of the hybrid predictor eventually exceeds that of a similarly sized stride predictor, but the disparity in performance is not as great as we saw with value prediction.

Increasing the size of the rename or dependence predictors beyond the sizes we examined in this study did not provide further improvement in IPC. However, it is interesting to note that given a small predictor storage budget, a dependence predictor performs comparably to a similarly sized stride value predictor. Moreover, a rename predictor performs similarly to a comparably sized stride address predictor.

Figure 18 shows the performance for the different predictors we examined for reexecution recovery. Here we show the impact of combining predictors with choosers. The combination of all four predictors ($VRAD$) performs a little better than just a combination of value and dependence prediction, but requires over twice as much storage space. The V, A, R, and D points on this graph correspond to the points on Figure 17 which have been marked with asterisks.

10. Summary

As execution windows continue to grow and as load latencies increase, it becomes imperative to provide aggressive load speculation to expose more instruction level parallelism. To this end, a great deal of research has been invested in devising means to disambiguate stores and loads, predict their addresses and values, and improve their communication.

In this paper, we provide a comparison of the interaction between Dependence Prediction, Address Prediction, Value Prediction, and Memory Renaming. For Address and Value prediction we examine four different types of predictors – last value prediction, stride prediction, context prediction, and a hybrid (choose between stride and context). For memory renaming we examine the approach of Tyson and Austin [30] to communicate values. For dependence prediction we examine Store Sets [14] as a technique to detect store aliasing. We also examine choosers to combine these four types of prediction to increase performance.

These prediction techniques are used to eliminate stalls associated with loads, and will be of benefit if the stalls are removed from the critical path. Address prediction can be used to reduce the average 5.9 cycle effective address latency shown in Table 2. Dependence prediction can be used to eliminate the average 4.7 cycle alias disambiguation latency. When

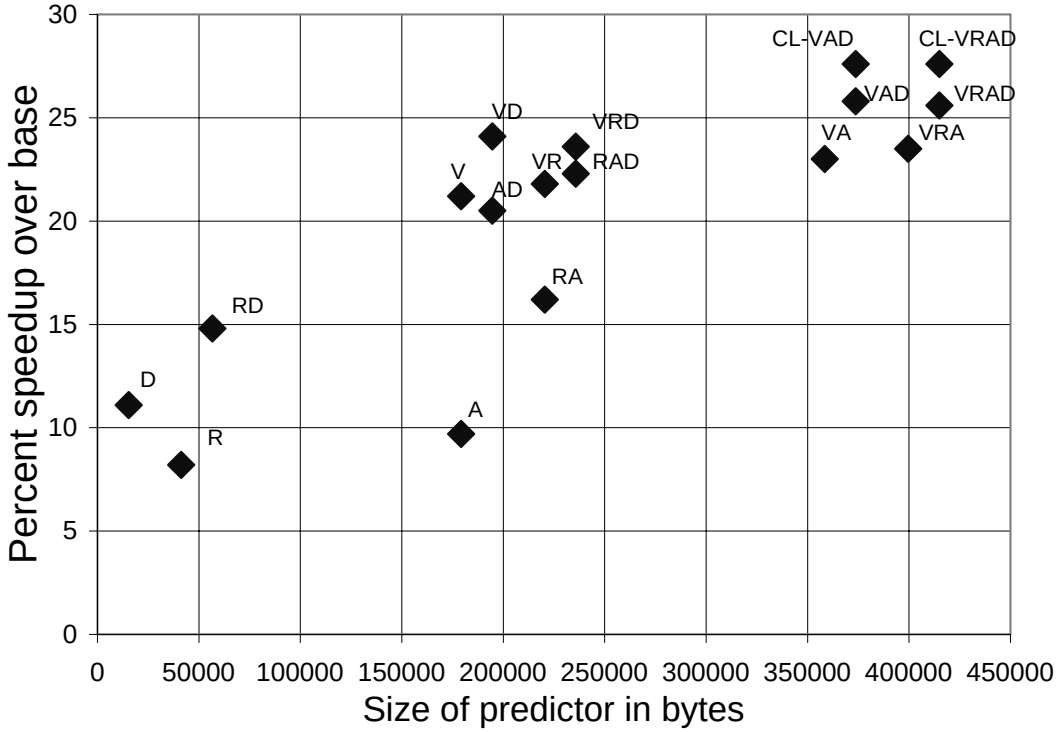

Figure 18: Analysis of the tradeoffs between performance and hardware cost for different predictor configurations. The x-axis shows the size of the various predictor structures used in bytes and the y-axis shows the percent speedup in IPC over the base configuration. Each point represents a particular load speculation configuration - R = memory renaming, D = store sets dependence prediction, A = hybrid address prediction, V = hybrid value prediction. Combinations of predictors, mediated by the Load-Spec-Chooser are also shown (i.e. VR = hybrid value prediction and rename prediction).

a load is correctly value or memory rename predicted it can eliminate the latency from the effective address, alias disambiguation, and the memory access (11.6 total cycles on average as seen in Table 2) from load's data dependency path.

In summary, we made the following observations:

- Value prediction can provide the largest performance improvement out of any one technique for both reexecution and squash recovery. Using just value prediction with reexecution, we obtained a 21% speedup. Adding store set prediction to value prediction increases speedup to 24%. Adding address prediction on top of this increases speedup to 26% on average. Combining check-load prediction with this increased the speedup to 28% on average.

- We found that 11.5% speedup can be achieved for squash recovery using value prediction, and 10.5% using dependence prediction. A high confidence counter was needed to achieve this performance. When combined, they provide a 17% speedup. Check-load prediction for squash recovery provided no performance gains, because the confidence threshold for value prediction was already very high in order to achieve speedups for squash recovery.

- Value prediction provides larger speedups than renaming. In comparing the hit rates of these two predictors, 27.7% of the time the value predictor hits and the rename predictor chooses not to predict or mispredicts. In comparison, rename correctly predicts 9.3% of the loads that the value predictor chooses not to predict or mispredicts.

- The difference in speedups for value and address prediction with squash recovery in comparison to perfect confidence prediction shows that there is still a lot of potential for improvement. These results show that improving confidence prediction and designing predictors that intelligently select which instructions to speculate can achieve significant gains. This has led to another study where we improve value prediction performance by intelligently selecting which instructions to value predict [8].

- In conducting this research, we experimented with updating the address, value, and rename predictors speculatively and during the write-back stage. We found that there is a definite performance advantage to updating the predictors speculatively rather than waiting. We also examined the effect of using an oracle to update the confidence when the prediction is made. There are performance differences for some programs between an oracle confidence update and updating the confidence once the outcome of the prediction is known. It may take several cycles after the load is fetched before the prediction is resolved and the confidence counter can be updated, and this can lead to a stale confidence counter update. Future work entails quantifying these effects, and examining techniques for speculatively updating the confidence counters.

- We used the `-fastfwd` option in SimpleScalar/Alpha 3.0, to skip past the initialization phases of the programs examined. The load speculation speedups seen when simulating just the start of the program compared to fast forwarding were very different. For example, `tomcatv` saw a 68% execution speedup using value prediction

when simulating the initial part of the program, in comparison to 5.8% speedup after fast forwarding. In contrast, `vortex` saw an 11% execution speedup with value prediction in the initial part of the program, but saw a 27% execution speedup after fast forwarding. These results show that simulation studies need to fast forward past the initialization stage of a program before gathering results. This led us to study the time varying behavior of programs to find representative samples of the programs to simulate [3].

This paper focused on the interaction of dependence prediction, address prediction, value prediction, and memory renaming using table sizes large enough (sizes are given in previous sections) to achieve good performance for each type of predictor.

Given the performance gains from dependence prediction and its small hardware needs, dependence prediction should be added to future processor designs. In addition, store sets should be included in future architecture simulation studies as the default memory disambiguation architecture. Even the Wait bit independence predictor provided a 7% speedup for squash recovery, by only associating a few bits with each cache line. Value prediction and address prediction appear to be the next speculation techniques to evaluate adding to a processor, because of their potential performance gains. Value prediction was shown to provide larger gains in performance than address, but address prediction could also be used to improve performance for data prefetching.

Acknowledgments

We would like to thank Todd Austin, Alan Eustace, Kourosh Gharachorloo, and Norm Jouppi for providing useful comments on this research. We are in debt to Todd Austin for porting SimpleScalar to the Alpha. This work was funded in part by NSF CAREER grant No. CCR-9733278, NSF grant No. CCR-9808697, and a grant from Compaq Computer Corporation.

References

[1] G. Reinman and B. Calder, "Predictive techniques for aggressive load speculation," in *31st International Symposium on Microarchitecture*, Dec. 1998.

[2] D. C. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[3] T. Sherwood and B. Calder, "The time varying behavior of programs," Tech. Rep. UCSD-CS99-630, University of California, San Diego, Aug. 1999.

[4] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *22nd Annual International Symposium on Computer Architecture*, pp. 333–344, June 1995.

[5] E. Rotenberg, S. Bennett, and J. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *29th Annual International Symposium on Microarchitecture*, December 1996.

[6] R. Kessler, E. McLellan, and D. Webb, "The alpha 21264 microprosessor architecture," in *International Conference on Computer Design*, Dec. 1998.

[7] S. McFarling, "Combining branch predictors," Tech. Rep. TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[8] B. Calder, G. Reinman, and D. Tullsen, "Selective value prediction," in *26th Annual International Symposium on Computer Architecture*, May 1999.

[9] E. Jacobsen, E. Rotenberg, and J. Smith, "Assigning confidence to conditional branch predictions," in *29th International Symposium on Microarchitecture*, Dec. 1996.

[10] D. Grunwald, A. Klauser, S. Manne, and A. Pleskun, "Confidence estimation for speculation control," in *25th Annual International Symposium on Computer Architecture*, June 1998.

[11] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *International Conference on Parallel Processing*, pp. 245–257, Aug. 1991.

[12] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *24th Annual International Symposium on Computer Architecture*, May 1997.

[13] M. Lipasti and J. Shen, "The performance potential of value and dependence prediction," in *EUROPAR-97*, Aug. 1997.

[14] G. Chrysos and J. Emer, "Memory dependence prediction using store sets," in *25th Annual International Symposium on Computer Architecture*, June 1998.

[15] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high performance processors," *IEEE Transactions on Computers*, vol. 5, pp. 609–623, May 1995.

[16] T. M. Austin and G. S. Sohi, "Zero-cycle loads: Microarchitecture support for reducing load latency," in *28th Annual International Symposium on Microarchitecture*, pp. 82–92, Dec. 1995.

[17] R. J. Eickemeyer and S. Vassiliadis, "A load instruction unit for pipelined processors.," *IBM Journal of Research and Development*, vol. 37, pp. 547–564, July 1993.

[18] J. Gonzalez and A. Gonzalez, "Speculative execution via address prediction and data prefetching," in *11th International Conference on Supercomputing*, pp. 196–203, July 1997.

[19] B. Black, B. Mueller, S. Postal, R. Rakvie, N. Utamaphethai, and J. P. Shen, "Load execution latency reduction," in *12th International Conference on Supercomputing*, June 1998.

[20] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," in *17th International Conference on Architectural Support for Programming Languages and operating Systems*, pp. 138–147, Oct. 1996.

[21] Y. Sazeides and J. E. Smith, "The predictability of data values," in *30th International Symposium on Microarchitecture*, pp. 248–258, Dec. 1997.

[22] Y. Sazeides and J. E. Smith, "Modeling program predictability," in *25th Annual International Symposium on Computer Architecture*, June 1998.

[23] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," in *30th Annual International Symposium on Microarchitecture*, Dec. 1997.

[24] F. Gabbay and A. Mendelson, "The effect of instruction fetch bandwidth on value prediction," in *25th Annual International Symposium on Computer Architecture*, 1998.

[25] M. Lipasti and J. Shen, "Exceeding the dataflow limit via value prediction," in *29th International Symposium on Microarchitecture*, Dec. 1996.

[26] F. Gabbay and A. Mendelson, "Speculative execution based on value prediction." EE Department TR 1080, Technion - Israel Institue of Technology, Nov. 1996.

[27] J. Gonzalez and A. Gonzalez, "The potential of data value speculation to boost ilp," in *12th International Conference on Supercomputing*, 1998.

[28] B. G. Z. M. Burtscher, "Prediction outcome history-based confidence estimation for load value prediction," *Journal of Instruction-Level Parallelism*, vol. 1, May 1999.

[29] A. Moshovos and G. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *30th International Symposium on Microarchitecture*, Dec. 1997.

[30] G. Tyson and T. M. Austin, "Improving the accuracy and performance of memory communication through renaming," in *30th Annual International Symposium on Microarchitecture*, pp. 218–227, Dec. 1997.