Exploiting Type Information in Load-Value Predictors

Nana B. Sam and Martin Burtscher
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853
{besema, burtscher}@csl.cornell.edu

ABSTRACT

To alleviate the high cost of main-memory accesses, computer architects have proposed various speculation mechanisms, including load-value prediction. A load-value predictor forecasts the result of load instructions, thus allowing dependent instructions to execute without having to wait for the memory access to complete. Unfortunately, costly mispredictions hinder the true potential of load-value prediction.

This paper describes several approaches to build more accurate and faster load-value predictors with little extra hardware by exploiting the hardware type of the load instructions. Our techniques are easily implementable in any CPU that supports multiple load types, e.g., byte, word, single, etc. Compared to traditional load-value predictors, our schemes substantially reduce the number of mispredictions with a concomitant speedup of the CPU. Moreover, we show that the type of a load can effectively be used as the selector in a hybrid predictor.

1. INTRODUCTION

Memory accesses, especially loads, can degrade the performance of a processor. First, due to the widening gap between CPU and memory speeds, memory accesses are slow and are becoming slower. Second, memory reads limit the available instruction-level parallelism because instructions that use the result of a load stall until the memory access is complete, which potentially lengthens the critical path of a program. Load-value prediction [6, 11] addresses these problems by predicting the result of load instructions. Dependent instructions can immediately consume the predicted value and are thus able to execute concurrently with the memory access. If the predicted value is incorrect, the speculation hardware must perform expensive recovery actions. Consequently, load-value predictors are only effective if the benefit of the correct predictions outweighs the penalty incurred by the mispredictions. In other words, avoiding mispredictions is crucial because they slow down the processor.

Most 32- and 64-bit CPUs support load instructions of different types, e.g., byte, word, long, IEEE single, and IEEE double loads as well as sign-extended, zero-extended, and locked variants of some of them. This paper investigates using this type information to design more accurate and faster predictors, to reduce the size of predictors without degrading their performance, and to improve the prediction coverage. Note that our classification of load types differs from prior work [11], which investigated the behavior of address, data, integer, and floating-point loads, i.e., higher-level data types. We exploit the hardware load types as follows.

Type-based misprediction avoidance: Due to the finite table size, different load instructions can map to the same predictor line, which almost always results in detrimental aliasing. Based on the type of the load instruction, it is often possible to detect such aliasing and inhibit the corresponding prediction, which lowers the misprediction rate. We investigate two such schemes.

Type-separated predictor tables: Directing the different load types to separate, smaller tables rather than to one large, shared table is beneficial because smaller tables can be accessed faster. Also, loads of different types generally do not correlate with each other. Thus, separating them can reduce negative interference. We analyze one such approach.

Type-based hybrid predictors: Hybrids combine multiple predictors in one and require a selector to decide which component to use for each prediction. We use the load type as the selector. This is advantageous because it eliminates the need to store selection information, simplifies and speeds up the predictor hardware, and reduces pollution because not all loads update all components, which improves the prediction accuracy and the predictor's effective capacity.

The remainder of this paper is organized as follows. Section 2 provides background information. Section 3 summarizes related work. Section 4 describes our type-based techniques in more detail. Section 5 presents the evaluation methodology. Section 6 studies the performance of our approaches. Section 7 concludes with a summary.

2. BACKGROUND

2.1 Load-Value Prediction

Predicting load values may initially seem almost impossible since a 32-bit word can hold over four billion distinct values. Fortunately, load instructions exhibit value locality [6, 11], i.e., they often fetch predictable sequences of values. Several predictors have been proposed that take advantage of this behavior to predict

load values before they are fetched from memory. We apply our type-based techniques to the following three predictors, which we believe to be representative in complexity and performance of many previously proposed predictors.

The *last value* predictor (LV) [6, 11, 16] predicts that a load instruction will load the same value it did the previous time it executed. LV can only predict sequences of repeating values.

The *stride 2-delta* predictor (ST2D) [16] remembers the last value for each load (like LV) but also maintains a stride, which is the difference between the last two loaded values. To make a prediction, ST2D adds the stride to the last value. When a load is completed, ST2D updates the last value but only updates the stride if it has encountered the same stride twice in a row, which enhances the performance [16]. ST2D can predict sequences with zero and non-zero strides.

The *third order differential finite context method* predictor (DFCM3) [7] computes a hash value out of the difference between the last three loaded values to index the predictor's second-level table [13, 14, 15]. This table stores the strides that followed previously seen sequence of three consecutive strides. Since the table is shared, load instructions can communicate information to one another in this predictor. Hence, after observing a sequence of load values, DFCM3 can predict any load that loads the same sequence or a different sequence with the same strides.

2.2 Confidence Estimation

Because making no prediction and waiting for the memory access to complete is faster than making an incorrect prediction and having to recover from it, most load-value predictors in the literature include a confidence estimator. Confidence estimators inhibit predictions that are likely to be incorrect [5, 11, 13, 14] and thus reduce the number of mispredictions and the associated recovery cost, which improves the predictor's overall performance.

The most frequently used confidence estimator, the bimodal confidence estimator [11, 13, 14], is based on saturating up/down counters and has four parameters: a maximum, a threshold, a penalty, and an award. The maximum is the upper bound of the counter (the minimum is always zero). A prediction is made only if the count is equal to or above the threshold. When an unpredictable value is encountered, the counter is decremented by the penalty; otherwise it is incremented by the award.

3. RELATED WORK

Lipasti et al. [11] investigated how predictable different kinds of load instructions are. They found that while all loads are value predictable to a degree, address loads have slightly better value locality than data loads, instruction address loads hold an edge over data address loads, and integer values are more predictable than floating-point values. We take advantage of the differences in predictability by separating loads by their hardware type rather than by their kind.

Gonzalez and Gonzalez [8] found that the benefit of data value prediction increases significantly as the instruction window size grows, indicating that value prediction is likely to play an important role in future processors. Moreover, they observed an almost linear correlation between the number of correctly predicted instructions and the resulting performance improvement, emphasizing the need for more accurate prediction approaches, which our techniques offer.

One result of Sazeides and Smiths's work [17] is that over half of the mispredicted branches have predictable input values, implying that a side effect of accurate value prediction should be improved branch prediction. Gonzalez and Gonzalez proposed predictor implementations to take advantage of this correlation [9]. Sodani and Sohi [19] build on the Gonzalez studies. They found, among other things, that resolving branches using predicted operands is only beneficial in the presence of low value misprediction rates, which our approaches provide.

Calder et al. [5] examined selection techniques to minimize predictor capacity conflicts by prohibiting unimportant instructions from using the predictor. They found that loads were responsible for most of the latency on the critical path and hence predicting only loads represents a good filtering criterion. We implicitly use this criterion because we only investigate load-value predictors.

In order to save space and power, Loh [12] replaced the traditional predictor with multiple smaller tables with different widths and proposed a data-width predictor to choose among the tables. Our techniques also result in space-saving predictors. Unlike in Loh's study, we use the load type information provided by the decoder and therefore do not need to measure the actual width of the load value nor do we need a data-width predictor.

4. TYPE-BASED TECHNIQUES

Most high-end microprocessors support different types of loads, such as byte, word, and long-word loads. Nevertheless, the value predictors in the literature treat every load alike. We propose the following schemes to exploit the different load types. Note that the load type is determined in the decoder and that our type checks are trivial and can be done rapidly.

4.1 Type-Based Misprediction Avoidance

Sometimes loads of different types are mapped to the same predictor line due to the finite table size. Moreover, in the DFCM3 predictor, all loads share the sec-

ond-level table. Thus, it is possible that a load is predicted using information from another load, which usually results in a costly misprediction. Such detrimental aliasing can often be detected by taking the type of load into account. We propose the following two schemes to avoid such likely mispredictions.

4.1.1 'Check'

Many architectures support zero-extended and sign-extended load values. In the Alpha 21264 ISA, which we use for our study, *byte* and *word* loads are zero-extended and *long* loads are sign-extended to 64 bits. Furthermore, single-precision floating-point loads expand the eight-bit memory-format exponent into an eleven-bit register-format exponent and set the 29 low-order fraction bits to zero [2].

We examine a simple scheme that inhibits predictions that do not match the format prescribed in the ISA for a particular load type. For example, assume a load instruction that expects a zero-extended value. In our scheme, we inhibit all predictions that have a non-zero bit in any bit position that corresponds to the zero-extended portion in the value. Sign-extended and single-to-double converted values can be tested in a similar manner. We call this scheme 'check'.

4.1.2 'Type-Tag'

Previously proposed load-value predictors depend on confidence estimation to filter out likely mispredictions. We use the type of a load instruction to further eliminate possible mispredictions. For instance, assume the current load instruction is a byte load and expects a value of 1. Let us further assume the selected predictor line contains a value of 2 from a word load and has a confidence above the threshold. In a conventional predictor a misprediction is made. However, with our 'type-tag' method, the prediction is inhibited due to the type mismatch. Hence, the predictor can identify and inhibit incorrect predictions based on a simple type test and thus avoid the recovery cost. A few additional bits (three in this study) per predictor line are required to record the load type.

4.2 Type-Separated Predictor Tables

Splitting larger tables into multiple smaller ones can be beneficial because smaller tables have faster access times. Moreover, having multiple tables allows the direction of different types of load instructions to different tables. Also, it allows each table to be independently optimized to the corresponding load type. We call this scheme 'split'.

The width of a conventional load-value-predictor table is determined by the largest load size. In this experiment, we split the conventional table with its 64-bit entries into four smaller ones of half the height for 8-, 16-, 32-, and 64-bit loads. Note that the 32- and 64-bit tables are shared between integer and floating-point loads. This arrangement results in a slightly lower overall size (in number of bits of storage) but provides space for twice as many load values.

4.3 Type-Based Hybrid Predictors

Hybrid predictors combine multiple predictors [4, 14, 20] and employ a selector to decide which component to use for each prediction. We utilize the type of the load instructions to make this decision. Doing so is beneficial as it eliminates the extra storage for the selector. Also, each load only updates one component of the hybrid, thus eliminating the pollution of the other components, which increases the hybrid's effective capacity. This is our 'type-hybrid' scheme.

5. METHODOLOGY

All our measurements are performed on a simulator derived from the SimpleScalar/Alpha 3.0 tool set [3], a suite of functional and timing simulation tools for the Alpha ISA. The simulations are execution-driven and include execution down speculative paths until the fault, TLB miss, branch misprediction, or load misspeculation is detected.

5.1 Simulation Framework

We simulated a 4-way, 7-stage, superscalar, out-of-order CPU (similar to the Alpha 21264) with a 128-entry instruction window, a 32-entry load/store buffer, a 32-entry 8-way instruction TLB, a 64-entry 8-way data TLB, both with a 30-cycle miss penalty, a 64KB, 2-way 1-cycle L1 instruction cache, a 64KB, 2-way 3-cycle L1 data cache, a unified 4MB, 4-way 20-cycle L2 cache, an 8K-entry hybrid gshare-bimodal branch predictor, four integer ALU units, two floating-point adders and one floating-point MULT/DIV unit. The data cache is write-back and non-blocking with two ports. The caches have a block size of 64 bytes. All functional units except the divide unit are pipelined to allow a new instruction to initiate execution each cycle. It takes 160 cycles to access main memory. This represents our baseline architecture.

We model 2048-entry load-value predictors. The predictors include a bimodal confidence estimator (CE) with three-bit saturating counters with a threshold of five, a penalty of three, and an award of one. We used the same CE configuration for all the predictors and did not vary it for this study. The predictors and confidence estimator are described in Section 2. Each prediction is made after decode, and the predictors are updated as soon as the true load value is available, there are no speculative updates, and an out-of-date prediction is made as long as there are pending updates to the same predictor line. We use the re-fetch misprediction recovery scheme [6]. It is identical to that used for recovering from branch mispredictions. We enable 'al-

ways no alias' dependence prediction (to predict aliases between loads and stores) since we believe future high-end CPUs will include a similar feature [10, 13].

5.2 Alpha 21264 Load Types

We consider all types of loads that are present in the Alpha 21264 architecture, except VAX and locked loads, which do not occur in our benchmark programs. Table 1 gives information about the load types.

Table 1: Alpha 21264 load types.

instr	type	special format	bits
ldbu	byte	zero-extended	8
ldwu	word	zero-extended	16
ldl	long	sign-extended	32
ldq, ldqu	quad (aligned, unaligned)	-	64
lds	IEEE single-precision	single to double	32
ldt	IEEE double-precision	-	64

5.3 Benchmarks

Table 2 describes the twelve C programs from the SPECcpu2000 benchmark suite [1] that we use for our measurements. They were compiled on a DEC Alpha 21264A processor with the DEC C compiler under the OSF/1 v5.1 operating system using a high optimization level. We use the SPEC-provided train inputs. We skipped over the initialization part of each program, which is usually not representative of the general program behavior [18]. Results are then reported for simulating each program for 300 million committed instructions. The remaining three C programs were not used because SimpleScalar could not run those binaries. We further excluded the C++ and Fortran programs due to the lack of a compiler. Table 2 shows the number of instructions (in billions) that we skipped before beginning the cycle-accurate simulations, the number of simulated load instructions (in millions), the percentage of executed instructions that are loads, and the instruction per cycle (IPC) on the baseline processor.

Table 2: Information about the simulated segments.

program	skipped insts (B)	simulated loads (M)	% loads	base IPC
ammp	4.2	76	25.2	1.415
art	3.6	83	27.7	0.822
bzip2	3.3	76	25.4	1.732
crafty	3.3	91	30.3	1.664
equake	4.2	115	38.5	1.081
gcc	4.2	102	33.9	1.470
gzip	3.0	64	21.2	1.677
mcf	1.2	106	35.3	0.623
mesa	0.6	65	21.8	1.638
twolf	4.2	78	25.9	1.244
vortex	3.3	92	30.8	1.557
vpr	4.2	100	33.3	1.066

6. RESULTS

6.1 Baseline

Figure 1 shows the distribution of the load types for the simulated segment of each of the twelve programs. We find that *quad words* are loaded most often. On average, *ldbu, ldwu, ldl, ldqu, ldq, lds,* and *ldt* represent 9.8%, 3.5%, 18.5%, 11.3%, 38.9%, 5.4%, and 12.6% of the loads, respectively. This variety in load type distribution motivates our approaches.

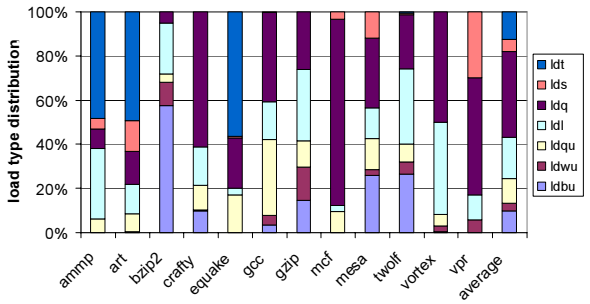

Figure 1: Load type distribution.

To establish a baseline for our type-based predictors, we evaluate the performance of the benchmark programs on our base architecture when traditional (i.e., non-type-based) load-value predictors are used. We find the average prediction coverage of the LV, ST2D and DFCM3 predictors to be 40.2%, 43.8%, and 50.0%, respectively.

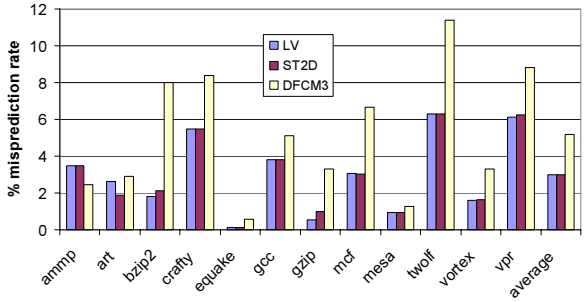

Figure 2: Misprediction rate.

Figure 2 shows the percentage of these predictions that is incorrect. For most of the programs, DFCM3 has the highest misprediction rate. This seems contrary to previous research that indicates that context predictors work the best. However, in our study we have observed that context predictors outperform their simpler counterparts when the number of entries are higher than 2048. For smaller sizes, the context predictors do not perform as well. This is because all loads share the second-level table in DFCM3, which increases aliasing. The exception is *ammp*, which probably loads non-strided sequences that are best captured by DFCM3.

Unless otherwise stated, the performance of our type-based techniques in the following subsections is expressed relative to this baseline, which already includes load-value predictors.

6.2 'Check'

Our 'check' scheme (Section 4.1.1) inhibits predicted values that do not match the ISA-prescribed formats and thus reduces mispredictions. Figure 3 shows the misprediction reduction over the baseline predictors when we inhibit guaranteed mispredictions using 'check'.

We find that 'check' reduces mispredictions by up to 22%. DFCM3 clearly benefits the most from this technique since all loads share the second-level table. Some of these aliases are caught by 'check'. Note that 'check' will never inhibit a correct prediction.

To support faster clock speeds, processor pipelines are getting longer. Since increasing the number of pipeline stages also increases the misprediction recovery cost, we expect 'check' to become even more helpful in the future.

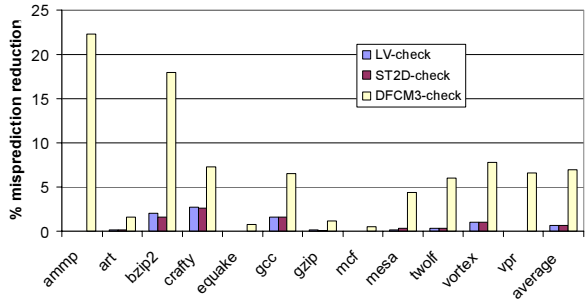

Figure 3: 'Check' misprediction reduction.

6.3 'Type-Tag'

The 'type-tag' approach (Section 4.1.2) inhibits predictions whose types do not match that of the current load instruction. As in 'check', the goal is to reduce mispredictions. Figure 4 shows that 'type-tag' very effectively lowers the misprediction rate (by up to 72%).

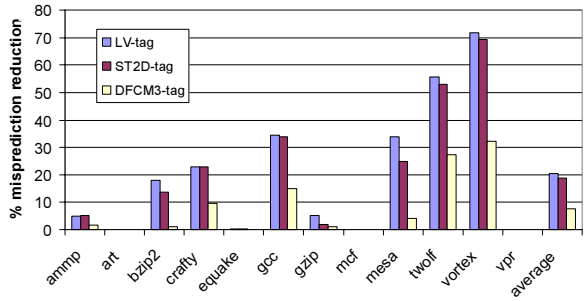

Figure 4: 'Type-Tag' misprediction reduction.

While 'check' cannot improve the frequently executed *ldq, ldqu,* and *ldt* instructions, 'type-tag' distinguishes between all seven load types. Interestingly, LV and ST2D benefit more from 'type-tag' than DFCM3. Since LV and ST2D predict values with a constant or a zero stride, it can be inferred that such sequences stem mostly from 64-bit loads whose aliasing cannot be detected by 'check' but is caught by 'type-tag'. It is natural to expect 'type-tag' to subsume 'check'. However, in some cases 'type-tag' inhibits correct predictions. For example, if a byte load expects 0 and the prediction returns a 0 but of type word, 'tag' inhibits the prediction. A program that suffers from this behavior is *mesa*, where even with a 33% reduction in mispredictions, a slowdown is observed (see Tables 3, 4 and 5). Fortunately, this does not occur often and performance is generally enhanced.

6.4 'Split'

The two goals of 'split' are to speed up the predictor access time and to increase the predictor capacity without increasing the number of transistors. Our 'split' scheme allocates separate tables for 8-, 16-, 32- and 64-bit loads. In order not to increase the overall size of the predictor, we reduce the number of lines in each table by a factor of two (Section 4.2). The resulting predictor is 7% smaller (in number of bits of storage) but can hold twice as many entries as the corresponding predictor with a single table.

Figure 5 shows the misprediction reduction of this scheme for the three predictors. *mesa* benefits the most with 56.8%. We believe this scheme has even more potential since we did not search for the split that gives the optimal performance. *equake* and *mcf*'s performance suffers because they have a highly imbalanced load-type distribution (Figure 1). For instance, over 90% of *mcf*'s loads are 64 bits wide. Reducing the number of lines in the 64-bit table thus increases aliasing and consequently the number of mispredictions. Another potential benefit of 'split' is the ability to predict multiple loads in the same cycle if the loads are of different type.

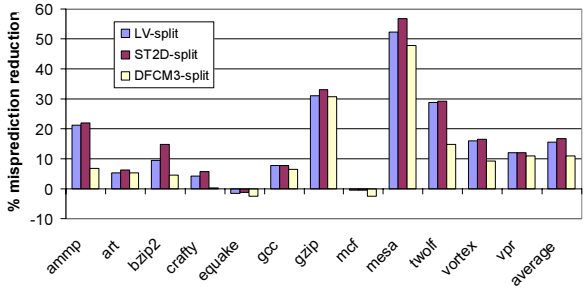

Figure 5: 'Split' misprediction reduction.

6.5 Performance

We compare the IPCs of our enhanced predictors to that of the conventional predictors in Tables 3, 4 and 5. We observe that on average LV and ST2D both perform well with 'type-tag' while DFCM3 performs the best with 'check'.

Table 3: IPCs of LV, LV-check, LV-tag and LV-split.

program	LV	LV-check	LV-tag	LV-split
ammp	1.416	1.423	1.438	1.432
art	1.182	1.188	1.196	1.201
bzip2	1.791	1.802	1.816	1.807
crafty	1.707	1.719	1.738	1.709
equake	1.119	1.125	1.133	1.134
gcc	1.504	1.512	1.527	1.523
gzip	1.712	1.721	1.737	1.732
mcf	0.656	0.660	0.666	0.662
mesa	2.061	2.071	2.027	2.054
twolf	1.267	1.273	1.299	1.284
vortex	1.840	1.858	1.886	1.891
vpr	1.219	1.225	1.237	1.234
average	1.456	1.465	1.475	1.472

Table 4: IPCs of ST2D, ST2D-check, ST2D-tag and ST2D-split.

program	ST2D	ST2D-check	ST2D-tag	ST2D-split
ammp	1.418	1.425	1.440	1.433
art	1.228	1.234	1.243	1.249
bzip2	1.792	1.803	1.817	1.808
crafty	1.706	1.720	1.738	1.709
equake	1.123	1.129	1.136	1.127
gcc	1.506	1.517	1.533	1.525
gzip	1.745	1.755	1.774	1.766
mcf	0.658	0.661	0.668	0.654
mesa	2.059	2.071	2.024	2.071
twolf	1.268	1.274	1.300	1.284
vortex	1.852	1.876	1.901	1.903
vpr	1.221	1.227	1.239	1.236
average	1.465	1.474	1.484	1.480

Table 5: IPCs of DFCM3, DFCM3-check, DFCM3-tag and DFCM-split.

program	DFCM3	DFCM3-check	DFCM3-tag	DFCM3-split
ammp	1.439	1.456	1.454	1.427
art	1.194	1.207	1.204	1.227
bzip2	1.734	1.773	1.735	1.731
crafty	1.695	1.716	1.712	1.698
equake	1.224	1.237	1.233	1.215
gcc	1.507	1.524	1.525	1.529
gzip	1.733	1.750	1.747	1.752
mcf	0.661	0.667	0.667	0.653
mesa	2.089	2.111	2.016	2.107
twolf	1.260	1.274	1.283	1.276
vortex	1.862	1.895	1.892	1.938
vpr	1.190	1.203	1.202	1.217
average	1.466	1.484	1.473	1.481

6.6 'Type-Hybrid'

As discussed in Section 4.3, hybrid predictors can deliver better performance than unit predictors. However, instead of using a meta predictor as the component selector, we propose using the load type. The goals of 'type-hybrid' are to speed up the predictor access and to eliminate the storage requirement of the traditional selector. Figure 6 shows the average predictability by type of the twelve programs. The predictability of a load is defined as the percentage of correct predictions when all the loads are predicted, i.e., no confidence estimator is used to filter out unpredictable loads. This gives us a better picture of which predictor favors which load type. Also, we generated the predictability data using functional simulation and, therefore, the predictors are updated immediately. In the cycle-accurate simulations, the predictors are updated at commit, resulting in some predictions being made with stale data.

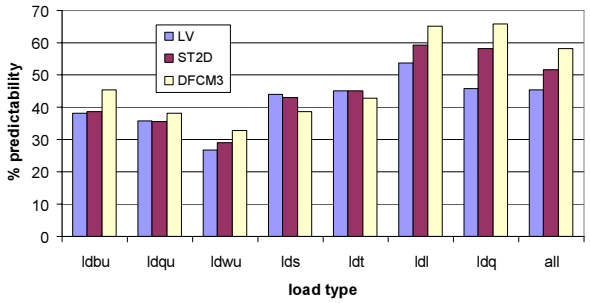

Figure 6: Average load-type predictability.

To determine which hybrid component should be used to predict the different load types in each program, we used cross-validation. For example, the configuration used to predict *twolf* was based on the average load-type predictability of the other eleven programs. We used this approach because we do not expect designers to change the mapping for each program that runs on a CPU, i.e., only one mapping is used for all programs. Cross-validation determines how well our approach would work if a designer chose a mapping with no prior knowledge of the programs that will be run on the processor.

We compare the performance of our 'type-hybrid' with a traditional hybrid, both of which include an LV, ST2D, and a DFCM3 component. In the traditional hybrid, the component with the highest confidence is chosen to make the prediction. We observe that our type-based hybrid reduces mispredictions by up to about 60% on average.

It is evident from the IPC results in Table 6 that 'type-hybrid' outperforms the traditional hybrid with the exception of *equake* and *mcf*. Note that our technique eliminates the storage requirement of the selector, resulting in a reduction in the overall predictor size and

power consumption. It also speeds up the predictor access because no selector needs to be accessed or updated. The improvement in performance, shown in Table 6, is solely due to the reduction in mispredictions and the larger effective capacity. Note that our performance gains from value prediction are lower than those published elsewhere because we include a load-store dependence predictor in our CPU.

Note that 'check' or 'type-tag' can be combined with 'type-hybrid' to obtain even more benefit.

Table 6: IPCs of traditional hybrid and 'type-hybrid'.

	traditional-hybrid	type-hybrid
ammp	1.449	1.452
art	1.215	1.216
bzip2	1.830	1.832
crafty	1.743	1.747
equake	1.146	1.130
gcc	1.536	1.541
gzip	1.783	1.791
mcf	0.672	0.664
mesa	2.110	2.119
twolf	1.295	1.299
vortex	1.879	1.886
vpr	1.251	1.254
average	1.492	1.494

7. CONCLUSIONS

Load instructions represent a large and growing performance bottleneck in microprocessors because of the gap between CPU and memory speeds. Load-value prediction addresses this problem by predicting the value a load instruction will fetch so that dependent instructions do not have to wait for the memory to supply the data. Unfortunately, costly mispredictions reduce the overall benefit of load-value prediction.

Previous studies treat all loads alike. We take advantage of the load type (byte, word, etc.) provided by the decoder to design *more accurate* and *faster* predictors with little additional hardware. We find that our type-based techniques can significantly reduce the misprediction rates and thus increase the speedup delivered by load-value predictors. Furthermore, we show that it is possible to split predictor tables into smaller tables, which have faster access times, without hurting prediction accuracy. Finally, we demonstrate that the type of a load can effectively serve as the component selector in a hybrid predictor, obviating the need for a separate selector.

8. ACKNOWLEDGEMENT

This work has been supported by the National Science Foundation (NSF) under Award #0208567.

9. REFERENCES

[1] SPECcpu2000 benchmarks. http://www.spec.org/osg/cpu2000

[2] Alpha Architecture Reference Manual. Fourth Edition. ftp.compaq.com/pub/products/alphaCPUdocs/alpha_arch_ref.pdf

[3] T. Burger, T. Austin. The SimpleScalar Tool Set, version 2.0. *ACM SIGARCH Computer Architecture News,* 1997.

[4] M. Burtscher, B. G. Zorn. Hybrid Load-Value Predictors. *IEEE Transaction on Computers, Vol. 51:7,* 2002, pp. 759-774.

[5] B. Calder, G. Reinman, D. M. Tullsen. Selective Value Prediction. *26th Annual Int. Symposium on Computer Architecture*, 1999, pp. 64-74.

[6] F. Gabbay. Speculative Execution Based on Value Prediction. Technical Report 1080, Department of Electrical Engineering, Technion-Israel Institue of Technology, 1996.

[7] B. Goeman, H. V. Dierendonck, K. DeBosschere. Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency. *7th International Symposium on High-Performance Computer Architecture*, 2001, pp. 207-216.

[8] J. Gonzalez, A. Gonzalez. The Potential of Data Value Speculation to Boost ILP. *12th International Conference on Supercomputing*, 1998, pp. 21-28.

[9] J. Gonzalez, A. Gonzalez. Control-Flow Speculation through Value Prediction for Superscalar Processors. *International Conference on Parallel Architectures and Compilation Techniques*, 1999, pp. 57-65.

[10] R. E. Kessler, E. J. McLellan, D. A. Webb. The Alpha 21264 Microprocessor Architecture. *International Conference on Computer Design*, 1998, pp. 90-95.

[11] M. H. Lipasti, C. B. Wilkerson, J. P. Shen. Value Locality and Load Value Prediction. *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 138-147.

[12] G. H. Loh. Width-Partitioned Load Value Predictors. *Journal of Instruction-Level Parallelism*, 2003, pp. 1-23.

[13] G. Reinman, B. Calder. Predictive Techniques for Aggressive Load Speculation. *31st IEEE/ACM International Symposium on Microarchitecture*, 1998, pp. 127-137.

[14] B. Rychlik, J. Faistl, B. Krug, J. P. Shen. Efficacy and Performance Impact of Value Prediction. *International Conference on Parallel Architectures and Compilation Techniques*, 1998, pp. 148-154.

[15] Y. Sazeides, J. E. Smith. Implementations of Context Based Value Predictors. Technical Re-

port ECE-97-8, University of Wisconsin, Madison, Wisconsin, 1997.

[16] Y. Sazeides, J. E. Smith. The Predictability of Data Values. *13th International Symposium on Microarchitecture*, 1997, pp. 248-258.

[17] Y. Sazeides, J. E. Smith. Modeling Program Predictability. *25th International Symposium on Computer Architecture*, 1998, pp. 73-84.

[18] T. Sherwood, E. Perelman, G. Hamerly, B. Calder. Automatically Characterizing Large Scale Program Behavior. *10th International Conference on Architectural Support for Programming Languages and Operating Systems,* 2002, pp. 45-57.

[19] A. Sodani, G. S. Sohi. Understanding the Differences between Value Prediction and Instruction Reuse. *31st Annual IEEE/ACM International Symposium on Microarchitecture*, 1998, pp. 205-215.

[20] K. Wang, M. Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. *30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997, pp. 358-363.