A Comparison of the Benefits of Dependence Prediction and Value Prediction

Abhijit Paithankar
Dirk Grunwald

CU-CS-868-98

A Comparison of the Benefits of Dependence Prediction and Value Prediction

Abhijit Paithankar, Dirk Grunwald
[pvega,grunwald]@cs.colorado.edu

University of Colorado, Department of Computer Science
Boulder, CO 80309-0430

Abstract

This paper compares the efficacy of different load value prediction and dependence prediction mechanisms on super-scalar, out-of-order processors. We consider two load value prediction methods and two dependence prediction methods. We also compare the effect of two recovery mechanisms.

Our conclusions are that dependence predictors are very cost effective, greatly increasing performance. By comparison, current value predictors add little performance above that of dependence predictors; in fact, value predictors hurt performance when using a "re-fetch" recovery mechanism. The experimental comparisons are done using a cycle-level simulator, and we use that simulator to explain the results.

1 Introduction

Before a memory operation can complete, a processor must determine the address that is being requested, check for possible data dependences and issue the operation for that memory address. The processor must insure that all data dependence constraints are met; this is usually enforced by not executing a load instruction until the memory addresses used by all prior store instructions are known.

A speculative processor can mask the latency of these operations by speculating on the outcome of various decisions. Load value prediction [10] speculates on the final outcome of a memory load, allowing subsequent operations to continue immediately. Dependence prediction allows the processor to speculate on the outcome of memory data dependences, allowing memory loads to issue before the addresses of outstanding stores are known. Load value prediction and dependence prediction serve similar ends – allowing operations dependent on the load data to issue earlier, increasing the instruction level parallelism. Dependence prediction allows subsequent instructions to start earlier because the load operation can execute when the memory address is known; there is no need to wait for pending stores to resolve their addresses. Like dependence prediction, value prediction can eliminate the delay caused by address resolution, but it can also eliminate the delay of accessing memory.

With both methods, all load operations must actually execute to insure that the speculative operations were predicted correctly. Both value prediction and dependence prediction use a recovery mechanism to correct the processor state when a prediction is shown to be incorrect. Dependence and value prediction can slow program execution if they make poor predictions because instructions may be re-executed several times.

There has been considerable investigation into value prediction [10, 19, 4, 15]. Most of these studies compare value predictors by their accuracy and coverage. Lipasti *et al* [10] uses a performance model to compare the instructions per cycle (IPC) of a processor using value prediction to one without value prediction; however, the simulated processor did not use dependence prediction.

In this paper, we compare value and dependence prediction using a super-scalar, out-of-order performance model. Our conclusions are that dependence prediction is at more effective than value prediction for current processors. We believe value prediction is considerably more difficult to implement efficiently than dependence prediction, and large improvements in value predictor performance will be required to justify the cost of value prediction.

In the next section, we describe dependence prediction and value prediction using concrete examples and describe common implementations. Following that, we describe the simulation model we use and the assumptions we make in our experimental design. We then describe the results of our experiments.

Issue #	PC	Insn	Addr	Value
1	0x100	ST	X	10
2	0x100	ST	X	10
3	0x100	ST	Y	30
4	0x200	ST	X	40
5	0x300	ST	Z	50
6	0x400	LD	X	
7	0x500	LD	W	

Figure 1: Sample program execution highlighting the problems being solved by value and dependence prediction

2 Background and Related Work

We will use the simple example in Figure 1 to highlight the problems that dependence and value prediction are trying to solve. In this figure, we show the execution of seven instructions. Each instruction has an "issue number", which defines the issue order of the instructions. Each issued instruction has an associated program counter address – issue numbers 1-3 were generated by the same instruction in the program. Likewise, each instruction has an associated address and value that can only be determined during program execution.

Using conventional memory ordering semantics, load instructions must load the most recent value generated by all previous store instructions. A load is dependent on a store if both instructions reference the data memory address *and* there are no intervening stores to the same location. For example, the load of "X" at issue six should result in the value stored by issue four. The load of issue "W" is not dependent on any instructions shown in this example. Issue four is *not* dependent on issues 1 or 2 since those results are older than the more recent issue 4.

Although instructions issue in a specific order, we assume they may execute in a different order. The microarchitecture is responsible for insuring the execution order obeys all data dependence relations. Store instructions can be reordered by the use of a "store buffer" – when a store instruction is issued, it is allocated a store buffer entry. When the store executes, it writes the resulting operand to that store buffer entry; when (or if) the store is retired, the entry is actually written to memory. This means load instructions have access to the store results of all previous non-committed stores *via* the store buffer.

2.1 Dependence Prediction

Since instructions can execute out-of-order, a load instruction may have resolved the address for that load before the address for all stores is known. This means the load must wait to determine if there is an outstanding memory dependence that should be observed. In a *conservative dependence model*, load instructions can not execute until the addresses are known for *all* previously issued store instructions. In practice, most load instructions *do not* depend on previous, unresolved store instructions, and this conservative model greatly reduces the possible IPC.

In a speculative dependence model, a load is assumed to have *no dependences* on unresolved stores. Using this model, load instructions would issue and access the cache (or store buffer) as soon as possible. However, the load must check the address of each outstanding store as it is resolved, and then *trap and replay* the load instruction if a conflict is detected. For example, assume the instruction execution order for our example is 2, 6, 4. The store at issue 2 is executed and the value is written to the store buffer. The load at issue 6 is executed, and the value from issue 2 is used by subsequent instructions. Finally, the store at issue 4 executes. All load instructions following issue 4 are checked to see if the store address matches their load address; if so, that load instruction, and any instruction using the value from that load, must be re-executed. A more sophisticated mechanism could compare both the address being generated *and* the value being stored. If the load actually loaded the correct value (from the wrong store instruction), there is no need to trap and replay the store.

The mechanism we described is a fully, or naive, speculative dependence model. Various mechanisms, described later, have been proposed that attempt to predict the dependence relationships and issue loads only if it's unlikely that they depend on unresolved store instructions.

2.2 Load Value Prediction

Load value prediction (LVP) attempts to further reduce the delay encountered by loads by predicting the *value* of the load instruction. Instructions dependent on that load can thus execute immediately, before the load address is resolved or the cache is accessed. As with dependence prediction, the processor must still perform the actual load operation and must supply a mechanism to trap and replay program execution following a mispredicted load value. If the value of the load matches the predicted value, execution continue unabated; otherwise, the processor traps execution.

2.3 Trap Mechanisms

There are two mechanisms to replay load instructions. The first, which we call *re-fetch*, behaves much like a branch misprediction – all instructions including and following the conflicting load instruction are discarded, program state is backed up to that load instruction and execution is resumed. The refetched instruction will now use the value from the correct store. This policy is wasteful of fetch and execution bandwidth because it may discard the results of many executed instructions that are not dependent on the incorrectly speculated load instruction. Alternatively, the processor may use a *re-execute* policy. In this policy, the load instruction and any instruction dependent on that load are simply marked as "not executed" and are re-executed. The instruction window is not cleared, and instructions that do not depend on the load need not be re-executed.

Note that all load instructions must eventually execute, even with a perfect load dependence or LVP predictor. Dependence prediction and LVP improve the program ILP by allowing the load to execute earlier; it does not reduce the number of memory accesses. With dependence prediction, instructions dependent on the load still can not execute until the address for the load is resolved and the cache access is finished. With value prediction, those instructions can execute immediately.

Dependence prediction requires that the processor record memory addresses and has a fast mechanism to compare addresses against pending loads and stores when addresses are computed. This will typically involve an associative search (i.e., CAMs) in the load-store buffer. A processor using LVP must check the *value* being used. A processor using LVP and conservative dependence prediction may not need the expensive CAM's used by dependence prediction. A processor using LVP and dependence prediction would need both mechanisms.

3 Hardware Mechanisms

Several papers have proposed different memory dependence prediction or recovery mechanisms [6, 3, 16, 12, 2, 8]. Memory renaming [18] and memory streamlining [13] attempt to predict store-load pairs that have memory data dependencies. However, these mechanisms further reduce the memory delay by forwarding results from the store operation to the consuming load and in certain cases can reduce memory accesses. Likewise, there has been considerable work on value prediction [10, 10, 19, 4] including load value prediction.

We chose to focus on two dependence prediction mechanisms and two load value prediction mechanisms. We implemented the *Load Wait Table* (LWT) [8] because it has been implemented in a current, aggressive out-of-order system. We also implemented the *Store Sets* (SS) mechanism [2] because its performance is so close to perfect that it seems unlikely that a better dependence prediction mechanism is possible. We do not consider mechanisms such as memory renaming or streamlining in this paper. We also implemented the *last value predictor* (LV) [10] and the *stride value predictor* (Stride) [4]. We chose the LV predictor because it was the first value predictor proposed and most papers compare their results to that predictor and it may be possible to extrapolate the results of this study to other studies. We implemented the stride predictor because it uses twice the state state but has a noticeable improvement over the simpler LV predictor.

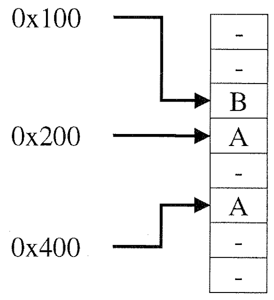

Figure 2: Diagram showing conceptual operation of the store sets mechanism

Tyson *et al* [18] compare trap mechanisms that refetch or re-execute instructions upon detecting mis-speculations, but in the context of memory renaming. Most other papers appear to assume a refetch mechanism. We implement both techniques and compare their effectiveness. We assume that the re-execution mechanism can locate all instructions that depend directly or indirectly on a mis-speculated instruction in a single cycle. This is done using a check-and-propagate circuit that starts with a bitmask representing the mis-speculated register. All instructions compare against the current bitmask and, if they depend on any register set within the bitmask, assert the bit corresponding to their output register. This circuit is similar to the arbitration logic that appears in many processors, but it's probably unrealistic to assume that it could be performed in a single cycle on a processor with a very large instruction window. Thus, our results for processors using a re-execution mechanism are optimistic.

3.1 Load-Wait Tables (LWT)

The DEC Alpha 21264 processor [8] uses load-wait tables to control speculative dependence prediction. The LWT is a table of one-bit registers indexed by the program counter at instruction fetch. Initially, load instructions are executed speculatively. If a load instruction causes a trap, the bit corresponding to that load in the LWT is set. The LWT values are used in the decode phase; if the value in the table is set and the instruction is a load, it is assumed that the load instruction will cause a trap and it is not issued speculatively. The LWT is periodically cleared; this ages the entries in the LWT as the processor moves between programs or regions within a program and reduces the number of false dependences encountered by the program. Our implementation of the LWT mechanism attempts to follow the implementation described in [8]. Since that paper did not mention the period when the LWT is cleared, we tried a number of variants. In our final implementation of the LWT mechanism, we augmented each LWT entry with an 8-bit countdown counter that was set to 255 when the LWT entry was set and is decremented each cycle. When the counter reached zero, the LWT entry was cleared. This had better performance than clearing the entire table after a fixed period.

The LWT exercises a coarse-grained control over the memory references. Consider the sample code fragment in Figure 1. If the LWT corresponding to address "0x400" is set, the instruction at issue 6 will wait for *all* store instructions (issue 1-5) to complete. No distinction is made about what store instruction causes the memory dependence.

3.2 Store Sets (SS)

Store sets [2] can be viewed as an extension of the LWT that tries to determine the loads and stores that tend to cause dependence violations. All load and store instructions index a *store set identification table* (SSIT), shown schematically in Figure 2. This is a table that (logically) assigns each load or store to one of a number of "sets" of related loads and stores. Initially, all entries are set to indicate there is no dependence, allowing speculative dependence prediction to occur. When a load dependence is mis-predicted, the store instruction determines the index using the address of the conflicting load. If the SSIT entry for that store has already been allocated a store set identifier, then the SSIT entry corresponding to the load is set to the same identifier. Otherwise, a new identifier is allocated and assigned to both entries in the SSIT.

The SSIT is accessed to determine the store set for operations as memory operations are decoded. Load instructions are then dependent *only on preceding store instructions within the same store set.* Figure 2 illustrates the SSIT following execution of the code fragment in Figure 1, assuming that issue six had executed prior to issue four. In subsequent executions, the load at address address "0x400" will be allowed to execute ahead of store instructions at address "0x100", *but not* ahead of store instructions from address "0x200".

In practice, store sets are a little more complicated, but the implementation is fairly simple. The SSIT is essentially a direct-mapped table of store-set identifiers, represented as an n-bit number. Identifiers are computed using a hash of the load address, introducing another opportunity for false dependences. Stores can belong to multiple store-sets using a heuristic described in [2], and stores within a store set must execute in issue order. The size of the table (*i.e.*, the number of entries in the SSIT) is independent of the number of store sets. In practice, 4092 entries holding one of 128 possible store sets provides very good performance [2] at the expense of 28Kbits of storage.

The implementation presented in [2] is slightly more complicated than necessary because the authors were attempting to use direct-mapped data structures, rather than associative structures, wherever possible. This opens the possibility that multiple (unrelated) stores can appear to belong to same store set, causing those stores to execute in order. Some of these constraints were imposed by the instruction issue window model assumed by the authors. There are a number of ways to reset the information in the SSIT; the method proposed in [2] simply clears the SSIT periodically.

We implemented essentially the same mechanism; however, our processor model uses a load-store queue to avoid write-after-write hazards and we did not include the restriction that stores within the store set execute in program order. In our implementation, we used a 1024-entry SSIT. Dependent load instructions recorded the program counter address of the store causing that dependence, and each store could belong to two different store sets. We found little performance difference between our implementation and the implementation proposed in [2], and each of the different implementations would be appropriate for different microarchitectures. Despite these minor differences, comparisons between the performance of our implementation and the one proposed in [2] shows that both have similar behavior when compared to a "perfect" memory dependence predictor.

3.3 Last Value Predictor (LV)

A simple scheme for predicting load values is simply to store the value produced by a load instruction when it is executed, and predict that value the next time that load instruction is seen. This is the approach used in [10]. Our implementation uses a untagged table of values indexed by the address of load instructions. Associated with each entry in the table is a 2-bit history counter that indicates the predictability of that value based on its recent dynamic behavior. A prediction is made only if the same value resulted from the two most recent executions of the load instruction (or loads mapping to the same line in the table). If a different value is seen, it replaces the old value in the entry corresponding to the load address, and resets the history counter to zero.

3.4 Stride Value Predictor (Stride)

We use a stride predictor as described in [19]. It seeks to exploit the repetitive nature of the *change* in data values produces by successive executions of a load instruction. The stride predictor consists of a table indexed by load instruction addresses. Each entry in the table contains a tag, a last value, a last stride, and a history counter to indicate how many times the current stride value was produced by the most recent execution(s) of the load instruction that maps to the entry.

A prediction is made if the same stride is seen at least twice consecutively. If the prediction fails, a new stride is computed and the history counter is set to 1. If a new load instruction is assigned to the table entry, then the history counter resets to zero. It is advanced to 1 when that load instruction is executed the next time, and the new value and the stride (the difference between the new and the old values) are written into the entry.

4 Experimental Methodology and System Configuration

We use a detailed pipeline-level execution-driven architecture simulator to model an out-of-order speculative superscalar processor similar to the DEC Alpha 21264 processor [8]. The simulator was constructed using the AINT [14] execution-driven simulation environment and models the true behavior of the micro-architecture including execution of instructions along speculative paths such as those following a mispredicted branch, dependence prediction or value prediction.

We used the SPEC95 benchmark suite to evaluate the performance of the simulated architectures. The programs were compiled using the `-O5 -g3` optimization flags and statically linked using the native compilers. Under Digital Unix, statically linking the binary enables a link-time optimization that unifies the different global object tables (GOT), considerably reducing procedure calling overhead and the number of run-time constants loaded by the programs [17]. The *ref* data sets were used for all applications except `compress`, for which we used an input size smaller than *ref* but larger than *train*[1]. We plotted the cache locality and branch predictability of the programs to determine "interesting" regions of execution. We then used a "fast execution" mode to skip the program initialization and then simulate the beginning of the interesting region. Table 1 shows the number of instructions skipped for each program; we skipped between 200 million and 2 billion instructions. We then switched to detailed microarchitectural simulations and measured 200 million instructions for each of the SPEC 95 benchmark programs.

We simulate a speculative out-of-order superscalar processor as shown in figure 3. The fetch unit fetches up to two consecutive cache blocks from the instruction cache each cycle. Other studies, such as [2], assume the processor can fetch non-consecutive blocks in the instruction cache, including the target of taken branches.

[1]With the *ref* input size, one complete iteration of compress takes approximately 2 billion Alpha instructions. We used a smaller input size to be able to practically simulate one complete iteration within a reasonable amount of time

Application	Instructions (millions)		Loads	Load Miss	Branch Mispred.
	Skipped	Exec'd	(millions)	Rate (%)	Rate (%)
compress	200	200	35.64	16.05	8.29
gcc	200	200	45.02	6.15	6.1
go	200	200	48.26	0.17	16.62
ijpeg	200	200	33.83	5.75	10.61
li	200	200	48.59	18.98	4.3
m88ksim	300	200	41.58	0.13	3.98
perl	700	200	55.37	0.12	2.84
vortex	200	200	45.78	0.69	1.52
applu	200	200	45.86	23.01	1.28
apsi	200	200	52.10	11.85	1.69
fpppp	200	200	62.92	0.01	4.43
hydro2d	200	200	47.12	47.16	0.26
mgrid	200	200	86.81	12.03	1.59
su2cor	2,000	200	36.35	33.71	13.26
swim	200	200	54.40	25.31	0.26
tomcatv	2,000	200	48.72	30.02	0.58
turb3d	100	200	45.70	1.09	2.53
wave5	1,200	200	51.68	18.10	0.56

Table 1: Applications simulated, including the number of instructions skipped and measured. The "loads" column shows the number of loads in the portion of the application that was measured.

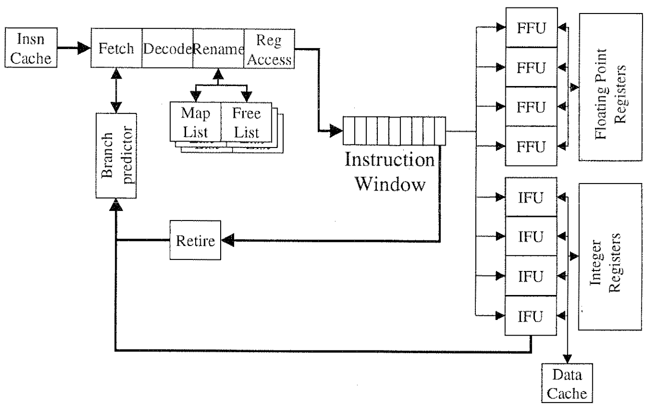

Figure 3: Baseline architecture model with an in-order front-end, out-of-order execution phase and in-order commit

Type	Latency (cycles)
Integer multiply	8-14
Int conditional move	2
Other int & logical	1
Floating point Multiply	4
FP Divide	16
Other FP	4
L1 DCache read	1
L2 load-to-use	12
Memory load-to-use	80

Table 2: Latencies of instruction execution and memory references

This results in a higher IPC than what we achieve. Instructions are placed in the fetch queue until the first branch that is predicted to be taken is encountered or there are no more instructions available. We use a combining branch predictor as described by McFarling [11], consisting of a bimodal predictor indexed by branch address and a gshare predictor indexed by the branch address and global history. The meta-predictor is indexed by the branch address.

The issue logic closely models that of the Alpha 21264, with the addition of a central instruction window. Instructions are decoded and entered into a in-order instruction buffer. The decode-rename unit determines the data dependencies between instructions. The selection logic selects instructions to execute when their inputs are ready. Instructions stay in the instruction buffer until they are ready to commit and instructions are committed in program-order. We used the central window mechanism in order to compare the effect of "re-fetch" and "re-execute" recovery mechanisms. The stages in the execution of a typical instruction are shown in figure 3. Memory instructions (loads and stores) have extra stages for performing the cache reference, in addition to the execution stage that computes the effective address of the memory reference. The execution latencies of the instructions are similar to those of the DEC Alpha 21264 processor, and are listed in table 2.

We examine the impact of the store-load dependence policy on an 8-way superscalar with a 512-entry instruction window. The processor can fetch eight instructions per cycle and complete 18 instructions from the ten integer units and eight floating point units. The processor has 128kB 4-way associative level-1 instruction and data caches and an 8MB 4-way associative level-2 unified cache. The branch predictor uses 8192 entries for the bimodal, gshare and meta predictors, taking 6kBytes of state. The parameters chosen for the issue-width and cache sizes are roughly twice those on the Alpha 21264. We assume the memory system can support 16 concurrent memory operations; we model the delay for all memory operations and bank contention, but do not model bus contention.

5 Analysis of Simulation Results

There are a number of questions we hoped to answer by this study. We address each in turn. Throughout this section, we will repeat data in various tables; for example, the results for conservative dependence prediction will appear in each table. We do this to simplify the comparison of a particular mechanism where appropriate, and to avoid cluttering the tables. We simulated both the re-fetch and the re-execute policies. Despite the assertions of other studies [18], we agree with Chrysos *et al* [2] that it is unlikely that a processor would meet timing goals if it used the large central window that is needed for a simple implementation of the re-execute mechanism. As we'll see, if we only examined the re-execute mechanism, we would draw incorrect conclusions about the efficacy of certain mechanisms.

5.1 Potential for Dependence Prediction

We wanted to determine the *relative potential* of dependence and value prediction, assuming we could build a "perfect" dependence or value predictor. Figure 4 shows the IPC for four processor configurations. In the first configuration, labeled "conservative", loads must wait for all priors stores before executing. This is the baseline architecture against which we compare all other configurations in this paper; since the dependence prediction was only recently posed as a problem for out-of-order architectures [12], we assume most other simulators assume conservative memory dependences. The remaining columns are self explanatory, except for the last column – that column shows the performance when perfect dependence, value *and* branch prediction are used. In this model, the processor wastes no effort on speculative work and is primarily limited by the instruction fetch and memory subsystem.

Perfect dependence prediction significantly improves performance (with an average speedup of 54% on the SPECint and 106% on the SPECfp) and perfect value prediction improves that further (115% on SPECint, 136% on SPECfp). The combination of perfect value *and* dependence prediction appears to offer little benefit; this makes sense since perfect value prediction would subsume perfect dependence prediction – not only do you know the outcome of the dependence decision, you also know the value. As we'll see, the combination of practical dependence and value predictors *does* offer improvement over either in isolation. Lastly, the development of a perfect dependence and load value predictor would not remove the need for improvements in branch prediction for integer programs.

The results comparing perfect predictors are fairly much as one would expect, but they serve to confirm that our simulator infrastructure is configured such that there is a sufficient memory delay that value prediction should be able to evince some performance benefit over dependence prediction.

5.2 Performance of Dependence Predictors

Table 3 compares the performance of the different dependence predictors. All but the first two columns are self-descriptive. The column marked "conservative" uses no dependence prediction and the column marked "naïve" uses naive value prediction, *i.e.* we assume there is no dependence for *any* load and they are issued as soon as their addresses resolve.

Any of the speculative dependence prediction techniques perform better than conservative speculation across both fetch policies. The different fetch policies introduce overheads only when dependence mispredictions occur; thus, the perfect and conservative dependence predictors have the same performance for the re-fetch and re-execute policies, since neither causes any mispredictions. Confirming the results of [2], store sets is the best realizable dependence predictor and is so close to perfect dependence prediction that it's unlikely that future work in pure dependence prediction is warranted.

All the dependence mechanisms perform better with the re-execute policy; this is expected since the re-execute policy basically lowers the cost of a dependence misprediction and only has a minor effect on the accuracy of the dependence predictor.

The results in Table 3 also confirm the design decision of the load-wait tables that were developed for the 21264 which uses a re-fetch policy [8]. Table 3(a) shows that the load-wait predictor has better performance than naïve speculation for refetch, but Table 3(b) shows that the naïve speculation outperforms the load-wait mechanism. The load-wait mechanism tends to be conservative, introducing false memory dependences thereby reducing the available parallelism but also reducing the number of replay traps. The *naive* mechanism has no false dependences, but encounters more traps. Since the cost of recovery in the re-execute policy is so low, the naïve mechanism has better performance.

Table 4 shows why dependence prediction improves the program IPC, using the conservative and store-sets mechanisms as counter points. For each mechanism, we show the cumulative number of cycles needed for each phase in the lifetime of a load instruction: waiting until the address is known, waiting until memory dependences are resolved and waiting for the value to be returned from memory. With the conservative mechanism, the largest fraction of time is spent waiting for dependences to resolve. Computing the address and accessing the cache are fairly quick by comparison.

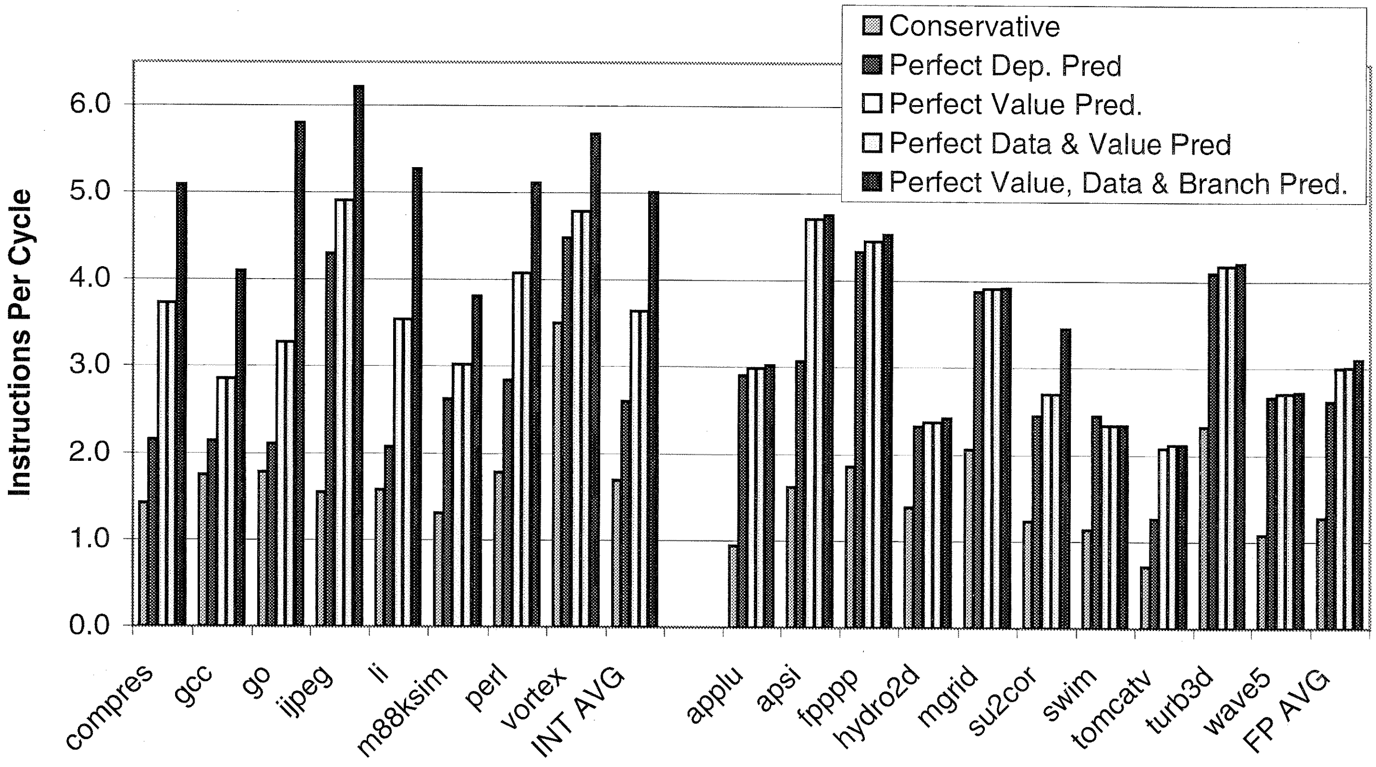

Figure 4: Comparison between the IPC of a processor using different combinations of a perfect dependence predictor, a perfect value predictor and a perfect branch predictor

With store-sets, a large fraction of the loads do not wait for the dependence, and can rapidly access the cache. More importantly, it also appears that unflagged loads tend to hit in the L1 cache, while flagged instructions have a longer average memory access time for the SPECint programs.

We also implemented two different *prediction confirmation mechanisms* to determine if they affected performance. In our baseline simulator, we declared a dependence misprediction if a speculated load later matches the address of an unresolved store *and* the value that was stored was different from the value that was loaded. We call this an *address-and-contents confirmation*. The alternative is an *address-only confirmation*. The address-and-contents confirmation should result in fewer dependence mispredictions than the address-only confirmation. We used the address-and-contents throughout our paper because that mechanism is needed when combining dependence and value prediction. We found that the results shown in this paper hold for either mechanism. As expected, the address-and-contents mechanism results in a higher IPC, but those improvements were consistent across all the dependence predictors. Although the address-and-contents improves the IPC, it would probably exacerbate timing constraints in a processor implementation since more comparators are needed.

5.3 Performance of Practical Load-Value Predictors

Table 5 compares the performance of the different value predictors. This table also shows the baseline conservative dependence model and the best speculative dependence model for comparison. The results confirm that the stride value predictor out-performs the last value predictor by a small amount; this makes sense since the stride predictor is a generalization of LV. What is surprising is that all of the realistic value predictors perform *worse* than the realistic store sets dependence predictor when using the re-fetch recovery policy. This is *not* the case when using the re-execute mechanism. Again, based on assumptions of how easy it is to implement re-fetch *vs.* re-execute, this indicates that considerable work needs to be invested in value prediction before it is viable for realistic architectures.

Earlier studies, particularly [9], showed a performance improvement for realistic value prediction, including the LV predictor we simulated. We conjecture that their baseline architecture used conservative dependence prediction; indeed, the LV predictor has a $\approx$ 16% performance improvement over the conservative model. However, LV or Stride value predictors appear to incur so many traps that simply using a dependence predictor would be a better choice for processors with a re-fetch policy. For a processor with re-execute, the practical value predictors offer roughly the same performance as a good dependence predictor. The LV predictor uses 8KBytes of state (1024 entries of 8 bytes each), and the Stride predictor uses $\approx$ 16KBytes (1024 entries of 8 bytes for value and 8 bytes for stride values). By comparison, the Store Set configuration recommended in [2] uses $\approx$3-4KBytes of storage.

Table 6 shows the time, in processor cycles, between a value prediction and when the value is actually used. It also shows the time until the true value is known, either confirming or refuting the value prediction. For most applications, there is a small difference between these times. This confirms the results of [5], where is was observed that value prediction has a hard time improving the program execution because there is little time between when the values are produced and they are consumed. This time is influenced by a number of factors, including the compiler scheduling algorithm and memory bandwidth. For some applications, such as `tomcatv`, the values are known *before* the predicted value can be used and value

	Cons-vative	Naïve Dep.	Load Wait	Store Sets	Perfect Dep. Pred.
compres	1.42	1.58	1.63	2.13	2.16
gcc	1.75	1.97	1.98	2.12	2.14
go	1.78	1.98	1.99	2.10	2.11
ijpeg	1.55	2.08	2.12	3.82	4.30
li	1.58	2.09	2.03	2.04	2.08
m88ksim	1.31	2.71	2.71	2.62	2.63
perl	1.78	1.95	1.98	2.78	2.84
vortex	3.50	3.27	3.28	4.45	4.48
INT AVG	**1.69**	**2.11**	**2.13**	**2.55**	**2.60**
	Cons-vative	**Naïve Dep.**	**Load Wait**	**Store Sets**	**Perfect Dep. Pred.**
applu	0.94	2.38	2.40	2.59	2.91
apsi	1.62	2.80	2.76	3.06	3.07
fpppp	1.86	3.34	3.31	4.33	4.33
hydro2d	1.39	2.15	2.15	2.30	2.32
mgrid	2.06	3.87	3.87	3.88	3.88
su2cor	1.23	2.10	2.02	2.45	2.45
swim	1.13	2.45	2.45	2.45	2.45
tomcatv	0.71	1.28	1.28	1.24	1.26
turb3d	2.32	3.10	3.10	4.02	4.10
wave5	1.08	2.26	2.34	2.67	2.67
FP AVG	**1.27**	**2.36**	**2.36**	**2.58**	**2.62**

(a) Performance with re-fetch

	Cons-vative	Naïve Dep.	Load Wait	Store Sets	Perfect Dep. Pred.
compres	1.42	2.15	2.14	2.13	2.16
gcc	1.75	2.09	2.08	2.14	2.14
go	1.78	2.10	2.10	2.11	2.11
ijpeg	1.55	4.18	3.62	3.82	4.30
li	1.58	2.02	2.03	2.05	2.08
m88ksim	1.31	2.74	1.71	2.63	2.63
perl	1.78	2.59	2.63	2.83	2.84
vortex	3.50	3.62	3.63	4.46	4.48
INT AVG	**1.69**	**2.51**	**2.33**	**2.57**	**2.60**

	Cons-vative	Naïve Dep.	Load Wait	Store Sets	Perfect Dep. Pred
applu	0.94	2.94	2.86	2.66	2.91
apsi	1.62	3.05	2.98	3.06	3.07
fpppp	1.86	4.30	4.18	4.33	4.33
hydro2d	1.39	2.32	2.26	2.30	2.32
mgrid	2.06	3.88	3.88	3.88	3.88
su2cor	1.23	2.41	2.19	2.45	2.45
swim	1.13	2.45	2.45	2.45	2.45
tomcatv	0.71	1.32	1.28	1.24	1.26
turb3d	2.32	3.80	3.89	4.02	4.10
wave5	1.08	2.68	2.50	2.67	2.67
FP AVG	**1.27**	**2.63**	**2.55**	**2.58**	**2.62**

(b) Performance with re-execute

Table 3: Performance of different dependence predictors with two recovery mechanisms

prediction has no benefit.

We conducted another experiment to see how sensitive value predictors are to the time when the table is updated. In one configuration, the processor updated the LV (or stride) tables when an instruction *executes*, but before we know if that instruction will actually commit. In the second configuration, we updated the table when the instructions actually commit. These configurations have virtually identical performance. We conjecture that this occurs because allocating at execute introduces spurious entries into the LV table due to speculative execution, but it also updates the results in a more timely fashion so that subsequent loads can see the most recently predicted value. Similarly, allocate at commit eliminates spurious entries, but it also reduces the opportunities for loads to update the table to benefit subsequent loads.

5.4 Combining Dependence Prediction and Load-Value Predictors

Finally, we compare the benefits of value and dependence prediction. In this configuration, we combine the store sets and the stride predictors. When a load is issued, we determine if we can predict the value *and* determine if we should immediately issue the load. The load instruction can only cause a single replay trap – eventually the load completes and we verify the value or a store causes a dependence trap and forwards the value to the load at the same time.[2]

Table 7 compares this configuration to the previous best dependence and value predictors. The performance of the refetch model is hurt by the addition of the value predictor and the performance of the re-execute model is helped by a small amount. Again, the poor accuracy of existing value predictors appears to be the culprit.

6 Conclusions

We have conducted a preliminary study comparing the benefits of load dependence prediction and value prediction. As with all simulation studies, this study is limited in that it address a small portion of a large parameter space. However, we believe we have chosen a processor configuration that would be feasible by 2001 or 2002.

Our conclusions vary depending on the credibility of various implementation factors. We believe that it it unlikely that processors will be able to implement an efficient re-execute mechanism, and that re-fetch is a more likely recovery mechanism. Given this, it seems unlikely that current value predictors will improve performance sufficiently to warrant their inclusion in an implementation. On the other hand, dependence predictors should be included in any processor implementation (or, for that matter, processor simulation).

[2]This is a slight simplification. In any speculative configuration in this study, loads can receive several dependence traps due to cascading dependences within the program.

	Conservative Dependence			Store-Sets Dependence Predictor					
	All Loads			Unflagged Loads			Flagged Loads		
	Addr. Res.	Dep. Res.	Mem. Ref Compl.	Addr. Res.	Dep. Res.	Mem. Ref Compl.	Addr. Res.	Dep. Res.	Mem. Ref Compl.
compres	18.38	36.63	60.34	11.60	-	13.50	5.71	12.90	17.09
gcc	8.46	17.14	23.43	5.10	-	6.70	2.86	13.00	16.80
go	6.78	6.91	14.16	5.23	-	6.34	4.92	1.22	7.12
ijpeg	47.25	150.71	155.60	10.40	-	13.50	2.89	50.40	54.31
li	8.98	25.81	28.17	4.31	-	5.95	4.20	1.62	6.46
m88ksim	3.31	112.74	106.12	2.85	-	3.88	3.47	2.69	6.44
perl	8.49	28.02	29.59	4.58	-	5.65	2.90	3.51	6.13
vortex	17.36	46.73	48.97	4.30	-	5.52	2.39	2.31	5.18
INT AVG	**14.88**	**53.09**	**58.30**	**6.05**	**-**	**7.63**	**3.67**	**10.96**	**14.94**

	Conservative Dependence			Store-Sets Dependence Predictor					
	All Loads			Unflagged Loads			Flagged Loads		
	Addr. Res.	Dep. Res.	Mem. Ref Compl.	Addr. Res.	Dep. Res.	Mem. Ref Compl.	Addr. Res.	Dep. Res.	Mem. Ref Compl.
applu	51.84	248.72	249.53	4.11	-	35.33	4.92	18.50	26.58
apsi	48.36	190.63	189.36	3.29	-	6.38	2.53	64.60	61.87
fpppp	8.02	188.29	188.09	2.65	-	3.68	2.24	11.50	13.47
hydro2d	13.66	192.62	196.10	4.04	-	33.53	3.06	65.80	62.14
mgrid	3.09	206.74	209.62	2.49	-	29.16	2.91	10.20	11.24
su2cor	33.01	137.28	139.84	5.51	-	17.70	4.40	12.50	13.49
swim	2.04	254.94	254.95	2.07	-	44.96	-	-	-
tomcatv	2.01	238.24	246.47	2.00	-	33.85	2.00	254.00	254.90
turb3d	56.85	120.39	124.08	2.64	-	4.46	3.15	26.20	27.30
wave5	44.32	170.64	170.49	3.08	-	14.00	3.19	22.20	24.13
FP AVG	**26.32**	**194.85**	**196.85**	**3.19**	**-**	**22.31**	**2.74**	**48.45**	**49.41**

Table 4: Cycles spent in each phase of the lifetime of a load instruction for different dependence constraints. For the Store Sets model, "flagged" loads are those that are thought to have dependences, while "unflagged" loads are though to not have dependences. The program `swim` has no significant number of flagged dependences.

We feel that this work underlines two interesting research directions. First, it would be very useful to find a scalable, implementable way to build a re-execute recovery mechanism. Indeed, this would probably contribute more to processor performance than adding a better value predictor to a processor with a re-fetch recovery mechanism. Second, it is clear that there is considerable "headroom" in value prediction, but that accurate comparisons of value predictors needs to include measures of accuracy and coverage *and* information from performance models. Simple metrics such as accuracy and coverage do not capture the temporal dynamics of value prediction that influence the final performance.

References

[1] ACM SIGARCH and IEEE Computer Society TCCA. *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, 1998.

[2] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* [1].

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

[4] Freddy Gabbay and Avi Mendelson. Can program profiling support value prediction? In *Proceedings of the 30th Annual International Symposium on Microarchitecture* [7], pages 270–280.

[5] Freddy Gabbay and Avi Mendelson. The effect of instruction fetch bandwidth on value prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* [1], pages 272–281.

[6] 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 *Proc. ASPLOS VI*, pages 123–128, October 1994.

	ns- ive	ɔre ɛts	ıst lue	ide	fect Pred.
compres	1.42	2.13	1.54	1.56	3.73
gcc	1.75	2.12	1.82	1.85	2.86
go	1.78	2.10	1.80	1.84	3.27
ijpeg	1.55	3.82	2.00	1.95	4.90
li	1.58	2.04	1.97	1.91	3.54
m88ksim	1.31	2.62	2.59	2.57	3.02
perl	1.78	2.78	1.80	1.86	4.07
tex	50	45	76	11	78
AVG	**69**	**55**	**97**	**00**	**64**

	ns- ive	ɔre ɛts	ıst lue	ide	fect Pred.
applu	0.94	2.59	2.27	2.27	2.99
apsi	1.62	3.06	2.72	2.68	4.70
fpppp	1.86	4.33	3.21	3.27	4.45
hydro2d	1.39	2.30	2.14	2.15	2.37
mgrid	2.06	3.88	3.89	3.87	3.91
su2cor	1.23	2.45	2.11	2.10	2.70
swim	1.13	2.45	2.25	2.07	2.34
tomcatv	0.71	1.24	1.28	1.22	2.08
turb3d	2.32	4.02	3.07	3.10	4.18
ve5	38	67	24	22	71
\VG	**27**	**58**	**32**	**28**	**00**

(a) Performance with re-fetch

	Cons- vative	Store Sets	Last Value	Stride	Perfect Value Pred.
compres	1.42	2.13	2.18	2.21	3.73
gcc	1.75	2.14	2.08	2.10	2.86
go	1.78	2.11	2.06	2.07	3.27
ijpeg	1.55	3.82	4.22	4.23	4.90
li	1.58	2.05	2.21	2.22	3.54
m88ksim	1.31	2.63	2.76	2.76	3.02
perl	1.78	2.83	2.59	2.59	4.07
vortex	3.50	4.46	3.50	3.71	4.78
INT AVG	**1.69**	**2.57**	**2.54**	**2.57**	**3.64**

	Cons- vative	Store Sets	Last Value	Stride	Perfect Value Pred.
applu	0.94	2.66	2.94	2.94	2.99
apsi	1.62	3.06	3.06	3.06	4.70
fpppp	1.86	4.33	4.30	4.31	4.45
hydro2d	1.39	2.30	2.35	2.32	2.37
mgrid	2.06	3.88	3.89	3.89	3.91
su2cor	1.23	2.45	2.42	2.41	2.70
swim	1.13	2.45	2.45	2.45	2.34
tomcatv	0.71	1.24	1.32	1.32	2.08
turb3d	2.32	4.02	3.84	3.84	4.18
wave5	1.08	2.67	2.68	2.68	2.71
FP AVG	**1.27**	**2.58**	**2.64**	**2.63**	**3.00**

(b) Performance with re-execute

Table 5: Performance of different value predictors with two recovery mechanisms

[7] IEEE Computer Society TC-MICRO and ACM SIGMICRO. *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Research Triangle Park, North Carolina, December 1–3, 1997.

[8] R. E. Kessler, E. J. McLellan, and D. A. Webb. The alpha 21264 microprocessor architecture. In *1998 Intl. Conference on Computer Design (ICCD'98)*, 1998.

[9] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, December 1996.

[10] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *ASPLOS-VII*, pages 138–147, October 1996.

[11] Scott McFarling. Combining branch predictors. TN 36, DEC-WRL, June 1993.

[12] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, pages 181–193, Denver, Colorado, June 2–4, 1997. ACM SIGARCH and IEEE Computer Society TCCA.

[13] Andreas Moshovos and Gurindar S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th Annual International Symposium on Microarchitecture* [7], pages 235–245.

[14] Abhijit Paithankar. AINT: A Tool for Simulation of Shared-Memory Multiprocessors. Master's thesis, University of Colorado at Boulder, 1996.

[15] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *Proceedings of the 30th Annual International Symposium on Microarchitecture* [7], pages 248–258.

[16] Yiannakis Sazeides, Stamatis Vassiliadis, and James E. Smith. The performance potential of data dependence speculation and collapsing. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 238–247, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[17] Amitabh Srivastava and David W. Wall. Link-time optimization of address calculation on a 64-bit architecture. *ACM SIGPLAN*, 29(6):49–60, June 1994. Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.

[18] Gary Tyson and Todd Austin. Improving the accuracy and performance of memory communication through renaming. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 218–227, December 1997.

[19] Kai Wang and Manoj Franklin. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture* [7], pages 281–290.

	Last Value		Stride	
	Spec to Use	Spec to Resolution	Spec to Use	Spec to Resolution
compress	4.23	4.95	1.73	4.31
gcc	1.85	4.72	1.85	4.66
go	1.48	4.25	1.46	4.15
ijpeg	3.35	4.76	3.20	4.70
li	1.41	4.53	1.38	4.42
m88ksim	1.12	3.38	1.15	3.40
perl	1.47	3.70	1.46	3.67
vortex	1.46	3.84	1.46	3.79
INT AVG	**2.05**	**4.27**	**1.71**	**4.14**

	Last Value		Stride	
	Spec to Use	Spec to Resolution	Spec to Use	Spec to Resolution
applu	3.95	22.00	3.96	22.08
apsi	1.90	3.68	1.75	3.63
fpppp	1.78	3.42	1.74	3.46
hydro2d	4.23	31.00	4.14	30.54
mgrid	15.70	95.10	15.63	92.63
su2cor	5.14	9.84	4.22	9.66
swim	6.44	27.10	5.40	21.92
tomcatv	7.53	3.99	3.26	6.13
turb3d	1.99	3.88	2.00	3.80
wave5	1.74	5.13	1.74	5.04
FP AVG	**5.04**	**20.51**	**4.38**	**19.89**

(a) Performance with re-fetch

	Last Value		Stride	
	Spec to Use	Spec to Resolution	Spec to Use	Spec to Resolution
compress	5.12	5.62	2.11	4.83
gcc	2.05	5.24	2.07	5.15
go	1.58	4.47	1.54	4.37
ijpeg	5.95	7.70	5.52	8.01
li	1.50	5.39	1.47	5.18
m88ksim	1.12	3.41	1.15	3.43
perl	1.56	3.94	1.52	3.89
vortex	1.52	4.15	1.50	4.04
INT AVG	**2.55**	**4.99**	**2.11**	**4.86**

	Last Value		Stride	
	Spec to Use	Spec to Resolution	Spec to Use	Spec to Resolution
applu	4.43	21.40	4.44	21.44
apsi	2.08	3.91	1.99	3.81
fpppp	1.93	3.63	1.86	3.63
hydro2d	4.67	33.40	4.64	33.11
mgrid	15.70	95.30	15.68	92.86
su2cor	6.10	11.30	5.76	11.60
swim	12.50	45.70	12.62	43.79
tomcatv	20.90	34.30	5.58	30.76
turb3d	2.16	4.20	2.14	4.07
wave5	2.66	7.97	2.62	7.61
FP AVG	**7.31**	**26.11**	**5.73**	**25.27**

(b) Performance with re-execute

Table 6: Time, in cycles, until the value derived from a value predictor is used and the time until the true value is known.

	Cons-vative	Store Sets	Last Value	Stride	Store Sets & Stride	Perfect Data & Value
compres	1.42	2.13	1.54	1.56	1.76	3.73
gcc	1.75	2.12	1.82	1.85	1.95	2.86
go	1.78	2.10	1.80	1.84	1.91	3.27
ijpeg	1.55	3.82	2.00	1.95	3.03	4.90
li	1.58	2.04	1.97	1.91	2.02	3.54
m88ksim	1.31	2.62	2.59	2.57	2.59	3.02
perl	1.78	2.78	1.80	1.86	2.24	4.07
vortex	3.50	4.45	2.76	3.11	3.63	4.78
INT AVG	**1.69**	**2.55**	**1.97**	**2.00**	**2.26**	**3.64**

	Cons-vative	Store Sets	Last Value	Stride	Store Sets & Stride	Perfect Data & Value
applu	0.94	2.59	2.27	2.27	2.74	2.99
apsi	1.62	3.06	2.72	2.68	2.88	4.70
fpppp	1.86	4.33	3.21	3.27	4.17	4.45
hydro2d	1.39	2.30	2.14	2.15	2.28	2.37
mgrid	2.06	3.88	3.89	3.87	3.88	3.91
su2cor	1.23	2.45	2.11	2.10	2.41	2.70
swim	1.13	2.45	2.25	2.07	2.07	2.34
tomcatv	0.71	1.24	1.28	1.22	1.18	2.12
turb3d	2.32	4.02	3.07	3.10	3.93	4.18
wave5	1.08	2.67	2.24	2.22	2.60	2.71
FP AVG	**1.27**	**2.58**	**2.32**	**2.28**	**2.48**	**3.01**

(a) Performance with re-fetch

	Cons-vative	Store Sets	Last Value	Stride	Store Sets & Stride	Perfect Data & Value
compres	1.42	2.13	2.18	2.21	2.19	3.73
gcc	1.75	2.14	2.08	2.10	2.14	2.86
go	1.78	2.11	2.06	2.07	2.08	3.27
ijpeg	1.55	3.82	4.22	4.23	3.88	4.90
li	1.58	2.05	2.21	2.22	2.28	3.54
m88ksim	1.31	2.63	2.76	2.76	2.77	3.02
perl	1.78	2.83	2.59	2.59	2.71	4.07
vortex	3.50	4.46	3.50	3.71	4.17	4.78
INT AVG	**1.69**	**2.57**	**2.54**	**2.57**	**2.61**	**3.64**

	Cons-vative	Store Sets	Last Value	Stride	Store Sets & Stride	Perfect Data & Value
applu	0.94	2.66	2.94	2.94	2.69	2.99
apsi	1.62	3.06	3.06	3.06	3.06	4.70
fpppp	1.86	4.33	4.30	4.31	4.34	4.45
hydro2d	1.39	2.30	2.35	2.32	2.31	2.37
mgrid	2.06	3.88	3.89	3.89	3.89	3.91
su2cor	1.23	2.45	2.42	2.41	2.45	2.70
swim	1.13	2.45	2.45	2.45	2.45	2.34
tomcatv	0.71	1.24	1.32	1.32	1.24	2.12
turb3d	2.32	4.02	3.84	3.84	3.98	4.18
wave5	1.08	2.67	2.68	2.68	2.70	2.71
FP AVG	**1.27**	**2.58**	**2.64**	**2.63**	**2.59**	**3.01**

(b) Performance with re-execute

Table 7: Performance of combined value and dependence predictors with two recovery mechanisms