

# Predictive Techniques for Aggressive Load Speculation

Glenn Reinman      Brad Calder

Department of Computer Science and Engineering  
University of California, San Diego  
{greinman,calder}@cs.ucsd.edu

## 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 miss-speculation 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.

All of these approaches have been examined either separately or in pairs in other studies, but they have not been investigated together. This paper provides a detailed analysis of the interaction between Dependence Prediction, Address Prediction, Value Prediction, and Memory Renaming.

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. Dependence prediction architectures and performance are described in Section 3. Address prediction architectures and results are described in Section 4. Value prediction architectures and results are described in Section 5. The Memory Renaming architecture and results are described in Section 6. We combine all four types of speculation and describe their performance in Section 7. Our findings are summarize in Section 8.

## 2 Methodology and Baseline Architecture

The simulators used in this study are derived from the SimpleScalar/Alpha 2.1 and 3.0 tool set [3], 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 mis-prediction, or load mis-speculation.

To perform our evaluation, we collected results for the SPEC95 benchmarks, and due to space constraints only show results for all the C programs and 2 FORTRAN programs. The programs were

| Program  | Input    | # instr<br>exec (M) | # instr<br>fastfwd (M) | Base<br>IPC | % ld<br>exe | % st<br>exe |
|----------|----------|---------------------|------------------------|-------------|-------------|-------------|
| compress | ref      | 93                  | 0                      | 1.93        | 26.7        | 9.5         |
| gcc      | 1cp-decl | 1041                | 400                    | 2.33        | 24.6        | 11.2        |
| go       | 5stone21 | 32699               | 2000                   | 1.98        | 28.6        | 7.6         |
| jpeg     | specmun  | 34716               | 2000                   | 4.90        | 17.7        | 5.8         |
| li       | ref      | 18089               | 2000                   | 3.48        | 28.2        | 18.0        |
| m88ksim  | ref      | 76271               | 2000                   | 3.96        | 22.1        | 10.9        |
| perl     | scrabbl  | 28243               | 400                    | 3.03        | 22.6        | 12.2        |
| vortex   | vortex   | 90882               | 2000                   | 4.28        | 26.5        | 13.7        |
| su2cor   | ref      | 33928               | 2000                   | 3.79        | 18.7        | 8.7         |
| tomcatv  | ref      | 27832               | 2000                   | 3.81        | 30.3        | 8.7         |

Table 1: Program statistics for the baseline architecture.

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 -ifc`). Table 1 shows the data set we used in gathering results for each program, the number of instructions executed in the program to completion (in millions), the number of instructions fast forwarded through before starting our simulations (in millions), the baseline architecture IPC, and the percent of executed store and load instructions. We used the `-fastfwd` option in SimpleScalar/Alpha 3.0 to skip over the initial part of execution. 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 [7]. The fetch unit can deliver two basic blocks from the L-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 [22].

Second, we’ve given the processor a large window of execution, by modeling large reorder 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 [15]. 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

| Program  | Dcache<br>% Stalls | Load Delay (in cycles) |      |      | ROB<br>occ | % cycles<br>fetch delay |
|----------|--------------------|------------------------|------|------|------------|-------------------------|
|          |                    | ea                     | dep  | mem  |            |                         |
| 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                     |
| jpeg     | 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                    |
| su2cor   | 48.0               | 6.9                    | 2.4  | 21.3 | 280        | 11.9                    |
| tomcatv  | 48.1               | 1.1                    | 3.9  | 59.7 | 480        | 45.1                    |
| average  | 12.3               | 5.9                    | 4.7  | 11.6 | 190        | 9.1                     |

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 percent of 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.

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 [19]. 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 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, all stores are issued in-order with respect to prior stores, and each load must wait on the most recent store before it can speculatively issue.

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. If there is a miss in the data cache, the miss will only be processed if no alias is found in the store buffer.

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 (dep), and (3) the latency for fetching the data (mem). The 2nd to last column shows the average number of instructions in 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 each load spends 5.9 cycles waiting for its effective address. After the effective address is calculated, a load waits an additional 4.7 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 11.6 cycles on average reading the loaded data value from the store buffer of the data cache.

### 2.3 Load Miss-Speculation Recovery

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

#### 2.3.1 Squash Recovery

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

#### 2.3.2 Reexecution Architecture

A more aggressive recovery scheme would be to only *Reexecute* instructions dependent upon the mispredicted load. We model re-execution recovery, which only re-executes those instructions dependent (directly or indirectly) on the mis-speculated 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 re-executions.

### 2.4 Confidence Estimation

The address, value and rename load speculation techniques in this paper use a form of confidence estimation to decide when to speculate a load. 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 [14, 13].

We use two different sets of confidence counters – a conservative one for squash recovery and a more forgiving one for re-execution 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 re-execution. The parameters are read as follows. For squash, a confidence counter can have a max value of 31, and the confidence indicates prediction when the counter is 30 or above. If the counter is below 30, the predictor 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, this late update of the confidence counter, had a reasonable effect on the prediction accuracy for some programs. This is one reason for the high confidence threshold for squash recovery.

### 2.5 Load Speculation Architecture Costs

In the following sections we examine several different load speculation architectures. In gathering these results, we simulated several different sizes of the hardware structures and chose structures that were large enough to eliminate most of the aliasing effects, but not overly large. The goal of this study was to examine the interactions of these architectures performing at their best. Most of the architectures are direct mapped, since the benefit from load speculation will be achieved based on a fast lookup for the prediction information.

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.

## 3 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. [10] 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 [15]. If a load is found to be dependent upon a store, then its corresponding bit in the load table indexed by the instruction address 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 [20, 18]. Chrysos and Emer introduced Store Sets [6] 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.

### 3.1 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 in parallel with looking up its address in the data cache as described in Section 2.2.

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.

### 3.1.1 Blind

Blind prediction is an aggressive form of prediction that keeps predicting independence for a load until the load 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.

### 3.1.2 Wait

A *Wait* dependence predictor [15] 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 clear (turned off). The load speculatively issues as soon as its effective address is calculated. If the wait bit is set, the load waits until all prior store addresses have been calculated before issuing. On a misprediction, the wait bit is set to avoid mispredictions in the future. 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.

### 3.1.3 Store Sets

*Store sets* [6] 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. A cache called the SSIT [6] 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 LFST [6]) which tracks the last store operation to issue with that particular store set id. The LFST returns a store identifier indicating which store instruction may be aliasing the memory address the load is referencing. In our simulation, stores do not issue out of order, but loads can speculatively issue once the store contained in their LFST entry has issued. We use a 4K entry direct mapped SSIT and a 256 entry direct mapped

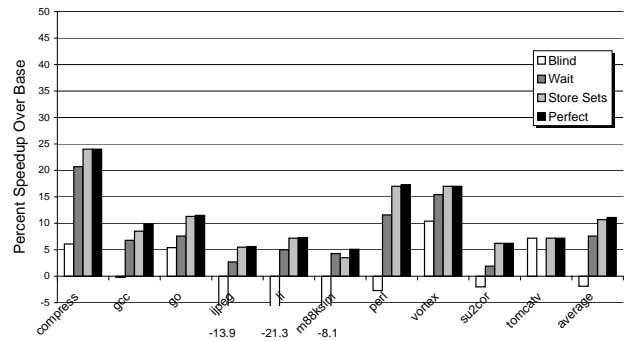


Figure 1: Percent speedup over baseline architecture for dependence prediction with squash recovery.

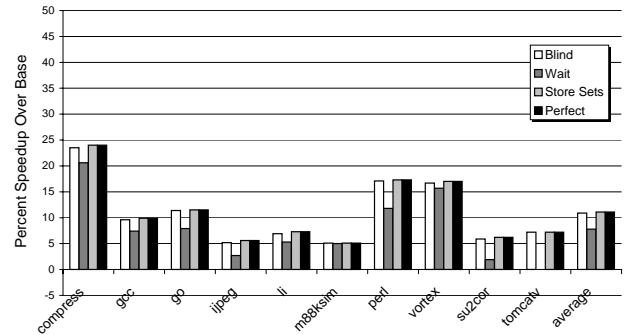


Figure 2: Percent speedup over the baseline architecture for dependence prediction with reexecution recovery.

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 [6].

### 3.1.4 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.

## 3.2 Performance of Dependence Predictors

Figures 1 and 2 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 the same performance as 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. 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.

| Program  | Dependence Predictor |      |      |            |       |      |      |
|----------|----------------------|------|------|------------|-------|------|------|
|          | Blind                | Wait |      | Store Sets |       |      |      |
|          |                      | % mr | % ld | % mr       | Indep |      | Dep  |
|          |                      |      |      | % 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  |
| jpeg     | 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  |
| m8ksim   | 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  |
| 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  |
| average  | 100.0                | 87.2 | 0.1  | 77.8       | 0.0   | 22.2 | 0.0  |

Table 3: Prediction statistics for dependence prediction.

## 4 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 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.

Several predictors have been proposed for address prediction, specifically to be used to reduce the latency of load instructions via prefetching [5, 1, 8]. Gonzalez and Gonzalez [11] 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. [2] 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.

### 4.1 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.

#### 4.1.1 Last Value (Address) Prediction

A last value predictor (LVP) [16] 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.

#### 4.1.2 Stride

A *stride* predictor [5, 8, 23] 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 [8, 23], 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.

#### 4.1.3 Context

A *context* predictor [23, 24, 26] 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 VHT, contains these last 4 values per entry. Another cache, called the VPT, 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 (or folded) using an xor hash into a single index into the VPT. This entry in the VPT contains the value to be predicted. Since a load entry can contain a number of different combinations of history values, it is necessary to make the VPT much larger than the VHT. Our VPT has 16K entries. Thus, the context predictor is able to keep track of a finite number of reference patterns that are not necessarily constrained by a fixed stride. This is the predictor used in [24], but is much smaller in size. Confidence counters are also used in the VHT to guide prediction. Unlike stride predictors, context predictors do not perform well on values that have not been seen before.

#### 4.1.4 Hybrid

Our hybrid predictor is similar to that proposed in [26] and [2]. 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.

#### 4.1.5 Perfect Confidence

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

## 4.2 Performance of Address Predictors

Figures 3 and 4 show the percent speedup obtained for Last Address Prediction, Stride Prediction, Context Prediction, Hybrid, and Perfect Confidence prediction. Results show that the context predictor does not improve performance for *su2cor* and *tomcatv*, whereas stride prediction provides some benefit. This is to be

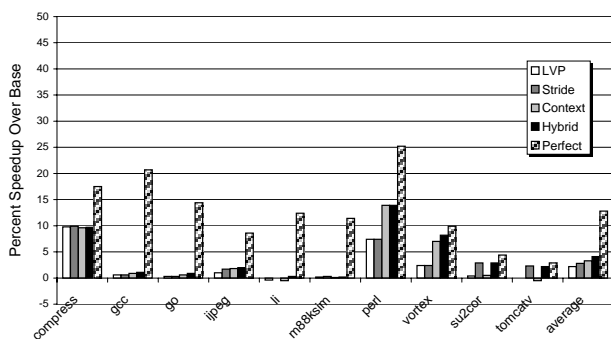


Figure 3: Percent speedup over the baseline architecture for address prediction with squash recovery.

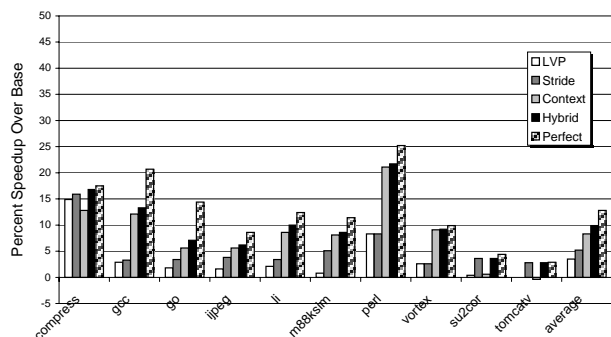


Figure 4: Percent speedup over the baseline architecture for address prediction with reexecution recovery.

| Program  | Address Predictor - Using (31,30,15,1) confidence |         |         |         |         |         |         |         | Perf<br>%<br>ld |
|----------|---|---------|---------|---------|---------|---------|---------|---------|-----------------|
|          | Lvp   |         | Stride  |         | Context |         | Hybrid  |         |                 |
|          | %<br>ld   | %<br>mr | %<br>ld | %<br>mr | %<br>ld | %<br>mr | %<br>ld | %<br>mr |                 |
| 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            |
| jpeg     | 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            |
| 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            |
| average  | 26.9  | 0.1     | 42.4    | 0.2     | 34.1    | 0.5     | 48.8    | 0.3     | 76.9            |

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

| Program  | Using (3,2,1,1) confidence |      |      |      |     |      |      |      |      |
|----------|----------------------------|------|------|------|-----|------|------|------|------|
|          | 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 |
| jpeg     | 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 |
| 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  |
| average  | 0.1                        | 12.1 | 9.4  | 5.2  | 0.0 | 9.1  | 32.9 | 1.0  | 30.2 |

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.

expected, since stride prediction should accurately predict array traversals. Context prediction is better suited for predicting pointer addresses and provides decent speedups for the C programs.

Table 4 shows the percent of loads predicted 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.

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 only be counted in one column (all the columns for a program will sum up to 100%). The results show that on average 17.3% (12.1%+5.2%) of the loads are correctly predicted by stride prediction, and context prediction either chooses not to predict or incorrectly predicts them. However, 9.4% of the load addresses are correctly predicted by context, but not by stride. The stride predictor has more coverage at predicting the FORTRAN addresses, whereas the context prediction allows more coverage for the C programs.

## 5 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 is value predicted, the value is used to update the current value and status of the load's destination register. This value will then be seen and used by subsequent instructions. The load still takes its normal path of execution for a non-speculative load — it calculates its effective address, accesses the store buffer and memory. When the load's value becomes available it is checked against the predicted value of the load instruction for miss-speculation. This is called a *check-load*, since the load is used to check the predicted value for misprediction.

Several architectures have been proposed for value prediction including last value prediction [16, 17], stride prediction [9, 12], context predictors [23], and hybrid approaches [26].

### 5.1 Value Prediction Techniques Examined

We simulate the five prediction architectures described in Section 4.1 for value prediction:

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

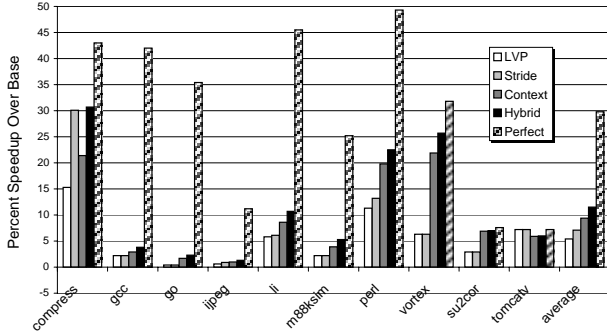


Figure 5: Percent speedup over the baseline architecture for Value Prediction with Squash recovery.

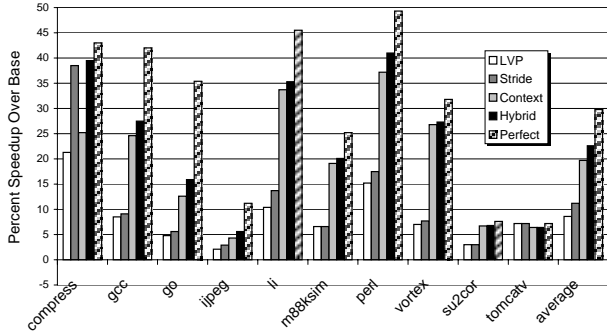


Figure 6: Percent speedup over the baseline architecture for Value Prediction with Reexecution recovery.

| Program  | Value Predictor - Using (31,30,15,1) confidence |      |        |      |         |      |        |      |      |      |
|----------|---|------|--------|------|---------|------|--------|------|------|------|
|          | Lvp   |      | Stride |      | Context |      | Hybrid |      | Perf | % ld |
|          | % ld  | % mr | % ld   | % mr | % ld    | % mr | % ld   | % mr |      |      |
| 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 |      |
| jpeg     | 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 |      |
| 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 |      |
| average  | 26.0  | 0.3  | 28.9   | 0.3  | 29.3    | 0.6  | 36.4   | 0.4  | 65.0 |      |

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

| Program  | Using (3,2,1,1) confidence |      |      |      |     |     |      |      |      |  |
|----------|----------------------------|------|------|------|-----|-----|------|------|------|--|
|          | 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 |  |
| jpeg     | 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 |  |
| 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 |  |
| average  | 0.1                        | 3.0  | 14.4 | 6.3  | 0.0 | 1.2 | 31.7 | 1.1  | 42.2 |  |

Table 7: Breakdown of correct value predictions. 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.

| Program  | % DL1 misses correctly predicted by value prediction |      |      |      |           |      |      |      |      |      |
|----------|--|------|------|------|-----------|------|------|------|------|------|
|          | Squash   |      |      |      | Reexecute |      |      |      |      | perf |
|          | lvp  | str  | ctx  | hyb  | lvp       | str  | ctx  | hyb  |      |      |
| 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 |      |
| jpeg     | 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 |      |
| 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 |      |
| average  | 12.2   | 12.3 | 12.3 | 16.2 | 22.3      | 24.9 | 22.0 | 30.1 | 42.4 |      |

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

## 5.2 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 5 and 6 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 predicting when the prediction has a high degree of confidence, using the 5 bit counter described in Section 2.4. The results show that squash recovery is able to achieve close to a 12% speedup on average and 23% speedup for re-execution recovery. A 30% reduction in execution time could be achieved with perfect confidence.

Table 6 shows the percent of loads predicted 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. 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 9.3% of loads are covered by stride prediction and not by context, but more (14.4%) are only covered by context prediction.

Table 8 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 12% to 22% of the data cache misses on average.

## 6 Memory Renaming

Research by Moshovos et al [21] and Tyson and Austin [25] found that memory communication between store and load instructions can be accurately predicted in hardware. Both of these approaches used special store buffers to find the load/store dependencies, and then an additional buffer to record the relationships found. Once a stable store/load relationship was identified via confidence counters, the hardware would forward store results directly to dependent loads, thereby improving the speed of communication through memory.

| Program  | Memory Renaming |       |      |       |           |       |  |         |       |      |       |           |       |  |         |       |       |
|----------|-----------------|-------|------|-------|-----------|-------|--|---------|-------|------|-------|-----------|-------|--|---------|-------|-------|
|          | Original        |       |      |       |           |       |  | Merging |       |      |       |           |       |  |         |       |       |
|          | Squash          |       |      |       | Reexecute |       |  | Squash  |       |      |       | Reexecute |       |  | Perfect |       |       |
|          | SP              | % lds | % MR | % DL1 | SP        | % DL1 |  | SP      | % lds | % MR | % DL1 | SP        | % DL1 |  | SP      | % lds | % DL1 |
| compress | 9.3             | 39.0  | 0.0  | 0.1   | 9.6       | 0.4   |  | 6.4     | 16.0  | 0.1  | 0.0   | 10.4      | 0.0   |  | 11.0    | 46.6  | 1.6   |
| gcc      | 3.0             | 18.1  | 0.9  | 4.6   | 8.1       | 18.2  |  | 1.5     | 7.5   | 0.5  | 0.3   | 4.1       | 2.4   |  | 12.6    | 41.8  | 26.3  |
| go       | 3.8             | 15.6  | 0.9  | 2.2   | 9.5       | 14.4  |  | 1.9     | 8.3   | 0.2  | 0.0   | 5.1       | 1.0   |  | 18.0    | 38.7  | 23.4  |
| jpeg     | 1.3             | 14.2  | 0.7  | 0.0   | 2.6       | 48.0  |  | 0.7     | 12.0  | 0.7  | 0.0   | 3.0       | 0.0   |  | 4.9     | 39.9  | 48.5  |
| li       | 4.7             | 29.1  | 0.4  | 35.0  | 10.5      | 44.8  |  | 5.9     | 15.7  | 0.2  | 0.0   | 12.1      | 0.7   |  | 12.8    | 43.6  | 51.5  |
| m8ksim   | 5.6             | 37.5  | 0.6  | 0.6   | 10.6      | 56.0  |  | 6.8     | 21.8  | 0.7  | 0.1   | 10.0      | 0.3   |  | 11.7    | 61.1  | 64.7  |
| perl     | 13.6            | 41.4  | 0.1  | 0.8   | 15.1      | 17.5  |  | 8.8     | 22.0  | 0.0  | 0.2   | 10.7      | 0.2   |  | 20.3    | 53.4  | 27.8  |
| vortex   | 9.6             | 34.6  | 0.1  | 28.1  | 10.7      | 29.6  |  | 4.3     | 15.4  | 0.1  | 0.0   | 5.6       | 0.0   |  | 14.0    | 46.6  | 31.8  |
| su2cor   | 5.2             | 45.2  | 0.1  | 51.1  | 4.9       | 51.5  |  | 2.0     | 7.6   | 0.0  | 0.1   | 4.4       | 12.8  |  | 5.1     | 46.3  | 52.5  |
| tomcatv  | -0.0            | 0.0   | 0.2  | 0.0   | 0.0       | 0.0   |  | 0.0     | 0.0   | 0.0  | 0.0   | 0.0       | 0.0   |  | 0.0     | 0.3   | 1.0   |
| average  | 5.6             | 27.5  | 0.4  | 12.2  | 8.2       | 28.0  |  | 3.8     | 12.6  | 0.2  | 0.1   | 6.5       | 1.8   |  | 11.0    | 41.8  | 32.9  |

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

Memory renaming keeps track of store/load dependencies in order to directly communicate a 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. If store has not been resolved, the value will be forwarded to the load from the store when it becomes ready.

As in value prediction, the predicted value is used to update the current value and status of the load’s destination register, and the load (check-load) takes its non-speculative path of execution through the processor. One difference is renaming may predict a dependence for a value that has not yet been calculated. In this case, the register status is updated to point to the instruction that will be producing the value. This is the same instruction that is producing the value for the store that the load predicts that it is dependent upon. As in value prediction, the check-load instruction is used to verify that the correct value was predicted. If a miss-speculation occurs, the value of the check-load is used.

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 value of the load. With dependence prediction the load still performs the store buffer and cache access before dependent instructions may use the speculative load’s value. 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.

## 6.1 Memory Renaming Techniques Examined

### 6.1.1 Original Renaming

We chose to study the communication approach of Tyson and Austin [25], and we obtained the modified SimpleScalar simulator used in their study. The memory communication predictor consists of a direct mapped 4K entry store/load table, a 1K entry value file, and a direct mapped 4K entry store address cache [25]. Both loads and stores index into the store/load table, which contains an index into the value file. Stores write their addresses into the store address cache, and their values or the instruction input dependency (if the value is not ready) into the value file entry specified by their store/load cache entry. When the load issues it accesses the store address cache as well, and if an aliasing store is found, the load is assigned the value file entry of that store for the next prediction.

Loads that do not hit in the store address cache are assigned new value file entries and use last value prediction. Loads look for their speculative value in the value file specified by their store/load table entry. If a value is in the file for the load, it will use the value similar to value prediction. Otherwise a dependency will be stored in the value file entry and the load must wait until that instruction has completed execution before it has a value for prediction. As with other tables, we use confidence counters in the store/load table to reflect the relative predictability of a value file entry, as described in Section 2.4.

### 6.1.2 Merging Renaming

To try and improve the performance of renaming we experimented with another form of renamer. Renaming deals with pairings of loads and stores, but often a load is conditionally dependent on more than one store, or visa versa. We tried to mimic the clustering of loads and stores using the same type of merging mechanism from store sets [6]. The store set to which a load or store belongs serves as the index into the value file. We still keep the store address cache and the store/load cache, and still detect load/store relationships in the manner described above, but we do not always allocate new value file entries when a load/store relationship is detected. Only if both the load and the store in the new relationship do not already possess value file entries is an allocation made. As in store sets, if both the load and the store have value file entries, the lesser of the two entry indices is chosen as the id for both the load and store. We also flush the STLD cache every 1 million cycles as in store sets. To retain the benefits from last value prediction, loads that do not alias an address in the store address cache are given their own value file entries.

## 6.2 Performance of Memory Renaming

Table 9 shows the speedup results for the original renaming architecture, merge renaming, and the original renaming with perfect confidence prediction. The first four columns for each predictor shows the percent speedup for squash over the baseline IPC, the percent of loads predicted for squash, the misprediction rate for squash, and the percent of load data cache misses correctly predicted using memory renaming. The next two columns for each predictor 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.

The results show that merge renaming performed worse than original renaming for all but 3 programs. This is due to the interference between too many overlapping load and store dependencies using the same value entry for prediction. This form of merging is

fine for dependence prediction, since a dependence prediction can be a correct prediction even if it is to an incorrect store, as long as the load doesn't issue before its true store dependencies effective address is calculated. However, for memory renaming a false dependency has a much higher chance of causing a misprediction.

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.

## 7 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, 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 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 (check-load) has to go through the baseline hardware memory disambiguation before the check-load can issue. This can create a long misprediction penalty for load value prediction and memory renaming. Since dependence and address prediction are very accurate, these could potentially be used to speculate the check-load instruction 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 prediction can be beneficial when the check-load prediction is correct and the value prediction is incorrect, decreasing 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.

### 7.1 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

| Using (3,2,1,1) confidence |      |      |      |      |      |      |     |      |     |
|----------------------------|------|------|------|------|------|------|-----|------|-----|
| Program                    | d    | da   | vd   | rd   | vda  | rda  | rvd | rvda | oth |
| 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 |
| jpeg                       | 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 |
| 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 |
| average                    | 20.1 | 15.0 | 3.4  | 6.1  | 24.3 | 3.2  | 3.0 | 24.8 | 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, Oth = 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.

(store sets, hybrid address, hybrid value, and original memory renaming) from the previous sections in this paper, and used the table sizes described in those sections.

Figure 7 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. For 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 7 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.

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 the contribution columns that were essentially all zero. Their contribution is instead accumulated in the last column (Other) in the table.

In Figure 7, 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. From the data cache miss results in Tables 8 and 9, value prediction correctly predicts more loads which are stalled on data cache misses than memory renaming. In fact, most of the correct predictions for data cache miss loads seen by renaming are from the last value prediction nature of the renaming architecture. When the renaming architecture correctly predicts a store dependency, the store's data will most likely still be in the data cache.

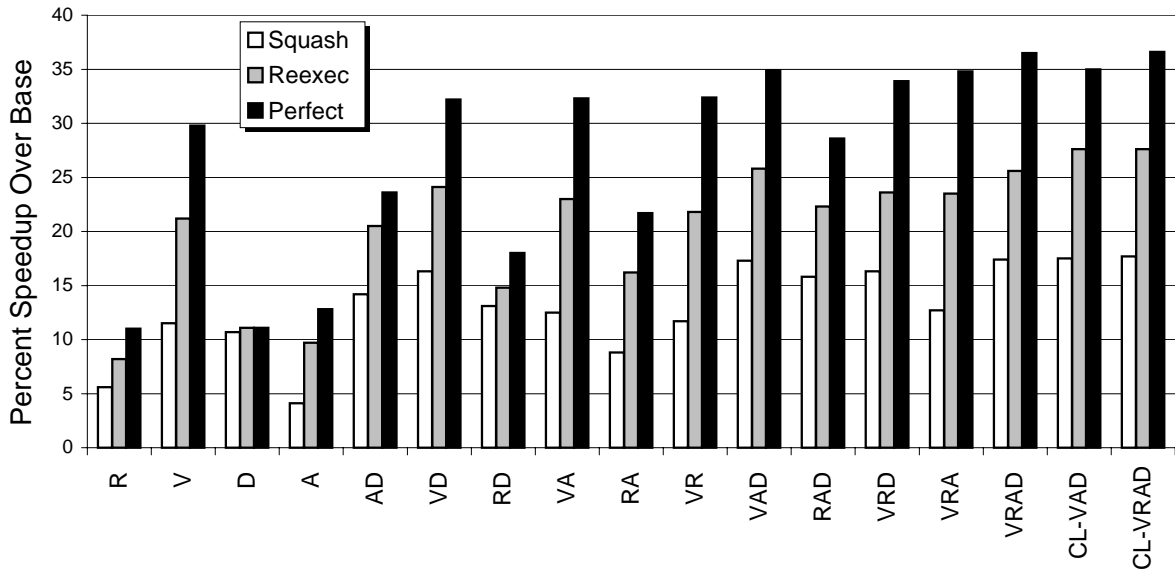


Figure 7: Average speedup results for Squash and Reexecution recovery for all combinations of the predictors using the Load-Spec-Chooser to decide which predictor to use. D = Store Set Dependence Prediction, V = Hybrid Value Prediction, A = Hybrid Address Prediction, R = Original Memory Renaming, and CL = Check-Load Prediction

## 8 Summary

As execution windows continue to grow, in an effort to expose more instruction level parallelism, and as load latencies increase, it becomes imperative to have aggressive load speculation. 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 [25] and examine using the merging technique of Store Sets [6] to communicate the values for memory renaming. 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 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 for value prediction, and 10.5% for 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 [4].
- 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. A study is needed to quantify program behavior for different parts of the program, finding out how applicable a given simulation sample is to the rest of the program. 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.

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. In doing so, some predictors were larger and used more hardware resources than others. The hardware resources for address and value stride prediction was roughly 1/2 that of the data cache. The resources for memory renaming were about equal to the size of the data cache. The resources for the context predictor was roughly twice the size of the data cache. The size of the store sets predictor is about 1/32 that of the data cache, making it a very cost effective design.

Given the performance gains from dependence prediction and its small hardware needs, dependence prediction should be added to future processor designs. 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 would be the next speculation technique to evaluate adding to a processor, because of its potential performance gains. We are currently examining the tradeoffs performed in this paper assuming a fixed amount of hardware resources for each architectural combination.

## Acknowledgments

We would like to thank Todd Austin, Alan Eustace, Kourosh Gharachorloo, and Norm Jouppi and the anonymous reviewers 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] T. M. Austin and G. S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *28th Annual International Symposium on Microarchitecture*, pages 82–92, December 1995.
- [2] 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.
- [3] 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.
- [4] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. Technical Report UCSD-CS98-597, University of California, San Diego, September 1998.
- [5] T-F. Chen and J-L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.
- [6] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *25th Annual International Symposium on Computer Architecture*, June 1998.
- [7] 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*, pages 333–344, June 1995.
- [8] R. J. Eickemeyer and S. Vassiliadis. A load instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37:547–564, July 1993.
- [9] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institute of Technology, November 1996.
- [10] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *International Conference on Parallel Processing*, pages 245–257, August 1991.
- [11] J. Gonzalez and A. Gonzalez. Speculative execution via address prediction and data prefetching. In *11th International Conference on Supercomputing*, pages 196–203, July 1997.
- [12] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ilp. In *12th International Conference on Supercomputing*, 1998.
- [13] D. Grunwald, A. Klauser, S. Manne, and A. Pleskun. Confidence estimation for speculation control. In *25th Annual International Symposium on Computer Architecture*, June 1998.
- [14] E. Jacobsen, E. Rotenberg, and J.E. Smith. Assigning confidence to conditional branch predictions. In *29th International Symposium on Microarchitecture*, December 1996.
- [15] R.E. Kessler, E.J. McLellan, and D.A. Webb. The alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, December 1998.
- [16] 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*, pages 138–147, October 1996.
- [17] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, December 1996.
- [18] M.H. Lipasti and J.P. Shen. The performance potential of value and dependence prediction. In *EUROPAR-97*, August 1997.
- [19] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.
- [20] 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.
- [21] A. Moshovos and G.S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *30th International Symposium on Microarchitecture*, December 1997.
- [22] 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.
- [23] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, December 1997.
- [24] Y. Sazeides and J. E. Smith. Modeling program predictability. In *25th Annual International Symposium on Computer Architecture*, June 1998.
- [25] G. Tyson and T. M. Austin. Improving the accuracy and performance of memory communication through renaming. In *30th Annual International Symposium on Microarchitecture*, pages 218–227, December 1997.
- [26] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, December 1997.