Classifying Load and Store Instructions for Memory Renaming

Glenn Reinman† Brad Calder† Dean Tullsen† Gary Tyson‡ Todd Austin*

†Department of Computer Science and Engineering, University of California, San Diego
‡Electrical Engineering and Computer Science Department, University of Michigan
*Microcomputer Research Labs, Intel Corporation

Abstract

Memory operations remain a significant bottleneck in dynamically scheduled pipelined processors, due in part to the inability to statically determine the existence of memory address dependencies. Hardware memory renaming techniques have been proposed to predict which stores a load might be dependent upon. These prediction techniques can be used to speculatively forward a value from a predicted store dependency to a load through a value prediction table. However, these techniques require large, time-consuming hardware tables.

In this paper we propose a software-guided approach for identifying dependencies between store and load instructions and the Load Marking *(LM) architecture to communicate these dependencies to the hardware. Compiler analysis and profiles are used to find important store/load relationships, and these relationships are identified during execution via hints or an n-bit tag. For those loads that are not marked for renaming, we then use additional profiling information to further classify the loads into those that have accurate value prediction and those that do not. These classifications allow the processor to individually apply the most appropriate aggressive form of execution for each load.*

1 Introduction

Accurate determination of memory dependencies between store and load instructions will be one of the keys to performance on future superscalar processors. Compiler techniques have been developed for finding loop carried dependencies and disambiguating potentially aliased loads and stores. When these dependencies can be determined, the compiler can effectively expose instruction level parallelism and loop-based parallelism. The problem of determining these dependencies in a timely fashion during execution has proven more difficult. Processors allow loads and stores to execute out-of-order by comparing the load and store addresses. The latency in calculating the addresses to do this comparison can be several cycles, delaying memory re-ordering until late in the pipeline. Therefore, dynamic scheduling of memory operations is constrained not only by real dependences, but often by non-existent dependencies.

Two hardware approaches were recently proposed that find dependencies between the load and store instructions and then use these dependencies during the fetch stage of the processor for *Memory Renaming* [14, 20]. The hardware would predict which loads were dependent upon stores to enable loads to bypass memory by communicating the value of the store directly to the load instruction. As a by-product, this technique also enables accurate value prediction [11, 21, 18] for some loads. While these hardware techniques were successful at finding store/load dependencies, they require a large store cache of prior store instructions to find the store/load dependencies, a buffer to hold the dependencies found, and a value file to provide the predicted value.

The goal of this paper is to extend the prior memory renaming technique of Tyson and Austin [20]. The memory renaming architecture either (1) communicated a value from a store to a load for value prediction, or (2) provided last value prediction for load instructions. We propose the *Load Marking* (LM) architecture to use profiles to accurately classify which instructions would benefit more from (1) memory communication, (2) last value prediction, or (3) neither of these prediction techniques. The Load Marking architecture performs this classification and labels those instructions using profiling hints. Our results show that these store/load dependencies can be accurately found via compiler analysis and profiling. This reduces the amount of required hardware for memory renaming, and improves the performance of proposed memory renaming architectures by providing hardware hints or explicit tags in the instructions.

We examine two techniques to communicate load-store dependences to the architecture: load marking hints (LM hints), and a small n-bit tag (which we call the Memory Renaming Tag, MRT). When LM hints are used, they provide a filter indicating which store and load instructions will contribute positively to hardware memory renaming. When MRT tags are used, the tags are assigned by the compiler and are contained within the store and load instruction. A store and load with the same tag are mapped to the same value prediction entry in a table similar to hardware Memory Renaming [20].

Other loads are marked as either candidates for load value prediction or no prediction. By only applying the appropriate optimization to each load, mis-speculations are reduced dramatically. Using these techniques, we are able to identify nearly 37% of all dynamic instructions to use values produced by either store-load prediction or value prediction, and achieve more than a 98% prediction accuracy.

In section 2 we describe the motivation for this research and related work. We then describe the functionality of the memory communication tag architecture in section 3. Section 4 briefly describes the profiling techniques we developed to capture the store/load relationship information needed to accurately allocate the MRT tags. In section 5 we describe the algorithm used for allocating the tags to the load/store instructions. The profile and simulation methodology is described in section 6, and the results are described in section 7. We conclude the paper in section 8.

Published in the Proceedings of the International Conference on Supercomputing, June 1999.

2 Related Work

The Load Marking architecture provides static prediction for memory communication and value prediction. Hardware solutions for these problems have been proposed, each showing improvements in processor performance.

2.1 Memory Address Disambiguation

A number of dynamic memory disambiguation techniques have been proposed to improve the accuracy of dependence speculation [6, 8, 15].

The Memory Conflict Buffer (MCB) proposed by Gallagher et al. [8] provides a hardware solution with compiler support to allow load instructions to speculatively execute before stores. The addresses of speculative loads are stored with a conflict bit in the MCB. All potentially ambiguous stores probe the MCB and set the conflict bit if the store address matches the address of a speculative load. The compiler inserts a check instruction at the point where the load is known to be non-speculative. The check instruction checks the speculative load's conflict bit in the MCB; if not set, the speculation was correct, otherwise the load was mis-speculated.

A similar approach for software-based speculative load execution was proposed by Moudgill and Moreno [15]. Instead of using a hardware buffer to check addresses, they check values. They allow loads to be speculatively scheduled above stores, and in addition they execute the load in its original location. They then check the value of the speculative load with the correct value. If they are different a recovery sequence must be executed.

A pure hardware approach for speculative load execution proposed by Franklin and Sohi [6], called the Address Resolution Buffer (ARB), directs memory references to bins based on their address and uses the bins to enforce a temporal order among references to the same address. The use of bins reduces the associativity of the search and allows for multiple disambiguation requests in one cycle, since the disambiguation process is decentralized and localized to a bin. The ARB provides forwarding of values from store to load instructions that are in the current instruction window.

2.2 Value Prediction

Lipasti et al. [11] describe a mechanism in which the value of a load instruction is predicted based on the previous values loaded by that instruction. In their work, they used a value table to store the values to predict and a confidence mechanism for deciding whether the value is likely to be correct based on past performance of the predictor. Further work has looked at predicting the value of instructions using stride, context predictor, and hybrid predictors [16, 18, 21].

Recent research has shown that instruction values have predictable behavior between different inputs [2, 3, 7]. These studies showed that profiling can be used to accurately guide last value prediction. Our research extends these profile-guided techniques by not only predicting the value for load instructions, but also predicting store/load relationships.

2.3 Memory Renaming

Research by Moshovos et. al. [14] and Tyson and Austin [20] found that memory communication between store and load instructions can be accurately predicted in hardware. Both of these approaches use special store caches to find the store/load dependencies, and then an additional buffer to record the relationships found. Once a stable store/load relationship was identified, the hardware would forward store results directly to dependent loads, thereby improving the speed of communication through memory. Our research shows that profiling can accurately find the important store/load dependencies.

To evaluate the effectiveness of our Load Marking architecture, we compare it to our previous hardware memory communication approach [20] shown in Figure 1. We used a modified version of the SimpleScalar simulation tools from this prior study; we use a derivative of that simulator to provide our new results and an IPC comparison with the original memory renaming architecture. For effective memory communication, the architecture has (1) a Store Cache to cache stores recently seen, (2) a Store/Load Cache to hold the dependencies found, (3) a Value File for rename/value prediction, and (4) a confidence mechanism (not shown in the Figure) to determine when to use the prediction. The next two paragraphs summarize the functionality of the memory renaming architecture for store and load instructions.

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 for last value prediction. If there was no store alias, then the load updates its Value File entry with the last value used by the load. For further details, please see the complete description of the memory renaming architecture in [20]. The goal of our Load Marking architecture is to simplify the hardware, by providing the communication between store and load instructions in software only requiring a Value File for prediction.

3 Adding Load Marking to Memory Renaming

The Load Marking architecture extends the instruction set architecture to identify different classifications of loads. For a given store or load, this classification identifies which part of the memory renaming architecture [20] the store or load can benefit from. Note that the proposed memory renaming architecture in [20] contains two parts. Either values can be communicated from stores to loads via the value file, or loads can benefit by using the value file for last value prediction. Therefore, in our classification loads can be candidates for either:

- *Rename* - Loads that have been identified as good candidates for static memory renaming.
- *Value Predict* - Loads that are found to be predictable above a certain threshold by last value prediction, but do not exhibit sufficient load/store relationship predictability.
- *Non-Speculative* - Loads that were not found to be sufficiently predictable.

Load marking allows the processor hardware to handle each load in the most efficient manner for speculative execution. At a minimum, an architecture to support load marking requires (1) ISA

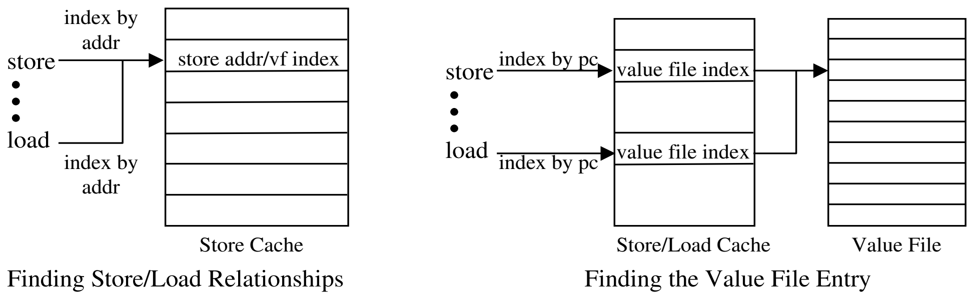

Figure 1: The structure of the memory renaming architecture [20]. The Store Cache is used to find the relationships between store and load instructions. The Store/Load cache is used to keep track of which Value File entries to use for store and load instructions. Store instructions use the value file 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.

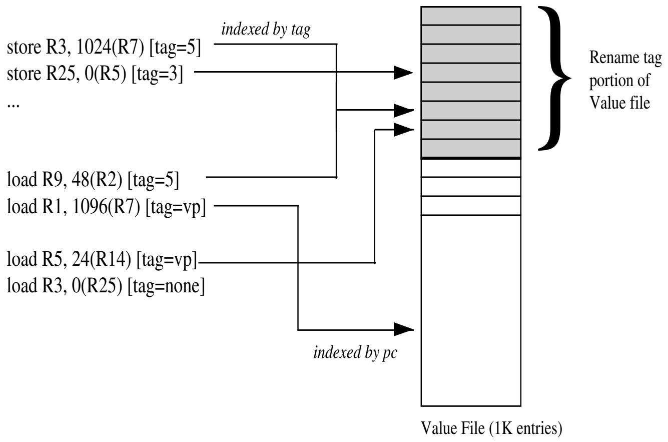

Figure 2: The structure of the value file for the Load Marking MRT architecture. A portion of the value file is used for static memory renaming.

bits to identify a load's classification, and (2) the value file to provide renaming and value prediction.

3.1 Implementation of Load Marking

In this paper we examine two Load Marking architectures. The first, *LoadMark Hints*, uses profiling to create hints to classify the type of speculation to use for a given load. The second architecture, *LoadMark MRT*, uses profiles to find dependencies between store and load instructions, and uses memory rename tags (MRT) to explicitly direct value forwarding from a store to a load instruction.

3.1.1 Load Marking Hints for Memory Renaming

Stores and loads that are found via profiling/compiler analysis to benefit from memory renaming are marked for renaming. Loads which benefit more from value prediction are marked for value prediction, and all other loads are marked as non-speculative. To provide LoadMark hints, the architecture needs to provide three new opcodes (rename-store, rename-load, value-predict-load). The compiler analysis uses profiles to filter out stores and loads from the memory renaming architecture. Stores not marked for renaming do not update the Store Cache. Load's marked as non-speculative do not use the Store/Load Cache shown in Figure 1. Load's marked for value prediction index into the Value File directly using their PC. This helps reduce the pressure on the renaming architecture, especially since only a small portion of the static store and load instructions contribute to renaming. This significantly reduces the size of the Store Cache and Store/Load Cache needed.

3.1.2 Load Marking Architecture with MRT Tags

For the LoadMark MRT architecture, dependencies between store and load instructions are represented by an MRT tag. Renaming occurs when the store and load have the same tag. This MRT tag serves as an index into the memory renaming Value File [20], which communicates the value of the store to the load during execution. The MRT tag and the bits needed to classify a load can be represented either with extra bits in load and store instructions, or with special marker instructions in the code. We assume the former.

Classification and tagging would require the 3 new opcodes (rename-store, rename-load, value-predict-load). In addition, log(n) bits added to the load and store instruction format, where n specifies the number of possible tags.

For loads marked for rename prediction, the Value File is used to communicate values between store and load instructions with the same MRT. This same value file is used to predict values for load instructions marked for last value prediction in the memory renaming architecture.

The MRT tags are used as the index into the Value File for tagged stores and loads. When a store with a valid MRT tag is entered into the reorder buffer (ROB), it updates the Value File with either the value of the store (to be written) or a pointer to the ROB entry that will produce the value for the store. After a load with a valid MRT tag is fetched, the MRT tag is used as a direct index into the Value File retrieving either a value or a pointer to a ROB entry.

When the number of MRT tags is less than the size of the Value File, rename prediction only uses the first n entries (n being the number of tags) in the Value File. Figure 2 shows the Load Marking MRT architecture examined in this paper. The only hardware needed is a Value File used to communicate the renaming, and this same Value File is used for value prediction. We report results for a Load Marking MRT architecture that uses 8 tags (3 bits). We examined using more tags (e.g., 1024 tags), but this provided only a minor improvements over an 8 MRT tag architecture.

Figure 2 shows that stores and loads with the same tag will map directly to the same entry in the Value File. Loads marked for value prediction use their PC to directly index into the Value File. These loads blindly value predict using the value stored in the table entry, ignoring all renaming. It is similar to the static profile-based value prediction scheme proposed in [7, 11].

Since the store/load relationships are represented by tags, the MRT architecture no longer needs the Store Cache and Store/Load Cache shown in Figure 1.

3.2 Non Speculative Loads

Loads that do not use memory renaming or value prediction use normal hardware alias detection to order loads with respect to stores. The technique we simulate is similar to the load/store disambiguation methods used by the Pentium Pro [13] or the Power PC-620 [10]. Loads are stalled until all earlier store addresses are known. Once all earlier store addresses are available, the disambiguator can determine if the load value should be forwarded from a store in the store buffer or from the data cache. If the load is sourced by a store in the store buffer and the store data is not yet available, the load stalls until it is available; otherwise, the load is scheduled to execute.

3.3 Consuming Predicted Values

The memory renaming architecture we model in this paper uses selective prediction [4], which only pays 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. Our architecture is different than memory renaming described in [20], since it delays the decision of using a predicted value until late in the pipeline.

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. In our architecture, 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.

We modified the register file 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 misspeculation penalty. We used this architecture in a prior selective value prediction study, and a more detailed description can be found in [4].

3.4 Mis-speculation Recovery

Each of the prediction schemes has the potential to violate dependencies. Memory renaming and value prediction can mispredict the dependence and mispredict the value.

We examine the same mis-speculation recovery options as [20]: squash and re-execute. Squash recovery re-fetches the instructions from the cache on a mispredict, and is analogous to branch misprediction recovery. Re-execution recovery re-executes only those instructions dependent (directly or indirectly) on the mis-speculated load. This is accomplished by re-injecting the correct loaded value onto the result bus. Instructions that had used the speculative value would see the new value and re-execute, which may in turn cause more re-executions.

For the simulation results in this paper, we assume a minimum 8 cycle misprediction penalty (for both value and rename mispredicts) for squash recovery. The penalty will be longer if the incorrectly predicted load instruction is stalled in the processor pipeline. The instructions following the mispredicted load are then re-fetched just as in a branch misprediction. For re-execution recovery, the penalty accrued is the delay from resending the instructions through the processor pipeline, starting with instruction issue, once their dependencies are ready.

4 Profiling Analysis

Our load marking and dependence tagging architecture depends on profile analysis of dynamic load-store behavior. This section briefly describes the two types of profiling information we collect to classify the loads for memory renaming, value prediction, and non-speculation. These profiles are then fed back into the compiler to perform the load marking hint and rename tag allocation described in section 5. To find the relationships between store and load instructions, we record the addresses written by store instructions; these addresses are matched when a load instruction is executed.

In finding store/load dependencies we are only trying to find the dependencies that the memory renaming architecture can accurately predict. The renaming architecture cannot correctly predict loop carried dependencies with a dependence distance greater than one loop iteration, because a static characteristic of the load and store instruction (either the PC or the assigned MRT) is always used to index the prediction tables. Thus values from earlier iterations would always be overwritten by more recent iterations (assuming the load and store execute every iteration) for a loop carried dependency with a distance greater than one. See [20] for more details.

The profiler keeps track of a list of store dependencies found for each load. For the programs we examined, we found that on average 31% of the loads had 0 store dependencies (e.g., constant values), 32% of the loads had 1 dependency, and 37% of the loads had 2 or more dependencies. Statistics are kept so that after the profile is generated, we know for each load what the predication accuracy would be if (1) the load was predicted using store/load communication (called the Rename Prediction accuracy) or (2) the load was predicted using Last Value Prediction. These are the two options we have for the memory renaming architecture [20]. See Appendix A for more details on how the profiling was performed.

5 Identifying Loads and Stores for Memory Renaming

The profile data described in the previous section characterizes each load in several ways, including its potential for renaming success. This section describes the process of translating the profile information into specific load marking hints and tagging decisions.

A number of heuristics are applied to the data obtained through profiling to filter out infrequent load-store relationships, infrequently executed loads, and less predictable loads. Live variable analysis (LVA) is then used to direct the coloring of load-store dependences chosen to receive tags for the LoadMark Tag architecture. This allows efficient and non-destructive sharing of tags. This information can then be conveyed to the load marking architecture through the mechanisms described in section 3.

The flow graph that is created to perform this analysis is a global inter-procedural control flow graph of the program based on the execution profile. To make this graph manageable in performing the analysis we apply frequency heuristics described below to partition the graph into highly coupled regions. Therefore, we find store/load relationships and perform the tag allocation across procedure boundaries.

5.1 Rename Candidate Pruning Heuristics

We use a number of heuristics to reduce the number of candidates for memory renaming, and apply them in the following order:

1. *Low Frequency*

   This heuristic removes infrequently traversed edges from the flow graph. Basic blocks which then have no successors or predecessors are removed from the flow graph.

2. *Reach*

   The reach heuristic then removes load/store dependencies that have become partitioned by the low frequency heuristic. Stores which cannot reach a load are removed from the load table entry for that particular load.

3. *Rename Predict*

   The predict heuristic filters out loads with poor memory rename predictability. The prediction metric is calculated assuming all the store relationships found in the profiling phase for that load were given the same tag. If the predictability is above a predict threshold (currently 80%), then the load will be marked as a potential tagged load, and it will be passed to the merge heuristic. The predictability of the load is based on the rename prediction metric for a load, as described in section A.1.

4. *Merge*

   The merge heuristic combines common store dependencies for a particular load under a single tag. This is useful in cases where a load is dependent on multiple stores along different paths of execution. By using the previous Rename Predict heuristic to determine taggable loads, we avoid cases where loop carried dependencies or stores along the same path of execution can make merging unattractive. For the programs we examined, a simple blind merge heuristic performed well, where all remaining store relationships for a load are given the same tag.

5. *Classify*

   The classify heuristic determines the prediction mechanism to be used for each load, if there were infinite tags available. Classify marks each store as *rename* or *normal*, and each load as described in section 3: *rename*, *value predict*, *non-speculative*.

At this point in the algorithm, *no further compiler analysis is needed for the LoadMark Hint architecture*. The steps to classify stores and loads described above, results in a classification to help guide which load and store instructions should use memory renaming and which load instructions should use value prediction.

For the LoadMark MRT architecture, additional compiler analysis is needed to allocate the MRT tags in the store and load instructions. The remainder of this section describes this tag allocation algorithm.

5.2 Allocation of Memory Renaming Tags

The memory tag corresponds to an entry in the value table, used to communicate the value between the related store/load pair. The allocation of memory tags consists of (1) identifying the live paths for store/load relationships, (2) coloring these relationships so that store/load pairs that are live at the same time are given different MRT tags, and (3) allocating tag numbers in such a way as to avoid conflicts with loads which are using the same value table entry for value prediction.

5.2.1 Live Range Analysis

In an effort to reduce the number of tags needed for a particular program, we run a version of live analysis to determine the live range for a tag from a particular store to its dependent loads.

We define the start of the live range in terms of marked store instructions. A store is a *def* of a particular store instruction, and a dependent load is a *use* of that store. The algorithm used to calculate the live range is different than normal live variable analysis, because the store (def) is not necessarily on all predecessor paths of the load. Using normal live analysis, the liveness of the store would tend to propagate all the way up the flow graph when every path containing a use (load) did not also contain a corresponding def (store). Therefore, we perform a depth first search from a store to each load that depends upon it instead, marking all basic blocks along every possible path from store to load as live for that store.

We also include in the live range the reachability of load/load relationships, marking all paths from the load back to itself. The load/load relationships identifying all the paths the MRT tag would need to be unique to allow the load to still benefit from last value prediction. A load may have a store dependency on one path to the load in a loop, but not on another path. In this situation the load will use the predicted value from the store part of the time, and last value prediction (from the load's previous value) the rest of the time. We need to include the load/load path in the live range for the load in order to make sure that the load's value table entry (specified by the tag) is not overwritten by a value from another instruction on the load/load reachability path.

The union of the store/load and the load/load resulting live sets forms the live range for a tagged store and load.

Before allocating the tag, we further reduce the candidate set by removing loads which do not make significant gains using MRT tags over value prediction. This makes tag allocation more efficient and reduces the number of necessary tags. After tag allocation, any loads that were classified for renaming but not given a tag, are then rechecked to see if they could be accurately classified for value prediction.

	Profile Data Set				Simulation Data Set								% Load Overlap	
		Dynamic # instr				Base	# instr	Dynamic # instr			Static lds			
Program	Input	Exe M	% ld	% st	Input	IPC	fastfwd M	Exe M	% ld	% st	total #	% exec	Stat	Dyn
compress	short	9	28.5	11.6	ref	1.1812	10	93	26.3	9.4	3058	18.7	17.0	99.6
gcc	1stmt	337	24.0	10.6	1cp-decl	1.5965	500	1041	23.7	10.3	69691	42.1	39.4	100.0
go	2stone9	546	28.0	7.7	5stone21	1.8543	500	32699	27.5	7.4	17669	76.4	67.7	100.0
li	puzzle	28243	22.9	12.3	ref	2.5794	500	18089	28.2	14.7	7266	28.3	23.6	99.6
perl	jumble	2883	21.9	13.5	scrabbl	2.4251	200	28243	24.4	14.3	20048	21.9	19.4	95.1
hydro2d	train	4447	24.0	7.9	ref	1.1151	500	42785	24.1	7.9	25255	16.6	16.5	100.0
su2cor	train	10744	20.5	9.1	ref	2.6399	500	33928	20.5	9.1	25626	18.1	18.1	100.0
tomcatv	train	4729	27.7	8.4	ref	2.7850	500	27832	27.9	8.4	23206	10.7	10.6	100.0

Table 1: Data sets used in gathering results for each program.

5.2.2 Coloring

Once live ranges have been calculated, loads are assigned colors that will correspond to *virtual tags*. Later, virtual tags will be mapped to physical tags. Our current design assumes that there are as many virtual tags as there are physical tags. The use of virtual tags allows us to defer the assignment of loads to particular value file locations. When a load is colored, other loads which are not live at the same time as the colored load and which do not have overlapping live ranges can be assigned the same color. Since it is possible that there are more loads than tags, loads are assigned colors in priority order, based on the number of correct predictions tag-based prediction would provide *over* the number of correct predictions that ordinary last value prediction would provide.

Once as many loads have been colored as the number of tags permits, the remaining loads are re-classified for value prediction if applicable.

5.2.3 Tag Allocation

Since tagged loads and ordinary value-predict loads will share a portion of the common value file, it is important to prevent as many collisions between loads as possible. We can estimate the frequency of use for each table entry based on our profiles of load activity. We then assign the virtual rename tags that are used the most, based on execution counts, to the physical tags which correspond to the value file entries least-used by value prediction.

6 Evaluation Methodology

To perform our evaluation, we collected information for some of the SPEC95 benchmarks. 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 two data sets we used in gathering results for each program, the number of instructions executed in millions (Exe M), and percent of executed instructions that were loads (%ld) and stores (%st). For the simulation data set we also show the base IPC obtainable for each benchmark without any value speculation, the number of instructions in each benchmark that we fast forwarded past before beginning simulation, the number of static store instructions, and the percentage of these that were executed. The final two columns show the percent of static and dynamic overlap for load instructions between the two input sets. The profiling data set is used to generate the load marking hints and tags. We then use the simulation data set to simulate the programs when gathering the statistical performance results. The Base IPC is smaller than in some prior studies, since we are modeling longer memory latencies (120 cycles to retrieve data from main memory).

We used ATOM [19] to instrument the programs and gather the rename and value profiles. During profiling, we collected data on temporal ordering of dependencies, the size of the MRT candidate set, the average number of stores upon which a load depends, and other useful metrics.

Fetch Interface	delivers two basic blocks per cycle, but no more than 8 instructions total
Instruction Cache	32k 2-way set-associative, 32 byte blocks, 12 cycle miss latency
Branch Predictor	hybrid - 8-bit gshare w/ 16k predictors [12] + 16k bimodal predictors 8 cycle mis-prediction penalty (minimum)
Out-of-Order Issue Mechanism	out-of-order issue of up to 16 operations per cycle, 128 entry re-order buffer, 32 entry load/store queue, loads may execute when all prior store addresses are known
Architecture Registers	32 integer, 32 floating point
Functional Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV
Functional Unit Latency (total/issue)	integer ALU-1/1, load/store-2/1, integer MULT-3/1, integer DIV-12/12, FP adder-2/1 FP MULT-4/1, FP DIV-12/12
Data Cache	32k 2-way set-associative, write-back, write-allocate, 32 byte blocks, 12 cycle miss latency four-ported, non-blocking interface, supporting one outstanding miss per physical register 512k 4-way set-associative, unified L2 cache, 64 byte blocks, 120 cycle miss
Virtual Memory	4K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 2: Baseline Simulation Model.

The simulators used in this study are derived from the SimpleScalar/AXP tool set [1], a suite of functional and timing simulation tools for the Alpha AXP ISA. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch mis-prediction. The baseline micro-architecture model is detailed in Table 2. The minimum misprediction penalty is 8 cycles for branches, and 8 cycles for value prediction and rename prediction when using squash (re-fetch) recovery. The penalty can be longer, based on the latency of a predicted instruction making its way through the pipeline.

Our baseline simulation configuration models a future generation micro-architecture. We've selected the parameters to capture two underlying trends in micro-architecture design. First, the model has an aggressive fetch stage, employing a variant of the collapsing buffer[5]. The fetch unit can deliver two basic blocks from the I-cache per fetch cycle, up to 8 instructions. If future generation micro-architectures 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[17]. 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 more from more precise load/store scheduling. To compensate for the added complexity of disambiguating loads and stores in a large execution window, we increased the store forward latency such that the scheduler requires two cycles to react to any new store address.

We investigated the use of both large and small hardware configurations in our simulation of the memory renaming experiments from [20]. The larger configuration consists of a 1024-entry, 4-way set associative Store/Load cache and a 1024-entry Value File with LRU replacement. To detect initial dependence edge bindings, we use a 1024-entry 4-way associative Store Address Cache. The smaller configuration uses a 256-entry 4-way set associative Store/Load Cache, a 256-entry Value File, and a 256-entry 4-way Store Address Cache.

The original renaming architecture had a 2-bit saturating confidence counter [9] associated with each Store/Load cache entry. As shown in [4], the confidence associated with a prediction is a critical component to achieving performance with value speculation. After experimenting with a variety of confidence levels, we settled on using a 4-bit confidence counter for squash recovery, and no confidence mechanism for re-execution recovery. Since re-execution has a very low mispeculation penalty, it can be advantageous to use all possible predictions. 1-bit confidence provided only slightly better results than no confidence for re-execution recovery, and 2-bit confidence performed worse. Squash recovery has a high misprediction penalty and therefore it is necessary to use high confidence to eliminate as many mispredictions as possible. Our 4-bit counter requires a threshold value of 14 to recommend prediction, is decremented by 7 on a misprediction, and is incremented by 1 on a correct prediction.

7 Performance Results

This section presents simulation results for guiding memory renaming using load marking. We show that software marking of loads and stores can help increase the performance of hardware-based memory renaming and that using explicit tags can perform as well as the hardware-based approach, providing memory renaming without the extra hardware for tracking the dependencies. All these results used the profile input to generate the LoadMark hints and tags, and the simulation input to perform the simulations.

7.1 Memory Renaming

We now examine the performance of the two architectures that use the load marking information, and compare our results to the original memory renaming architecture using selective prediction described in section 3.3.

The first architecture (LoadMark Hints) modifies the original memory renaming architecture to use LoadMark hints to indicate which stores and loads should use memory renaming, and which loads should use last value prediction. Only stores that are marked for renaming are put into the store cache. Only loads that are marked for renaming search the store cache for a dependency. If a dependency is found, it is recorded in the store/load table described in [20], so that both the store and load will use the same value table entry. Loads that are marked as last value prediction use the value table for last value prediction indexed by the load's PC.

The second architecture (LoadMark MRT) uses MRT tags to communicate directly to the hardware which value table entry to use for memory renaming. We use 3-bit tags to communicate the entry, using the tag allocation described in section 5. This allows 8 entries in the value table to be used for communication. This LoadMark MRT architecture only requires the Value File table and a confidence table for prediction. It does not require the Store Cache or Store/Load Cache, like the original memory renaming architecture.

Figures 3 and 4 show the performance for squash miss recovery for Value Files with 1024 and 256 entries respectively. We show the IPC speedup over the baseline architecture for the original memory renaming architecture using selective prediction, the load marking

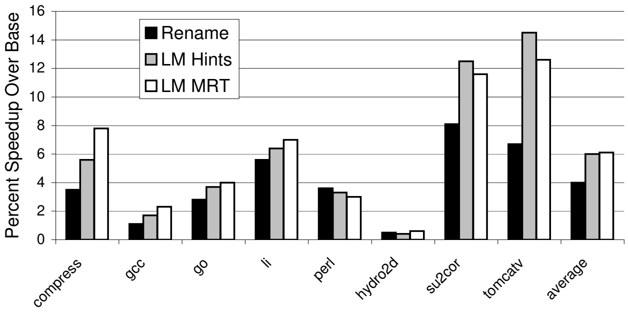

Figure 3: Percent speedup over baseline architecture for squash recovery with 1024 entry prediction tables.

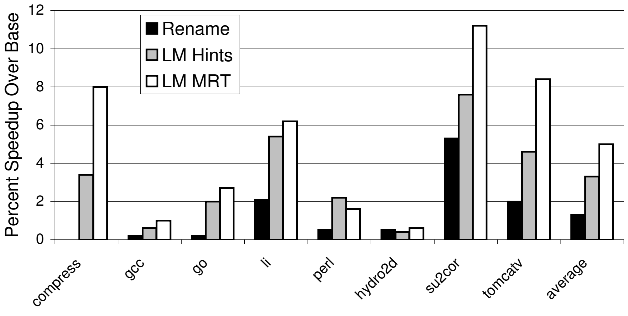

Figure 4: Percent speedup over baseline architecture for squash recovery with 256 entry prediction tables.

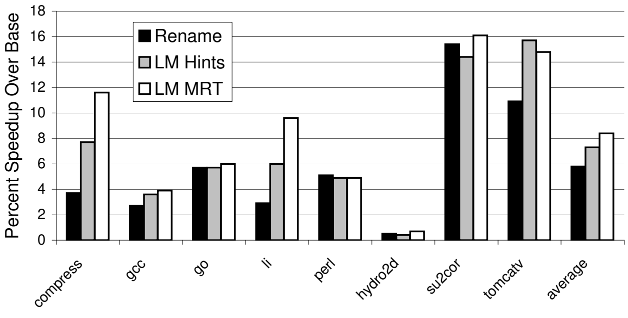

Figure 5: Percent speedup over baseline architecture for reexecution recovery with 1024 entry prediction tables.

hints architecture (LM Hints), and the load marking MRT architecture with hardware confidence prediction (LM MRT). The results for the 1024 entry Value File show that while the hardware renaming scheme provides a 4% speedup in performance, the load marking hints and MRT architectures can provide speedups around 6% by filtering out highly predictable loads and stores. This result is even more dramatic in `tomcatv` and `su2cor`. When the size of the Value File is reduced to 256 entries, rename performance drops to around 1% speedup, while the LM MRT architecture provides a 5% speedup, despite the smaller table size. This is an important result, since the prior work showed little to no improvement for renaming with squash recovery.

Figures 5 and 6 show the performance of re-execution when using 1024 and 256 entry prediction tables as described in section 6. In addition to showing the speedup over the baseline architecture of the rename, LM Hints, LM MRT architectures, we also show what speedups could be achieved with these architectures with perfect confidence prediction for the 256 entry results in figure 6. Perfect confidence prediction eliminates mispredictions by always using a predicted value if it is correct. This will give some measure of the absolute benefit of each technique. Despite the low misprediction penalty of re-execution recovery, the results show that load mark-

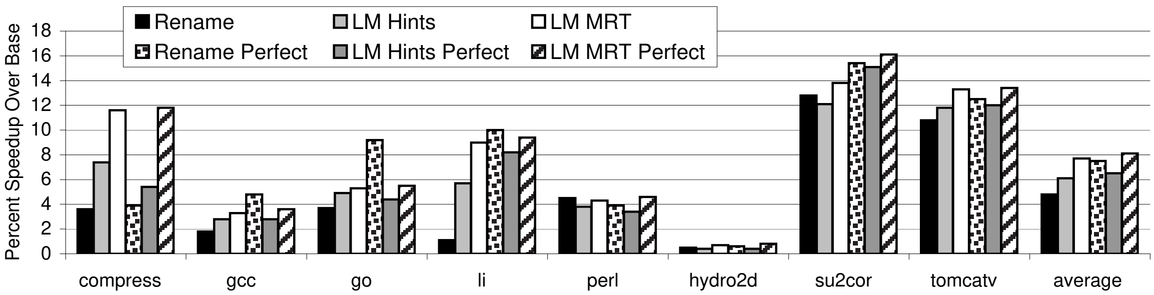

Figure 6: Percent speedup over baseline architecture for reexecution recovery with 256 entry prediction tables.

Programs	Rename		LoadMark Hints		MRT	
	% lds	% MR	% lds	% MR	% lds	% MR
compress	39.1	0.0	45.5	0.0	63.8	1.4
gcc	9.7	2.4	13.7	2.2	16.1	1.6
go	5.6	3.6	5.9	2.2	6.5	2.6
li	21.1	1.2	23.2	0.8	26.1	0.7
perl	32.5	1.7	25.1	1.7	24.8	1.7
hydro2d	57.6	0.3	57.9	0.2	64.3	0.4
su2cor	45.7	0.6	53.9	0.5	52.9	0.5
tomcatv	33.3	0.4	45.3	0.5	37.7	0.6
average	30.6	1.3	33.8	1.0	36.5	1.2

Table 3: Rename Prediction Accuracy for 1024 entry table with squash recovery.

Programs	Rename		LoadMark Hints		MRT	
	% lds	% MR	% lds	% MR	% lds	% MR
compress	30.7	1.7	41.1	1.1	63.8	1.4
gcc	3.9	3.9	7.6	4.3	9.5	3.0
go	2.0	5.1	3.0	3.3	4.5	3.9
li	16.1	2.7	22.3	1.1	24.5	0.8
perl	1.6	7.2	11.8	2.2	17.3	3.2
hydro2d	57.6	0.3	57.9	0.2	64.3	0.4
su2cor	9.9	3.4	19.7	1.8	33.3	0.0
tomcatv	4.4	0.0	10.4	1.1	24.9	0.5
average	15.8	3.0	21.7	1.9	30.3	1.6

Table 4: Rename Prediction Accuracy for 256 entry table with squash recovery.

ing still provides improvement over the original memory renaming architecture.

Tables 3 and 4 show the statistical effects of using load marking. They show the percent of time an instruction used a predicted value from a load for each architecture and the corresponding misprediction rate. For the 1024 entry table, we see that adding load marking hints to the renaming architecture provides an increase in the percent of instructions using predictions from 30.6% to 33.8% on average and decrease the miss rate from 1.3% to 1.0% on average. The use of MRT tags increases the number of predictions used to 36.5%, and reduces the miss rate down to 1.2% on average. Again, this effect is even more dramatic in the 256 entry Value File.

Load Marking provides several benefits to the original renaming architecture:

- The LoadMark hints increases the performance of the original renaming architecture by filtering out stores which should not be put into the Store Address Cache. In addition, filtering out loads that do not benefit from renaming or value prediction will reduce the contention for the hardware predictor and increase prediction accuracy. This reduction can be significant, since only a small fraction of the total number of static stores and loads can benefit from renaming.
- The analysis can provide improved prediction accuracy by finding loads that are more predictable using last value prediction than renaming. We found several loads that fit into this category, even though they had aliased stores in the profile.
- The compiler can analyze how a load interacts with all the stores it may be dependent on, and intelligently choose only a portion of them for memory renaming. For example, we found on average that 20% of the loads are dependent upon more than 4 stores, but not all of these stores may provide accurate prediction using memory renaming.
- Since the LoadMark MRT architecture provides explicit tags for stores and loads to communicate, a load will benefit from renaming without having to wait for the hardware to discover the store/load relationship. This is very important for small tables, which will have a lot of replacements due to capacity misses.

7.2 Discussion

Using MRT tags with the load marking architecture in itself is not a complete solution for memory renaming. Hardware techniques are still needed, especially because of pre-compiled library routines and system calls which all programs use. Even so, LoadMarking could be used in concert with hardware renaming to improve the prediction accuracy by providing stream-lined communication via the MRT tag registers.

Using LoadMark hints to filter stores and loads has been shown to provide improved performance. Library routines and system calls do not pose a problem for using such hints. The saving and restoring of registers across system calls are good candidates for dynamic hardware renaming. A compiler that used LoadMark hints could easily take this into consideration by not filtering out stores and loads in the vicinity of calls to external routines.

8 Conclusions

As execution windows continue to grow in an effort to expose more instruction level parallelism, it becomes imperative to accurately identify store/load communication in order to exploit that parallelism. To this end, a great deal of research has been invested in devising means to disambiguate stores and loads, predict their values, or improve their communication. In this paper, we motivate and describe the Load Marking architecture; an approach that effectively classifies previous load speculation techniques. Our approach extends the instruction set to classify loads according to which prediction technique works best. By classifying loads, the Load Marking architecture is capable of choosing the best available predictor for each load. This flexibility provides accurate memory communication and value prediction.

This paper described and evaluated a design of the Load Marking architecture using detailed microarchitecture timing simulation. For a reasonably sized Value File, we found that an average of 34% of the dynamically executed instructions in our simulations used values produced by loads using LoadMark hints, with an overall prediction accuracy of 99%. Load Marking MRT allowed 37% of the dynamically executed instructions to consume predictions on average, with an overall prediction accuracy of nearly 99%. Since different inputs were used to guide the marking and to gather the results, this shows that the classification of loads was very predictable even between different inputs sets.

Simulation results yielded 6% speedups on average for using LM hints for squash recovery. These results are important since prior memory renaming studies showed little to no improvement for squash (re-fetch) recovery, which is much simpler and more feasible to implement than reexecution recovery. In addition, the LoadMark hints require only 3 new opcodes to provide the ISA hints. The LoadMark MRT architecture with tags was able to also provide a 6.1% speedup on average for squash recovery, with significant hardware savings.

The Load Marking architecture showed large speedups for a few programs – 13% for `tomcatv` and 12% for `su2cor`, both for squash recovery. When using smaller table sizes, the Load Marking architecture was still able to provide good speedups for both squash and re-execute recovery. Original rename prediction suffered from the smaller table sizes and provided significantly less speedup than with the 1024-entry table, especially for squash recovery.

The focus of this paper was profile-based classification of loads and identification of load-store relationships. In the absence of profile information, this information must be gained through loop carried and alias data dependence compiler analysis. In future work we will explore how well the compiler can classify loads and stores using only static analysis.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments. This work was funded in part by NSF CAREER grants No. CCR-9733278 and No. MIP-9701708, UC MICRO grant No. 97-018, and grant from Compaq Computer Corporation.

References

[1] 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.

[2] B. Calder, P. Feller, and A. Eustace. Value profiling. In *30th International Symposium on Microarchitecture*, pages 259–269, December 1997.

[3] B. Calder, P. Feller, and A. Eustace. Value prediction and optimization. *Journal of Instruction Level Parallelism*, 1(1), March 1999.

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

[5] T.M. Conte, K.N. Menezes, P.M. Mills, and B.A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333–344, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA.

[6] M. Franklin and G. S. Sohi. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[7] F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *30th International Symposium on Microarchitecture*, pages 270–280, December 1997.

[8] D.M. Gallagher, W.Y. Chen, S.A. Mahlke, J.C. Gyllenhaal, and W.W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Six International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, October 1994.

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

[10] D. Levitan, T. Thomas, and P. Tu. The powerpc 620 microprocessor: A high performance superscalar risc microprocessor. In *Proceedings of Spring CompCon*, pages 285–291, 1995.

[11] 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.

[12] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[13] Intel boosts pentium pro to 200 mhz. *Microprocessor Report*, 9(17), June 1995.

[14] A. Moshovos and G.S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *30th International Symposium on Microarchitecture*, pages 235–245, December 1997.

[15] M. Moudgill and J. H. Moreno. Run-time detection and recovery from incorrectly reordered memory operations. Technical Report RC 20857, IBM Research Report, May 1997.

[16] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *31st International Symposium on Microarchitecture*, December 1998.

[17] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 24–34, December 1996.

[18] Y. Sazeides and James E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, December 1997.

[19] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[20] 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.

[21] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, pages 281–290, December 1997.

A Profiling Details

This Appendix provides the description of the profiler we used to find the store/load relationships that the memory renaming architecture can take advantage of and accurately predict. Figure 7 shows the steps taken when profiling a store instruction. Each static store instruction is assigned a unique ID during instrumentation. When a store instruction is executed, its unique ID is used to access the corresponding store entry in the store array. The store entry structure has hash pointers so that the store entry can be inserted into and quickly found in a hash table, which is hashed by the current address of the store. The address, value, and current time stamp of the store are updated as shown in the figure. For each store, we only keep track of the last address seen by the store. If the current address of the store is different than its last address, the current address of the store is looked up in the hash table to find the Last Store that Referenced that Address (LSRA). The LSRA store's address is then invalidated, since we allow a given address (memory location) to be owned by at most one store a time.

Figure 8 shows the code sample used to profile load instructions. When a load is executed, its unique ID is used to access the corresponding load entry in the load array. The first part of the routine updates the prediction statistics described in section A.1. The rest of the profile routine updates the store/load relationship list, and updates the fields for last value, last address, time stamp, and number of times the load was executed. In order to update the relationship list, the address of the load is looked up in the store hash table to find a prior store with the same address. If a store/load relationship is found, then the corresponding store is recorded in the store list for the load unless this same load has been executed more recently than the store.

The store/load list finds all load/store dependencies, but it does not provide a complete picture of the effectiveness of renaming for a load. The renaming architecture described in section 3 will forward or predict the value of the *last* instruction to update the value table entry. When a load is encountered during profiling and finds a store relationship, the load-store dependence would be renamed correctly only if the store were the last instruction to update the value table entry. But another store sharing the same tag, or even this same load could have written to the value table more recently. To get an accurate picture on how effective renaming would be for a load we gather the Renamed Prediction metric, which models the prediction the load would see with the renaming and value prediction hardware.

A.1 Value Prediction and Rename Prediction Metrics

As seen in Figure 8, we record two value prediction metrics when profiling load instructions. The first metric is last value prediction, which predicts for a load the value of the load the last time it was executed. This is called the *load LVP*.

The other metric, called *Rename Prediction* (RP), provides the prediction accuracy of the load if it were to use renaming (in an infinite value table). This models the prediction that the hardware would provide if a single MRT tag was used for all stores in a load's store list. The RP accuracy of the load is determined by examining how many times the load's current value matched the value of the *most recent store* (MRS) alias executed, because the MRS would have potentially updated the value table entry last. If the load was more recently executed than the MRS, the correct rename prediction for a load would instead occur if the load's last value matched the current value of the load, because the prior load would have more recently updated the value table entry with its last value. The code to count the number of correct rename predictions is shown in Figure 8.

```
/* SE (store entry) is the current store's data structure */

/* LSRA is the last store instruction that referenced addr */

void profile_store (uniqueID, addr, value) {
  SE = store_array[uniqueID];

  if (SE->last_addr != addr) {
    /* Invalidate the address of store that last accessed addr */
    LSRA = store_lookup_hashtable_by_addr(addr);
    if (LSRA != NULL)
      LSRA->invalidate_address_and_hash_pointers();

    SE->last_addr = addr;
    SE->update_address_hash_pointers();
  }

  /* Update store stats */
  SE->last_value = value;
  SE->time_stamp = ++time;

}
```

Figure 7: Profile Code for Store Instructions

```
/* LE (load entry) is the data structure for current load */

/* MRS (most recent store) is the most recently executed store
  in the load's store list according to the time stamp */

/* SE (store entry) is the store that last referenced the address
  addr, if one exists. */

void profile_load (uniqueID, addr, value) {
  LE = load_array[uniqueID];

  /* Update Last Value Prediction metric */
  if (LE->last_value == value)
    LE->num_cor_LVP ++;

  /* Update Rename Prediction metric */
  MRS = LE->find_most_recent_store();
  if (MRS != NULL)
    if (LE->time_stamp > MRS->time_stamp) {
      /* Load would have last updated the value table entry */
      if (LE->last_value == value)
        num_cor_rename_pred ++;
    } else {
      /* MRS would have last updated the value table entry */
      if (MRS->last_value == value)
        num_cor_rename_pred ++;
    }

  /* Insert store/load relationship into store list for load */
  SE = store_lookup_hashtable_by_addr(addr);
  if ((SE != NULL) && (SE->time_stamp > LE->time_stamp))
    LE->update_store_relationship_list(SE);

  /* Update load stats */
  LE->last_addr = addr;
  LE->last_value = value;
  LE->time_stamp = ++time;
  LE->num_times_exe ++;
}
```

Figure 8: Profile Code for Load Instructions.