Improving the Accuracy and Performance of Memory Communication Through Renaming

Gary S. Tyson
The University of Michigan
tyson@eecs.umich.edu

Todd M. Austin
Intel Microcomputer Research Labs
taustin@ichips.intel.com

Abstract

As processors continue to exploit more instruction level parallelism, a greater demand is placed on reducing the effects of memory access latency. In this paper, we introduce a novel modification of the processor pipeline called **memory renaming**. *Memory renaming applies register access techniques to load instructions, reducing the effect of delays caused by the need to calculate effective addresses for the load and all preceding stores before the data can be fetched. Memory renaming allows the processor to speculatively fetch values when the producer of the data can be reliably determined without the need for an effective address. This work extends previous studies of data value and dependence speculation. When memory renaming is added to the processor pipeline, renaming can be applied to 30% to 50% of all memory references, translating to an overall improvement in execution time of up to 41%. Furthermore, this improvement is seen across all memory segments – including the heap segment which has often been difficult to manage efficiently.*

1 Introduction

Two trends in the design of microprocessors combine to place an increased burden on the implementation of the memory system: more aggressive, wider instruction issue and higher clock speeds. As more instructions are pushed through the pipeline per cycle, there is a proportionate increase in the processing of memory operations – which account for approximately 1/3 of all instructions. At the same time, the gap between processor and DRAM clock speeds has dramatically increased the latency of the memory operations. Caches have been universally adopted to reduce the average memory access latency. Aggressive out-of-order pipeline execution along with non-blocking cache designs have been employed to alleviate some of the remaining latency by bypassing stalled instructions.

Unfortunately, instruction reordering is complicated by the necessity of calculating effective addresses in memory operations. Whereas, dependencies between register-register instructions can be identified by examining the operand fields, memory dependencies cannot be determined until much later in the pipeline (when the effective address is calculated). As a result, mechanisms specific to loads and stores (e.g., the MOB in the Pentium Pro [1]) are required to resolve these memory dependencies later in the pipeline and enforce memory access semantics. To date, the only effective solution for dealing with ambiguous memory dependencies requires stalling loads until no earlier unknown store address exists. This approach is, however, overly conservative since many loads will stall awaiting the addresses of stores that they do not depend on, resulting in increased load instruction latency and reduced program performance. This paper proposes a technique called *memory renaming* that effectively predicts memory dependencies between store and load instructions, allowing the dynamic instruction scheduler to more accurately determine when loads should commence execution.

In addition to reordering independent memory references, further flexibility in developing an improved dynamic schedule can be achieved through our technique. Memory renaming enables load instructions to retrieve data before their own effective address have been calculated. This is achieved by identifying the relationship between the load and the previous store instruction that generated the data. A new mechanism is then employed which uses an identifier associated with the store-load pair to address the value, bypassing the normal addressing mechanism. The term memory renaming comes from the similarity this approach has to the abstraction of operand specifiers performed in *register renaming.* [7].

In this paper, we will examine the characteristics of the memory reference stream and propose a novel architectural modification to the pipeline to enable speculative execution of load instructions early in the pipeline (before address calculation). By doing this, true dependencies can be eliminated, in particular those true dependencies supporting the complex address calculations used to access the program data. This will be shown to have a significant impact on overall performance (as much as 41% speedup for the experiments presented).

The remainder of this paper is organized as follows: Section 2 examines previous approaches to speculating load instructions. Section 3 introduces the memory reordering approach to speculative load execution and evaluates the regularity of memory activity in order to identify the most successful strategy in executing loads speculatively. In section 4, we show one possible integration of memory renaming into an out-of-order pipeline implementation. Section 5 provides performance analysis for a cycle level simulation of the techniques. In section 6 we state conclusions and identify future research directions for this work.

2 Background

A number of studies have targeted the reduction of memory latency. Austin and Sohi [2] employed a simple, fast address calculation early in the pipeline to effectively hide the memory latency. This was achieved by targeting the simple *base+offset* addressing modes used in references to global and stack data.

Dahl and O'Keefe [5] incorporated address bits associated with each register to provide a hardware mechanism to disambiguate memory references dynamically. This allowed the compiler to be more aggressive in placing frequently referenced data in the register file (even when aliasing may be present), which can dramatically reduce the number of memory operations that must be executed.

Lipasti and Shen [8] described 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 *load value prediction unit* to hold the predicted value along with a *load classification table* for deciding whether the value is likely to be correct based on past performance of the predictor. They observed that a large number of load instructions are bringing in the same value time after time. By speculatively using the data value that was last loaded by this instruction before all dependencies are resolved, they are able to remove those dependencies from the critical path (when speculation was accurate). Using this approach they were able to achieve a speedup in execution of between 3% (for a simple implementation) to 16% (with infinite resources and perfect prediction).

Sazeides, Vassiliadis and Smith [10] used address speculation on load instructions to remove the dependency caused by the calculation of the effective address. This enables load instructions to proceed speculatively without their address operands when effective address computation for a particular load instruction remains constant (as in global variable references).

Finally, Moshovos, Breach, Vijaykumar and Sohi [9] used a memory reorder buffer incorporating data dependence speculation. Data dependence speculation allows load instructions to bypass preceding stores before ambiguous dependencies are resolved; this greatly increases the flexibility of the dynamic instruction scheduling to find memory instruction ready to execute. However, if the speculative bypass violates a true dependency between the load and store instructions in flight, the state of the machine must be restored to the point before the load instruction was mis-speculated and all instructions after the load must be aborted. To reduce the number of times a mis-speculation occurs, a prediction confidence circuit was included controlling when bypass was allowed. This confidence mechanism differs from that used in value prediction by locating dependencies between pairs of store and load instructions instead of basing the confidence on the history of the load instruction only. When reference prediction was added to the Multiscalar architecture, execution performance was improved by an average of 5-10%.

Our approach to speculation extends both value prediction and dependence prediction to perform targeted speculation of load instructions early in the architectural pipeline.

3 Renaming Memory Operations

Memory renaming is an extension to the processor pipeline designed to initiate load instructions as early as possible. It combines dependence prediction with value prediction to achieve greater performance than possible with either technique alone. The basic idea behind memory renaming is to make a load instruction look more like a register reference and thereby process the load in a similar manner. This is difficult to achieve because memory reference instructions, unlike the simple operand specifiers used to access a register, require an effective address calculations before dependence analysis can be performed. To eliminate the need to generate an effective address from the critical path for accessing load data, we perform a load speculatively, using the program counter (PC) of the load instruction as an index to retrieve the speculative value. This is similar to the approach used in Lipasti and Shen's value prediction except that in memory renaming this is performed indirectly; when a load instruction is encountered, the PC address associated with that load (LDPC) is used to index a dependence table (called the *store-load cache*) to determine the likely producer of the data (generally a store instruction). When it is recog-

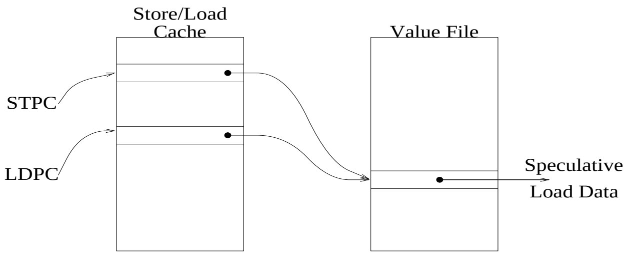

Figure 1: Support for Memory Renaming.

nized that the data referenced by a load instruction is likely produced by a single store instruction, the data from the last execution of that store can be retrieved from a *value file*. Accessing this value file entry can be performed (speculatively) without the need to know the effective address of the load or the store instruction, instead the value file is indexed by a unique identifier associated with the store-load pairing (this mechanism will be described in the next section). Store instructions also use the store-load cache to locate entries in the value file which are likely to be referenced by future loads. A store and load instruction pair which are determined to reference the same locations will map to the same value file index. Figure 1 shows an overview of the memory renaming mechanism.

This approach has the advantage that when a producer/consumer relationship exists, the load can proceed very early in the pipeline. The effective address calculation will proceed as usual, as will the memory operation to retrieve the value from the Dcache; this is done to validate the speculative value brought in from the value file. If the data loaded from the Dcache matches that from the value file, then speculation was successful and processing continues unfettered. If the values differ, then the state of the processor must be corrected.

In order for memory renaming to improve processor performance, it may be prudent to include a prediction confidence mechanism capable of identifying when speculation is warranted. As in value prediction and dependence prediction, we use a history based scheme to identify when speculation is likely to succeed. Unlike value renaming, we chose to identify stable store-load pairings instead of relying on static value references. To see the advantage of using a producer/consumer relationship as a confidence mechanism, analysis is shown for some of the SPEC95 benchmarks.

All programs were compiled with GNU GCC (version 2.6.2), GNU GAS (version 2.5), and GNU GLD (version 2.5) with maximum optimization (-O3) and loop unrolling enabled (-funroll-loops). The Fortran codes were first converted to C using AT&T F2C version 1994.09.27. All experiments were performed on an extended version of the SimpleScalar [3] tool set. The tool set employs the SimpleScalar instruction set, which is a (virtual) MIPS-like architecture [6].

Table 1: Benchmark Application Descriptions

Bench-mark	Instr (Mil.)	Loads (Mil.)	Value Locality	Addr Loc.	Prod Loc.
go	548	157	25 %	30 %	43 %
m88ksim	492	127	44 %	28 %	62 %
gcc	264	97	32 %	29 %	53 %
compress	3.5	1.3	15 %	37 %	50 %
li	956	454	24 %	23 %	55 %
perl	10	4.6	31 %	24 %	55 %
intAVE	378	149	29 %	29 %	53 %
tomcatv	2687	772	43 %	48 %	66 %
su2cor	1034	331	29 %	27 %	68 %
hydro2d	967	250	65 %	25 %	76 %
mgrid	4422	1625	42 %	30 %	75 %
fpAVE	2277	744	44 %	32 %	71 %

There are several approaches to improving the processing of memory operations by exploiting regularity in the reference stream. Regularity can be found in the stream of values loaded from memory, in the effective address calculations performed and in the dependence chains created between store and load instructions. Table 1 shows the regularity found in these differing characteristics of memory traffic. The first three column show the benchmark name, the total number of instructions executed and the total number of loads. The forth column shows the percentage of load executions of constant, or near constant values. The percentage shown is how often a load instruction fetches the same data value in two successive executions. (this is a measure of value locality). As shown in the table, a surprising number of load instruction executions bring in the same values as the last time, averaging 29% for SPEC integer benchmarks and 44% for SPECfp benchmarks. While it is surprising that so much regularity exists in value reuse, these percentages cover only about a third of all loads. Column 5 shows the percentage of load executions that reference the same effective address as last time. This shows about the same regularity in effective address reuse. The final column shows the percentage of time that the producer of the value remains unchanged over successive instances of a load instruction – this means that the same store instruction generated the data for the load. Here we see that this relationship is far more stable – even when the values transferred change, or when a different memory location is used for the transfer, the relationship between the sourcing store and the load remains stable. These statistics led us to the use of dependence pairings between store and load instructions to identify when speculation would be most profitable.

4 Experimental Pipeline Design

To support memory renaming, the pipeline must be extended to identify store/load communication pairs, promote their communications to the register communication infrastructure, verify speculatively forwarded values, and recover the pipeline if the speculative store/load forward was unsuccessful. In the following text, we detail the enhancements made to the baseline out-of-order issue processor pipeline. An overview of the extensions to the processor pipeline and load/store queue entries is shown in Figure 2.

4.1 Promoting Memory Communication to Registers

The memory dependence predictor is integrated into the front end of the processor pipeline. During decode, the store/load cache is probed (for both stores and loads) for the index of the value file entry assigned to the dependence edge. If the access hits in the store/load cache, the value file index returned is propagated to the rename stage. Otherwise, an entry is allocated in the store/load cache for the instruction. In addition, a value file entry is allocated, and the index of the entry is stored in the store/load cache. It may seem odd to allocate an entry for a load in the value file, however, we have found in all our simulations that this is a beneficial optimization; it promotes constants and rarely stored variables into the value file, permitting these accesses to also benefit from faster, more accurate communication and synchronization. In addition, the decode stage holds confidence counters for renamed loads. These counters are incremented for loads when their sourcing stores are predicted correctly, and decremented (or reset) when they are predicted incorrectly.

In the rename stage of the pipeline, loads use the value file index, passed in from the decode stage, to access an entry in the value file. The value file returns either the value last stored into the predicted dependence edge, or if the value is in the process of being computed (*i.e.* in flight), a load/store queue reservation station index is returned. If a reservation station index is returned, the load will stall until the sourcing store data is written to the store's reservation station. When a renamed load completes, it broadcasts its result to dependent instructions; the register and memory scheduler operate on the speculative load result as before without modification.

All loads, speculative or otherwise, access the memory system. When a renamed load's value returns from the memory system, it is compared to the predicted value. If the values do not match, a load data mis-speculation has occurred and pipeline recovery is initiated.

Unlike loads, store instructions do not access the value file until retirement. At that time, stores deposit their store data into the value file and the memory system. Later renamed loads that reference this value will be able to access it directly from the value file. No attempt is made to maintain coherence between the value file and main memory. If their contents diverge (due to, for example, external DMAs), the pipeline will continue to operate correctly. Any incoherence will be detected when the renamed load values are compared to the actual memory contents.

The initial binding between stores and loads is created when a load that was not renamed references the data produces by a renamed store. We explored two approaches to detecting these new dependence edges. The simplest approach looks for renamed stores that forward to loads in the load/store queue forwarding network (i.e., communications between instructions in flight). When these edges are detected, the store/load cache entry for loads is updated accordingly. A slightly more capable approach is to attach value file indices to renamed store data, and propagate these indices into the memory hierarchy. This approach performs better because it can detect longer-lived dependence edges, however, the extra storage for value file indices makes the approach more expensive.

4.2 Recovering from Mis-speculations

When a renamed load injects an incorrect value into the program computation, correct program execution requires that, minimally, all instructions that used the incorrect value and dependent instructions be re-executed. To this end, we explored two approaches to recovering the pipeline from data mis-speculations: *squash* and *re-execution* recovery. The two approaches exhibit varying cost and complexity - later we will see that lower cost mis-speculation recovery mechanisms enable higher levels of performance, since they permit the pipeline to promote more memory communication into the register infrastructure.

Squash recovery, while expensive in performance penalty, is the simplest approach to implement. The approach works by throwing away all instructions after a mis-speculated load instruction. Since all dependent instructions will follow the load instruction, the restriction that all dependent instructions be re-executed will indeed be met. Unfortunately, this approach can

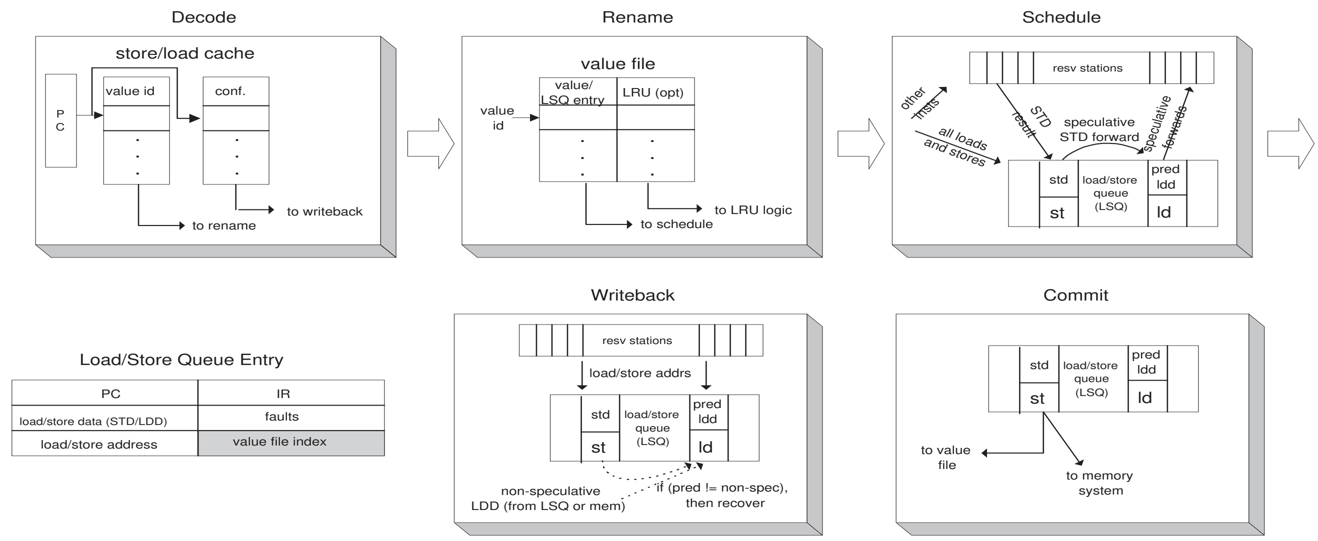

Figure 2: Pipeline Support for Memory Renaming. Shown are the additions made to the baseline pipeline to support memory renaming. The solid edges in the *writeback* stage represent forwarding through the reservation stations, the dashed lines represent forwarding through the load/store queue. Also shown are the fields added (shown in gray) to the instruction re-order buffer entries.

throw away many instructions independent of the mis-speculated load result, requiring many unnecessary re-executions. The advantage of this approach is that it requires very little support over what is implemented today. Mis-speculated loads may be treated the same as mis-speculated branches.

Re-execution recovery, while more complex, has significantly lower cost than squash recovery. The approach leverages dependence information stored in the reservation stations of not-yet retired instruction to permit re-execution of only those instructions dependent on a speculative load value. The cost of this approach is added pipeline complexity.

We implemented re-execution by injecting the correct result of mis-speculated loads onto the result bus - all dependent instructions receiving the correct load result will re-execute, and re-broadcast their results, forcing dependent instructions to re-execute, and so on. Since it's non-trivial for an instruction to know how many of its operands will be re-generated through re-execution, an instruction may possibly re-execute multiple times, once for every re-generated operand that arrives. In addition, dependencies through memory may require load instructions to re-execute. To accommodate these dependencies, the load/store queue also re-checks memory dependencies of any stores that re-execute, re-issuing any dependent load instructions. Additionally, loads may be forced to re-execute if they receive a new address via instruction re-execution. At retirement, any re-executed instruction will be the oldest instruction in the machine, thus it cannot receive any more re-generated values, and the instruction may be safely retired. In Section 5, we will demonstrate through simulation that re-execution is a much less expensive approach to implementing load mis-speculation recovery.

5 Experimental Evaluation

We evaluated the merits of our memory renaming designs by extending a detailed timing simulator to support the proposed designs and by examining the performance of programs running on the extended simulator. We varied the the confidence mechanism, mis-speculation recovery mechanism, and key system parameters to see what affect these parameters had on performance.

5.1 Methodology

Our baseline simulator is detailed in Table 2. It is from the SimpleScalar simulation suite (simulator *sim-outorder*) [3]. The simulator executes only user-level instructions, performing a detailed timing simulation of an 4-way superscalar microprocessor with two levels of instruction and data cache memory. The simulator implements an out-of-order issue execution model. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or mis-prediction. The model employs a 256 entry re-order buffer that implements renamed register storage and holds results of pending instructions. Loads and stores are placed into a 128 entry load/store queue. In the baseline simulator, stores execute when all operands are ready; their values, if speculative, are placed into the load/store queue. Loads may execute when all prior store addresses have been computed; their values come from a matching earlier store in the store queue (*i.e.*, a store forward) or from the data cache. Speculative loads may initiate cache misses if the address hits in the TLB. If the load is subsequently squashed, the cache miss will still complete. However, speculative TLB misses are not permitted. That is, if a speculative cache access misses in the TLB, instruction dispatch is stalled until the instruction that detected the TLB miss is squashed or committed. Each cycle the re-order buffer issues up to 8 ready instructions, and commits up to 8 results in-order to the architected register file. When stores are committed, the store value is written into the data cache. The data cache modeled is a four-ported 32k two-way set-associative non-blocking cache.

We found early on that instruction fetch bandwidth was a critical performance bottleneck. To mitigate this problem, we implemented a limited variant of the collapsing buffer described in [4]. Our implementation supports two predictions per cycle within the same instruction cache block, which provides significantly more instruction fetch bandwidth and better pipeline resource utilization.

When selecting benchmarks, we looked for programs with varying memory system performance, *i.e.*, programs with large and small data sets as well as high and low reference locality. We analyzed 10 programs from the SPEC'95 benchmark suite, 6 from the integer codes and 4 from the floating point suite.

All memory renaming experiments were performed with a 1024 entry, 2-way set associative store/load cache and a 512 entry value file with LRU replacement. To detect initial dependence edge bindings, we propagate the value file indices of renamed store data into the top-level data cache. When loads (that were not renamed) access renamed store data, the value file index stored in the data cache is used to update the load's store/load cache entry.[1]

5.2 Predictor Performance

Figure 3 shows the performance of the memory dependence predictor. The graph shows the hit rate of

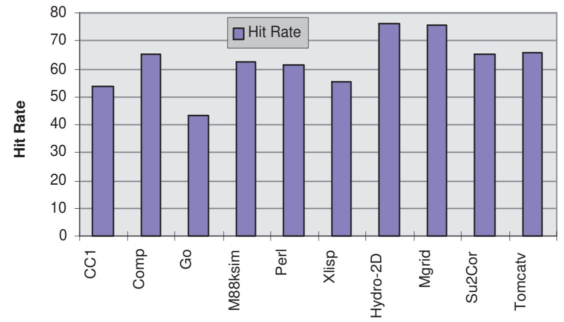

Figure 3: Memory Dependence Predictor Performance.

the memory dependence predictor for each benchmark, where the hit rate is computed as the number of loads whose sourcing store *value* was correctly identified after probing the value file. The predictor works quite well, predicting correctly as many as 76% of the program's memory dependencies - an average of 62% for all the programs. Unlike many of the value predictor mechanisms [8], dependence predictors work well, even better, on floating point programs.

To better understand where the dependence predictor was finding its dependence locality, we broke down the correct predictions by the segment in which the reference data resided. Figure 4 shows the breakdown of correct predictions for data residing in the global, stack, and heap segments. A large fraction of the correct dependence predictions, as much as 70% for *Mgrid* and 41% overall on the average, came from stack references. This result is not surprising considering the frequency of stack segment references and their semi-static nature, *i.e.*, loads and stores to the stack often reference the same variable many times. (Later we leverage this property to improve the performance of the confidence mechanisms.) Global accesses also account for many of the correct predictions, as much as 86% for *Tomcatv* and 43% overall on the average. Finally, a significant number of correct predictions come from the heap segment, as much as 40% for *Go* and 15% overall on the average. To better understand what aspects of the program resulted in these correct predictions, we profiled top loads and then examined their sourcing stores, we found a number of common cases where heap accesses exhibited dependence locality[2]:

[1]Due to space restrictions, we have omitted experiments which explore predictor performance sensitivity to structure sizes. The structure sizes selected eliminates most capacity problems in the predictor, allowing us to concentrate on how effectively we can leverage the predictions to improve program performance.

[2]These examples are typical of program constructs that challenge even the most sophisticated register allocators. As a result, only significant advances in compiler technology will eliminate these memory accesses. The same assertion holds for global ac-

Fetch Interface	fetches any 4 instructions in up to two cache block per cycle, separated by at most two branches
Instruction Cache	32k 2-way set-associative, 32 byte blocks, 6 cycle miss latency
Branch Predictor	8 bit global history indexing a 4096 entry pattern history table (GAp [11]) with 2-bit saturating counters, 8 cycle mis-prediction penalty
Out-of-Order Issue Mechanism	out-of-order issue of up to 8 operations per cycle, 256 entry re-order buffer, 128 entry load/store queue, loads may execute when all prior store addresses are known
Architected Registers	32 integer, 32 floating point
Functional Units	8-integer ALU, 4-load/store units, 4-FP adders, 1-integer MULT/DIV, 1-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, 6 cycle miss latency four-ported, non-blocking interface, supporting one outstanding miss per physical register 256k 4-way set-associative, unified L2 cache, 64 byte blocks, 32 cycle miss
Virtual Memory	4K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 2: Baseline Simulation Model.

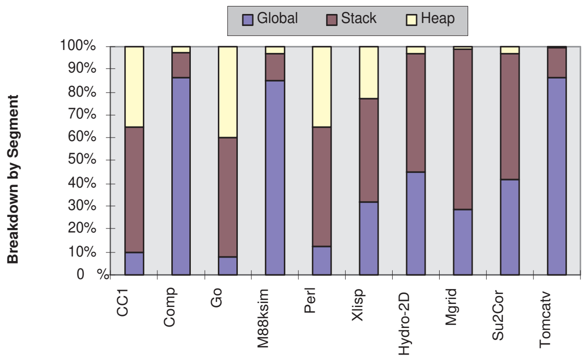

Figure 4: Predictor Hits by Memory Segment.

- repeated accesses to aliased data, which cannot be allocated to registers
- accesses to loop data with a loop dependence distance of one[3]
- accesses to single-instance dynamic storage, *e.g.*, a variable allocated at the beginning of the program, pointed to by a few, immutable global pointers

As discussed in Section 3, a pipeline implementation can also benefit from a confidence mechanism. Figure 5 shows the results of experiments exploring the efficacy of attaching confidence counters to load instructions. The graphs show the confidence and coverage for a number of predictors. *Confidence* is the success rate of high-confidence loads. *Coverage* is the fraction of correctly predicted loads, without confidence, covered by the high-confidence predictions of a particular predictor. Confidence and coverage are shown for 6 predictors. The notation used for each is as follows: XYZ, where X is the count that must be reached before the predictor considers the load a high-confidence load. By default, the count is incremented by one when the predictor correctly predicts the sourcing store value, and reset to zero when the predictor fails. Y is the count increment used when the opcode of the load indicates an access off the stack pointer. Z is the count increment used when the opcode of the load indicates an access off the global pointer. Our analyses showed that stack and global accesses are well behaved, thus we can increase coverage, without sacrificing much confidence, by incrementing their confidence counters with a value greater than one.

As shown in Figure 5, confidence is very high for the configurations examined, as much as 99.02% for *Hydro-2D* and at least 69.22% for all experiments that use confidence mechanisms. For most of the experiments we tried, increasing the increments for stack and global accesses to half the confidence counter performed best. While this configuration usually degrades confidence over the baseline case (an increment of one for all accesses), coverage is improved enough to improve program performance. Coverage varies significantly, a number of the programs, *e.g.*, *Compress* and *Hydro-2D*, have very high coverage, while others, such as *CC1* and *Perl* do not gain higher coverage until a significant amount of confidence is sacrificed. Another interesting feature of our confidence measurements is the relative insensitivity of coverage to the counter threshold once the confidence thresholds levels rise above 2. This rein-

cesses, all of which the compiler must assume are aliased. Stack accesses on the other hand, can be effectively register allocated, thereby eliminating the memory accesses, given that the processor has enough registers.

[3] Note that since we always predict the sourcing store to be that last previous one, our predictors will not work with loop dependence distances greater than one, even if they are regular accesses. Support for these cases are currently under investigation.

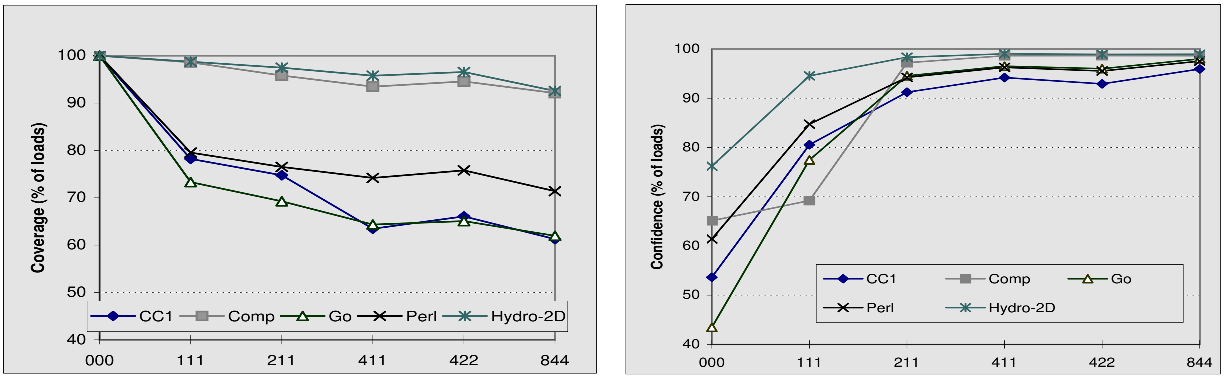

Figure 5: Confidence and Coverage for Predictors with Confidence Counters.

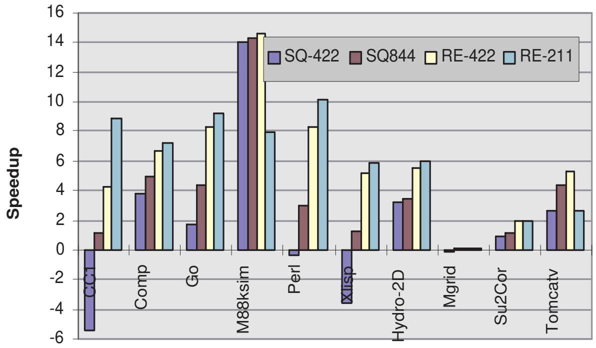

Figure 6: Program Performance with Varied Predictor/Recovery Configuration.

forces our earlier observation that the memory dependencies in a program are relatively static – once they occur a few times, they very often occur in the same fashion for much of the program execution.

5.3 Pipeline Performance

Predictor hit rates are an insufficient tool for evaluating the usefulness of a memory dependence predictor. In order to fully evaluate it, we must integrate it into a modern processor pipeline, leverage the predictions it produces, and correctly handle the cases when the predictor fails. Figure 6 details the performance of the memory dependence predictor integrated into the baseline out-of-order issue performance simulator. For each experiment, the figure shows the speedup (in percent, measured in cycles to execute the entire program) with respect to the baseline simulator.

Four experiments are shown for each benchmark in Figure 6. The first experiment, labeled *SQ-422*, shows the speedup found with a dependence predictor utilizing a 422 confidence configuration and squash recovery for load mis-speculations. Experiment *SQ-844* is the same experiment except with a 844 confidence mechanism. The *RE-422* configuration employs a 422 confidence configuration and utilizes the re-execution mechanism described in Section 3 to recover from load mis-speculations. Finally, the RE-211 configuration also employs the re-execution recovery mechanism, but utilizes a lower-confidence 211 confidence configuration.

The configuration with squash recovery and the 422 confidence mechanism, *i.e.*, SQ-422, shows small speedups for many of the programs, and falls short on others, such as *CC1* which saw a slowdown of more than 5%. A little investigations of these slowdowns quickly revealed that the high-cost of squash recovery, *i.e.*, throwing away all instructions after the mis-speculated load, often completely outweighs the benefits of memory renaming. (Many of the programs had more data mis-speculations than branch mis-predictions!) One remedy to the high-cost of mis-speculation is to permit renaming only for higher confidence loads. The experiment labeled *SQ-844* renames higher-confidence loads. This configuration performs better because it suffers from less mis-speculation, however, some experiments, *e.g.*, *CC1*, show very little speedup because they are still plagued with many high-cost load mis-speculations.

A better remedy for high mis-speculation recovery costs is a lower cost mis-speculation recovery mechanism. The experiment labeled *RE-422* adds re-execution support to a pipeline with memory renaming support with a 422 confidence mechanism. This design has lower mis-speculation costs, allowing it to to show speedups for all the experiments run, as much as 14% for *M88ksim* and an average overall speedup of over 6%. To confirm our intuitions as to the lower cost of re-execution, we measured directly the cost of squash recovery and re-execution for all runs by counting the number of instructions thrown away due to load mis-speculations. We found that overall, re-execution consumes less than 1/3 of the execution bandwidth required by squash recovery – in other words, less than 1/3 of the instructions in flight after a load mis-speculation are dependent on the mis-speculated load, on average. Additionally, re-execution benefits from not having to re-fetch, decode, and issue instructions after the mis-speculated load.

Given the lower cost of cost of re-execution, we explored whether speedups would be improved if we also renamed lower-confidence loads. The experiment labeled *RE-211* employs re-execution recovery with a lower-confidence 211 confidence configuration. This configuration found better performance for most of the experiments, further supporting the benefits of re-execution. We also explored the use of yet even lower-confidence (111) and no-confidence (000) configurations, however, mis-speculation rates rise quickly for these configurations, and performance suffered accordingly for most experiments.

Figure 7 takes our best-performing configuration, i.e., *RE-211*, and varies two key system parameters to see their effect on the efficacy of memory renaming. The first experiment, labeled *FE/2*, cuts the peak instruction delivery bandwidth of the fetch stage in half. This configuration can only deliver up to four instructions from one basic block per cycle. For many of the experiments, this cuts the average instruction delivery B/W by nearly half. As shown in the results, the effects of memory renaming are severely attenuated. With half of the instruction delivery bandwidth the machine becomes fetch bottlenecked for many of the experiments. Once fetch bottlenecked, improving execution performance with memory renaming does little to improve the performance of the program. This is especially true for the integer codes where fetch bandwidth is very limited due to many small basic blocks.

The second experiment in Figure 7, labeled *SF*3*, increases the store forward latency three-fold to three cycles. The store forward latency is the minimum latency, in cycles, between any two operations that communicate a value to each other through memory. In the baseline experiments of Figure 6, the minimum store forward latency is one cycle. As shown in the graph, performance improvements due to renaming rise sharply, to as much as 41% for *M88ksim* and more than 16% overall. This sharp rise is due to the increased latency for communication through memory – this latency must be tolerated, which consumes precious parallelism. Renamed memory accesses, however, may communication through the register file in potentially zero cycles (via bypass), resulting in significantly lower communication latencies. Given the complexity of load/store queue dataflow analysis and the requirement that it be performed in one cycle for one-cycle store forwards (since addresses in the computation may arrive in the previous cycle), designers may soon resort to larger load/store queues with longer latency store forwards. This trend will make memory renaming more attractive.

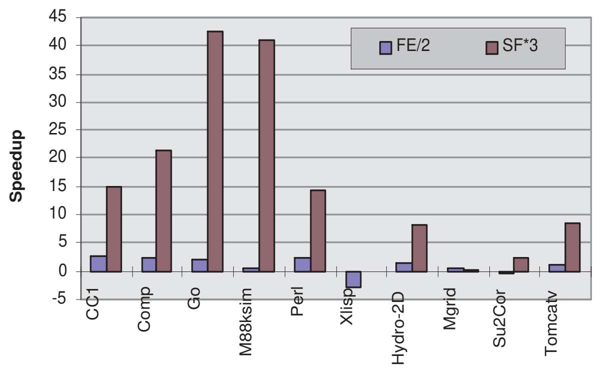

Figure 7: Program Performance with Varied System Configuration.

A fitting conclusion to our evaluation is to grade ourselves against the goal set forth in the beginning of this paper: build a renaming mechanism that maps all memory communication to the register communication and synchronization infrastructure. It is through this hoisting of the memory communication into the registers that permits more accurate and faster memory communication. To see how successful we were at this goal, we measured the breakdown of communication handled by the load/store queue and the data cache. Memory communications handled by the load/store queue are handled "in flight", thus this communication can benefit from renaming. How did we do? Figure 8 shows for each benchmark, the fraction of references serviced by the load/store queue in the base configuration, labeled *Base*, and the fraction of the references serviced by the load/store queue in the pipeline with renaming support, labeled *RE-422*. As shown in the figure, a significant amount of the communication is now being handled by the register communication infrastructure. Clearly, much of the short-term communication is able to benefit from the renamer support.

However, a number of the benchmarks, *e.g.*, *CC1*, *Xlisp* and *Tomcatv*, still have a non-trivial amount of

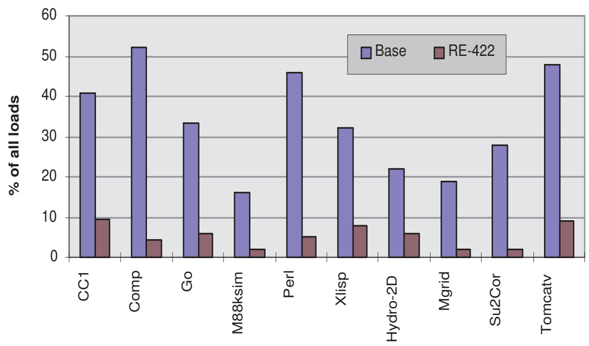

Figure 8: Percent of Memory Dependencies Serviced by Load/Store Queue.

short-term communication that was not identified by the dependence predictor. For these programs, the execution benefits from the load/store queues ability to quickly compute load/store dependencies once addresses are available. One goal of this work should be to improve the performance of the dependence predictor until virtually all short-term communication is captured in the high-confidence predictions. Not only will this continue to improve the performance of memory communication, but once this goal has been attained, the performance of the load/store queue will become less important to overall program performance. As a result, less resources will have to be devoted to load/store queue design and implementation.

6 Conclusions

In this paper we have described a new mechanism designed to improve memory performance. This was accomplished by restructuring the processor pipeline to incorporate a speculative value and dependence predictor to enable load instructions to proceed much earlier in the pipeline. We introduce a prediction confidence mechanism based on store-load dependence history to control speculation and a value file containing load and store data which can be efficiently accessed without performing complex address calculations. Simulation results validate this approach to improving memory performance, showing an average application speedup of 16%.

We intend to extend this study in a number of ways. The most obvious extension of this work is to identify new mechanisms to improve the confidence mechanism and increase the applicability of this scheme to more load instructions. To do this we are exploring integrating control flow information into the confidence mechanism. Another architectural modification is to improve the efficiency of squashing instructions effected by a mis-prediction. This is starting to become important in branch prediction, but becomes more important in value prediction because of the lower confidence in this mechanism. Also, the number of instructions that are directly effected by a misprediction in a load value is less than for a branch prediction allowing greater benefit from a improvement in identifying only those instructions that need to be squashed.

7 Acknowledgments

Finally, we would like to acknowledge the help of Haitham Akkary, who offered numerous suggestions which have greatly improved the quality of this work. We are also grateful to the Intel Corporation for its support of this research through the Intel Technology for Education 2000 grant.

References

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

[2] Todd M. Austin and Guri S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 82–92, November 1995.

[3] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: The simplescalar tool set. In *UW – Madison Technical Report #1308*, Jul 1996.

[4] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *22nd Annual International Symposium on Computer Architecture*, pages 333–344, Jun 1995.

[5] Peter Dahl and Matthew O'Keefe. Reducing memory traffic with cregs. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 100–111, November 1994.

[6] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[7] Robert Keller. Look-ahead processors. In *ACM Computing Surveys*, pages 177–195, December 1975.

[8] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[9] Andreas Moshovos, Scott Breach, T. N. Vijaykumar, and Gurindar Sohi. Dynamic speculation and synchronization of data dependences. In *24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.

[10] Yiannakis Sazeidis, Stamatis Vassiliadis, and James Smith. The performance potential of data dependence speculation and collapsing. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 238–247, November 1996.

[11] Tse-Yu Yeh and Yale Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, November 1991.