Hybrid Load-Value Predictors

Martin Burtscher, *Member*, *IEEE*, and Benjamin G. Zorn, *Member*, *IEEE Computer Society*

Abstract—Load instructions diminish processor performance in two ways. First, due to the continuously widening gap between CPU and memory speed, the relative latency of load instructions grows constantly and already slows program execution. Second, memory reads limit the available instruction-level parallelism because instructions that use the result of a load must wait for the memory access to complete before they can start executing. Load-value predictors alleviate both problems by allowing the CPU to speculatively continue processing without having to wait for load instructions, which can significantly improve the execution speed. While several hybrid load-value predictors have been proposed and found to work well, no systematic study of such predictors exists. In this paper, we investigate the performance of all hybrids that can be built out of a register value, a last value, a stride 2-delta, a last four value, and a finite context method predictor. Our analysis shows that hybrids can deliver 25 percent more speedup than the best single-component predictors. An examination of the individual components of hybrids revealed that predictors with a poor standalone performance sometimes make excellent components in a hybrid, while combining well-performing individual predictors often does not result in an effective hybrid. Our hybridization study identified the *register value + stride 2-delta* predictor as one of the best two-component hybrids. It matches or exceeds the speedup of two-component hybrids from the literature in spite of its substantially smaller and simpler design. Of all the predictors we studied, the *register value + stride 2-delta + last four value* hybrid performs best. It yields a harmonic-mean speedup over the eight SPECint95 programs of 17.2 percent.

Index Terms—Value prediction, value locality, load-value predictor, hybrid predictor, performance metrics.

1 Introduction

Processor technology is advancing at a rapid pace. Over the past two decades, CPU performance has roughly doubled every 18 months. Unfortunately, memory latencies have not improved as quickly. As a result, the speed gap between CPU and memory is constantly growing and has reached a point where it presents one of the biggest performance bottlenecks. *Load-value prediction* is a relatively new approach for improving the performance of microprocessors by breaking dependence chains and hiding the growing latencies. In this paper, we study which predictor combinations yield the most effective, hybrid load-value predictors.

Load instructions frequently fetch from predictable addresses [12]. Moreover, the fetched values themselves are often also predictable [17]. For instance, about half of all the load instructions in the SPECint95 benchmark suite retrieve the same value that they did the previous time they executed. Such behavior, which has been demonstrated explicitly on a number of architectures, is referred to as *value locality* [10], [17]. Load-value locality can be exploited to predict the result of a load instruction before it executes.

Correct load-value predictions enable the CPU to continue processing the dependent instructions without having to wait for the memory access to complete. Of course, it is only known whether a prediction was correct once the true value has been retrieved from memory, which can take many cycles. *Speculative execution* allows the CPU to continue execution with a predicted value before the prediction outcome is known. If it later turns out that the prediction was correct, the speculative status is simply dropped. If the prediction was incorrect, everything the CPU did with the incorrect value has to be purged and redone with the correct value.

Because branch predictors require a similar mechanism to recover from mispredictions, most modern CPUs already contain the necessary hardware to perform this kind of speculation [10]. However, recovering from mispredictions takes time and slows down the processor. Load-value prediction is therefore only effective if most of the predictions are correct.

Several distinct types of load-value locality have been identified and predictors to exploit them have been proposed [6], [10], [17], [25], [27], [28]. While the best performing predictors in the current literature are all hybrids [4], [20], [23], [28], no systematic study of such predictors exists. The goal of this paper is to evaluate all hybrids that can be built out of a register value, a last value, a stride 2-delta, a last four value, and a finite context method predictor to determine which components complement each other well and thus yield high-performing load-value predictors.

Our study identified novel hybrids that are smaller, simpler, and perform better than the best hybrids from the literature. Cycle-accurate simulations of a modern 64-bit RISC processor show that the new hybrids yield harmonic-mean speedups over the eight SPECint95 programs of up to 18 percent.

We show that hybrids are able to deliver 25 percent more speedup than the best single-component predictors and that different components contribute independently to the overall performance. We infer that the existing, distinct types of load-value locality can only be exploited effectively with

- *M. Burtscher is with the School of Electrical and Computer Engineering, Cornell University, Ithaca, NY 14853-3801. E-mail: burtscher@csl.cornell.edu.*
- *B.G. Zorn is with Microsoft Corporation, 1 Microsoft Way, Redmond, WA 98052-6399. E-mail: zorn@microsoft.com.*

Manuscript received 15 Dec. 2000; accepted 18 Dec. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 115252.

TABLE 1
Load-Value Locality

	predictability (%)				
program	reg	lv	st2d	l4v	fcm
compress	9.0	40.4	65.8	41.3	35.9
gcc	19.9	48.5	49.8	65.6	52.0
go	9.2	45.9	47.2	64.0	44.6
ijpeg	9.4	47.5	47.7	54.1	45.4
li	14.3	43.4	50.4	63.8	60.4
m88ksim	29.9	76.1	80.0	83.4	80.3
perl	19.8	50.7	51.4	80.6	70.9
vortex	17.8	65.7	65.3	78.6	66.9
average	16.2	52.3	57.2	66.4	57.0

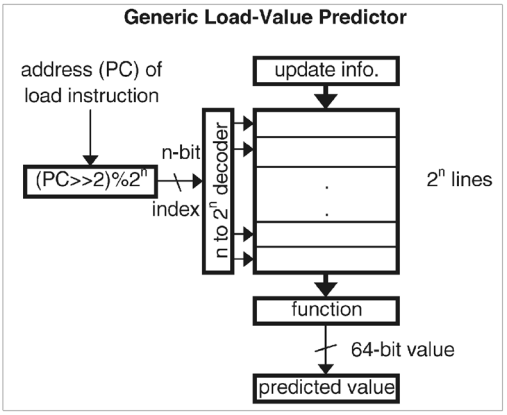


Fig. 1. Basic structure of a context-based load-value predictor.

multicomponent predictors (i.e., hybrids) in which each component is tailored to a different kind of locality.

Our analysis also revealed some unexpected results. For example, powerful individual components frequently do not yield effective hybrids when combined, while some components that perform rather poorly by themselves can form strong coalitions with other components. To explain this behavior, the *load latency* and the *time to first use* of the predicted loads need to be taken into account. The prediction rate and the number of correct predictions, on the other hand, are sometimes inversely correlated with the delivered speedup and are therefore not good performance indicators. Moreover, some hybrids yield a lower performance than their individual components because of *negative interference*.

The remainder of this paper is organized as follows: Section 2 introduces the five component predictors we study, Section 3 presents the evaluation methods, Section 4 discusses the performance of the various hybrids, Section 5 summarizes related work, and Section 6 concludes the paper.

2 Basic Load-Value Predictors

It is almost impossible to predict a random load value correctly. After all, a 32-bit word can hold over four billion distinct values and a 64-bit word over 10^{19} values. Even with only 20 equally distributed values, the odds of picking the correct value are merely five percent, which is probably too low to be useful. This is why almost all proposed load-value predictors make predictions based on *context*, that is, based on recently loaded values.

Using context is promising because load values tend to cluster, repeat, occur in iterating sequences, exhibit discernable patterns, and correlate with one another. Such behavior is referred to as *value locality* (or predictability). To illustrate the extent of exploitable load-value locality, we present Table 1. The table lists five types of predictability found in the eight benchmark programs we use throughout this study. The numbers reflect the percentage of executed load instructions that are predictable.

- The *register value* predictability (reg) indicates how frequently the target register of a load instruction already contains the value that the load is about to read.
- The *last value* predictability (lv) shows how often a load fetches a value that is identical to the previous value fetched by the same load instruction.
- The *stride 2-delta* predictability (st2d) reflects how frequently a load instruction loads a value that is identical to the last value plus a stride. The stride, which is the difference between the last and the second to last loaded value, is only updated if a new stride is encountered at least twice in a row.
- The *last four value* predictability (l4v) indicates how often a value is loaded that is identical to any one of the last four values fetched by the same load instruction.
- The *finite context method* predictability (fcm) shows how frequently a value is loaded that is identical to the value that followed the same sequence of last four values when it was last encountered by any load instruction in the program.

Note that the results for the finite context method are implementation specific, that is, they depend on the hash function and the table size. We used a direct-mapped, tagless, second-level table with 2,048 entries and a select-fold-shift-xor hash function to obtain these results (see Section 2.5). The finite context method is able to deliver significantly higher predictabilities with much larger table sizes [24], but we opted to show results for realistic predictor sizes.

Depending on the program and the type of locality, the load-value locality varies between nine and about 83 percent.

Because they retain information about recently loaded values, all context-based load-value predictors share the same general structure. In essence, they are caches that store previously seen values and possibly other data. Each line in the predictor retains information about one load instruction. However, it is possible that more than one load is mapped to the same line in the predictor. Fig. 1 shows the organization of a generic load-value predictor. Because our study is based on a 64-bit microprocessor, all predictors predict 64-bit values.

Context-based load-value predictors operate as follows: The predictions are computed out of the stored information.

The function that computes the predicted value can be as simple as the identity function (last value predictor) or as complex as accessing a lookup table (finite context method predictor). At any rate, making a prediction must be faster than accessing the conventional memory since the predicted value becomes obsolete as soon as the true load value is available.

Whenever the memory system satisfies a load request, the corresponding predictor line is updated with the true load value and maybe other information. Approaches in which not every load updates the predictor are possible but are beyond the scope of this study. All load instructions, whether they are predicted or not, access the memory and therefore update the predictor. Note that the benefit of load-value prediction does not come from removing load instructions but from breaking dependencies and hiding latencies, that is, from taking the load instructions out of the critical path.

All the predictors used in this study are direct mapped, meaning the n least significant bits of a load instruction's PC (that are not always zero) are used as an index into the predictor to select one of the 2^n predictor lines. Note that load-value predictors are indexed using the PC of the load instruction as opposed to conventional caches which use the effective address. Hence, the index for a predictor with 2^n lines is computed as follows:

$$index(PC_{\text{load\_instr}}) = (PC_{\text{load\_instr}} \gg 2)\%2^{\text{n}}$$

This is one of the simplest and fastest meaningful hash functions. Shifting right by two bit positions eliminates the two least significant bits that are always zero because instructions are word-aligned in the processor we use. Utilizing a more complex hash function may result in less aliasing but will most likely increase the critical-path length. Since direct-mapping results in only a little aliasing, even with moderate predictor sizes, this simple but effective hash function is used throughout the literature [10], [11], [16], [17], [25], [28].

In cache terminology, direct mapping implies the presence of tags. However, predictors do not have to be correct all the time and tags are therefore not mandatory. Due to the small amount of aliasing, load-value predictors often only incorporate partial tags or no tags at all to reduce their size. If two or more load instructions alias (i.e., they have the same index), they are forced to share a line in the predictor and may evict each other's information.

The generic load-value predictor from Fig. 1 can be tailored to exploit different kinds of load-value locality by selecting the kind of information that is retained and the computation that is performed with this information. The following sections describe possible implementations of five basic load-value predictors that exploit last value, register value, stride 2-delta, finite context method, and last four value locality. The last section discusses confidence estimators, which represent an important additional component in hardware-based load-value predictors.

2.1 Last Value Predictor

The *last value* predictor [10], [17] always predicts that a load instruction will load the same value that it did the previous time it was executed. Hence, the only information that needs to be stored in the predictor is the most recently loaded value. Predictions retrieve this value and updates store the true load value in the predictor to make it available for the next prediction.

The last value predictor's operation can formally be described as follows, where the numeric subscripts indicate the size in number of bits, "ld" refers to the load instruction being predicted or updated, "p_value" is the predicted value, and "u_value" is the update value. The first line, which describes the predictor, lists the fields making up a predictor line inside the curly brackets, followed by the name of the predictor and the number of predictor lines. In this case, the LV predictor's lines contain a single field, called "last_value," that is 64 bits wide and there are 2^{n} such lines, where n is a positive integer.

$$\text{predictor} : \{\text{last\_value}_{64}\}\ \text{LV}[2^{\text{n}}]$$

$$\begin{aligned}\text{prediction} &: \text{p\_value} = \text{LV}[index(\text{PC}_{\text{ld}})].\text{last\_value}\\ \text{update} &: \text{LV}[index(\text{PC}_{\text{ld}})].\text{last\_value} = \text{u\_value}\end{aligned}$$

The last value predictor is only able to predict sequences of repeating values (e.g., 3, 3, 3, 3), which occur surprisingly frequently, as Table 1 indicates. Such sequences stem from load instructions that repeatedly load the same runtime constant such as the starting address of a data structure or a floating-point constant.

2.2 Register Value Predictor

The *register value* predictor [27] is even simpler than the last value predictor. It always predicts that the target register of the load instruction (the register that is about to receive the loaded value) already contains the correct load value, i.e., that the load instruction is an NOP. No values have to be stored in the predictor. However, in Section 2.6, we will see that this predictor still needs to store some information to work well.

$$\text{predictor} : \{\}\ \text{Reg}[2^{\text{n}}]$$

$$\begin{aligned}\text{prediction} &: \text{p\_value} = \text{CPU.register}[target(\text{ld})]\\ \text{update} &: \text{no operation}\end{aligned}$$

Which sequences of load values the register value predictor can predict depends on the register allocator. Since none of our benchmark programs were compiled with load-value prediction in mind, any predictable sequences are artifacts of the code generation process and do not necessarily have a guiding principle behind them. It is, however, possible to significantly improve the register value predictability by modifying the register allocator [27]. Compiler support is also able to boost other types of value locality [2].

2.3 Stride 2-Delta Predictor

The *stride* predictor [10] computes the predicted value and is therefore able to predict never-before-seen values. In its conventional form, this predictor stores the last value along with the difference (called the stride) between the last and the second to last loaded value. The stride is added to the last value when a prediction is made to form the predicted value. Once the true load value is available, the predictor's

stride field is updated to reflect the difference between the last value (which is stored in the predictor) and the true load value. Then, the last value in the predictor is overwritten with the true load value. Since about 98 percent of all observed strides fall within the range of -128 to 127 [23], eight bits per predictor line are sufficient to capture almost all strides.

Unfortunately, the normal stride predictor makes two mispredictions at every transition from one predictable sequence to another. This is a problem in practice because programs fetch a surprisingly large number of short sequences of repeating values [3].

To remedy this shortcoming, a more sophisticated version of this predictor has been proposed called the *stride 2-delta* predictor [24]. The 2-delta refers to the fact that this predictor retains two strides. The first stride is identical to the one found in the conventional stride predictor. The second stride is only updated if the same stride has been seen at least twice in a row. The second stride is used for making predictions.

Of course, the second stride field also only needs to be eight bits wide. In the pseudocode describing the stride 2-delta predictor below, the function $lsb_{0...7}(x)$ extracts the eight least significant bits of x. Unless otherwise noted, all stride-predictor results in this study refer to the stride 2-delta predictor.

$$\begin{aligned}
\text{predictor} &: \{\text{last_value}_{64}, \text{stride1}_8, \text{stride2}_8\}\ \text{St2d}[2^n]\\
\\
\text{prediction} &: \text{p_value} = \text{St2d}[index(\text{PC}_{\text{ld}})].\\
&\qquad \text{last_value} + \text{St2d}[index(\text{PC}_{\text{ld}})].\text{stride2}\\
\text{update} &: \text{temp} = lsb_{0...7}(\text{u_value} - \text{St2d}[index(\text{PC}_{\text{ld}})].\\
&\qquad\quad \text{last_value});\\
&\quad \text{if}(\text{temp} == \text{St2d}[index(\text{PC}_{\text{ld}})].\text{stride1})\\
&\qquad \text{St2d}[index(\text{PC}_{\text{ld}})].\text{stride2} = \text{temp};\\
&\quad \text{St2d}[index(\text{PC}_{\text{ld}})].\text{stride1} = \text{temp};\\
&\quad \text{St2d}[index(\text{PC}_{\text{ld}})].\text{last_value} = \text{u_value}
\end{aligned}$$

The stride 2-delta predictor can predict sequences of repeating values that have a stride of zero. In addition, it can predict sequences that exhibit genuine stride behavior (e.g., -4, -2, 0, 2, 4). Such sequences are, however, not very frequent [10], [24] because register allocation assigns induction variables to registers, but they do occur, for example, when a program uses global variables as counters.

2.4 Last Four Value Predictor

The *last four value* predictor [6], [16] is similar to the last value predictor except every predictor line retains the four most recently loaded values instead of only the most recent value. Storing more than just the last value has been shown to improve performance even when scaling predictors to the same overall size [6]. The last four value predictor essentially consists of four independent last value predictors operating in parallel and a metapredictor that selects which of the four predictions to use. The operation of the metapredictor and the corresponding *select* function (see below) are discussed in Section 2.6.

$$\begin{aligned}
\text{predictor} &: \{\text{value_1}_{64}, \text{value_2}_{64}, \text{value_3}_{64}, \text{value_4}_{64}\}\ \text{L4V}[2^n]\\
\\
\text{prediction} &: \text{p_value} = select(\text{L4V}[index(\text{PC}_{\text{ld}})].\text{value_1},\\
&\qquad\qquad \text{L4V}[index(\text{PC}_{\text{ld}})].\text{value_2},\\
&\qquad\qquad \text{L4V}[index(\text{PC}_{\text{ld}})].\text{value_3},\\
&\qquad\qquad \text{L4V}[index(\text{PC}_{\text{ld}})].\text{value_4})\\
\text{update} &: \text{L4V}[index(\text{PC}_{\text{ld}})].\text{value_4} =\\
&\qquad \text{L4V}[index(\text{PC}_{\text{ld}})].\text{value_3};\\
&\quad \text{L4V}[index(\text{PC}_{\text{ld}})].\text{value_3} =\\
&\qquad \text{L4V}[index(\text{PC}_{\text{ld}})].\text{value_2};\\
&\quad \text{L4V}[index(\text{PC}_{\text{ld}})].\text{value_2} =\\
&\qquad \text{L4V}[index(\text{PC}_{\text{ld}})].\text{value_1};\\
&\quad \text{L4V}[index(\text{PC}_{\text{ld}})].\text{value_1} = \text{u_value}
\end{aligned}$$

In addition to the sequences that the last value predictor can predict, the last four value predictor is able to predict sequences of alternating values (e.g., -1, 0, -1, 0, -1) or, more generally, any repeating pattern of four or fewer values (e.g., 1, 2, 3, 1, 2, 3, 1, 2, 3). Such sequences occur more frequently than true stride behavior [6], [16]. In particular, alternating sequences of values arise relatively often when variables toggle between two values.

2.5 Finite Context Method Predictor

The most sophisticated nonhybrid predictor we investigate is the *finite context method* predictor [24], [25]. It retains the last four loaded values in every predictor line. However, since these values are only used to compute an index into the predictor's second level (a lookup table), they do not have to be stored in their full length but can be stored in a more compact, preprocessed form. The second level, a 2,048-entry cache, stores the values that follow every seen sequence of last four values (modulo the table size). Since the second level is shared, load instructions can communicate information to one another in this predictor. Hence, after fetching an arbitrary sequence of load values, any load instruction fetching the same sequence can be predicted correctly as long as the predictor entries have not been overwritten. For this study, we fix the size of the direct mapped, tagless second-level table of the finite context method predictor at 2,048 entries. The index into the second level (the *line*) is computed as follows:

$$\begin{aligned}
hash(\text{val}) &= \text{val}_{63...56} \oplus \text{val}_{55...48} \oplus \text{val}_{47...40} \oplus \text{val}_{39...32} \oplus\\
&\qquad \text{val}_{31...24} \oplus \text{val}_{23...16} \oplus \text{val}_{15...8} \oplus \text{val}_{7...0}\\
index2(\text{val1}, \text{val2}, \text{val3}, \text{val4}) &= hash(\text{val1}) \oplus hash(\text{val2})^*2\\
&\qquad \oplus hash(\text{val3})^*4 \oplus hash(\text{val4})^*8\\
\text{line} &= index2(\text{value_1}, \text{value_2}, \text{value_3}, \text{value_4})
\end{aligned}$$

The symbol "$\oplus$" in the above formulae represents the logical exclusive-or function. The $index2$ function is similar to the select-fold-shift-xor hash-functions used in the current literature for the finite context method predictor [21], [23], [25]. It utilizes all 64 bits of the four load values for computing the index. Furthermore, the values are shifted relative to one another so that sequences of constant values do not cancel each other out (i.e., always yield an index of zero) when they are exclusively-ored.

Another benefit of the above function is that part of it (i.e., $hash(\text{val})$) can be evaluated before the information is inserted into the first predictor level. Since $hash(\text{val})$ always yields an eight-bit result, each line in the first level of the predictor only needs to store four eight-bit values instead of four 64-bit values, reducing the size of the finite context

method predictor substantially. The following pseudocode describes the resulting predictor. The symbol "$\diamondsuit$" indicates composition.

$$
\begin{aligned}
&\text{level1} : \{\text{hash\_1}_8, \text{hash\_2}_8, \text{hash\_3}_8, \text{hash\_4}_8\}\ \text{FCM1}[2^n]\\
&\text{level2} : \{\text{value}_{64}\}\ \text{FCM2}[2048]\\
&\text{predictor} : \text{level1} \diamondsuit \text{level2}
\end{aligned}
$$

$$
\begin{aligned}
\text{prediction} : \text{line} = {}& \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_1} \oplus\\
& \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_2}{*}2 \oplus\\
& \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_3}{*}4 \oplus\\
& \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_4}{*}8;\\
\text{p\_value} = {}& \text{FCM2}[\text{line}].\text{value}
\end{aligned}
$$

$$
\begin{aligned}
\text{update} : \text{line} = {}& \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_1} \oplus\\
& \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_2}{*}2 \oplus\\
& \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_3}{*}4 \oplus\\
& \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_4}{*}8;\\
& \text{FCM2}[\text{line}].\text{value} = \text{u\_value};\\
& \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_4} =\\
& \ \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_3};\\
& \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_3} =\\
& \ \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_2};\\
& \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_2} =\\
& \ \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_1};\\
& \text{FCM1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{hash\_1} = \mathit{hash}(\text{u\_value})
\end{aligned}
$$

Finite context method predictors can predict long, reoccurring sequences of arbitrary values (e.g., 3, 7, 4, 9, 2, ..., 3, 7, 4, 9, 2). These sequences arise, for instance, during the repeated traversal of dynamic data structures. Note that FCM predictors can also predict constant and alternating sequences and sequences exhibiting stride behavior as long as the sequences repeat and their lengths do not exceed the size of the predictor's second-level table [24].

2.6 Confidence Estimation

A substantial fraction of the executed load instructions cannot be correctly predicted with the currently known prediction approaches. Attempting to predict these loads will inevitably result in mispredictions. Because recovering from mispredictions takes time, a high misprediction rate can incur a slowdown that more than eradicates the speedup from correct predictions. Hence, it is possible for a load-value predictor to decelerate the processor instead of speeding it up.

To keep the number of mispredictions at a minimum while leaving the correct predictions intact, almost all load-value predictors incorporate some form of *confidence estimator* to identify predictions that are likely to be incorrect so that they can be inhibited [1], [8], [17], [21], [23], [25], [27], [28]. Inhibiting such predictions reduces the number of mispredictions and the associated recovery cost and, hence, improves the predictor's overall performance.

One way of estimating the likelihood of a correct prediction is to look for discernable patterns in the predictability of a load instruction. The intuition is that the recent behavior often indicates what will happen next. For example, if a load was predictable every other time it was executed in the recent past, there is a good chance that the outcome of the next prediction will be the same as the outcome of the second to last prediction.

The *SAg* confidence estimator exploits such predictability patterns [3]. It stores the predictability history of each load in a bit-pattern in which the nth bit reflects the predictability of the nth to last execution. Usually, a one encodes a predictable value and a zero an unpredictable value.

Whenever the memory returns a load value, this value is compared with the predicted value (even if the prediction was not used) and the outcome of this comparison is shifted into the history, whereby the oldest bit is lost.

In order to use such histories as a measure of confidence, it is essential to know which ones are frequently followed by a correct prediction. The *SAg* confidence estimator uses saturating counters to record the number of predictable values that follow each possible history pattern. Predictions are only allowed if the counter value associated with the current prediction history is above a preset threshold. Thus, the counters dynamically assign a confidence to each history and continuously adjust which patterns should trigger a prediction and which ones should not.

The following pseudocode describes the operation of the *SAg* confidence estimator, which is named after the structurally identical *SAg* branch predictor [29]. $\boldsymbol{m}$ denotes the number of history bits in each line and $\boldsymbol{x}$ represents the number of bits in each saturating counter.

$$
\begin{aligned}
&\text{level1} : \{\text{history}_{\text{m}}\}\ \text{SAg1}[2^n]\\
&\text{level2} : \{\text{count}_{\text{x}}\}\ \text{SAg2}[2^m]\\
&\text{conf\_estim} : \text{level1} \diamondsuit \text{level2}
\end{aligned}
$$

$$
\begin{aligned}
\text{prediction} : \text{predict} = {}& (\text{SAg2}[\text{SAg1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{history}].\\
& \text{count} >= \mathbf{threshold})\\
\text{update} : {}& \text{SAg2}[\text{SAg1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{history}].\text{count} =\\
& (\text{predicted\_value} == \text{true\_value})\ ?\\
& \mathit{min}(\mathbf{top} - 1, \text{SAg2}[\text{SAg1}[\mathit{index}(\text{PC}_{\text{ld}})].\\
& \quad \text{history}].\text{count} + 1)\ :\\
& \mathit{max}(0, \text{SAg2}[\text{SAg1}[\mathit{index}(\text{PC}_{\text{ld}})].\\
& \quad \text{history}].\text{count-}\mathbf{penalty});\\
& \text{SAg1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{history} =\\
& \ \mathit{LSB}_{0\ldots m-1}(\text{SAg1}[\mathit{index}(\text{PC}_{\text{ld}})].\text{history} \ll 1) |\\
& \ (\text{predicted\_value} == \text{true\_value})\ ?\ 1 : 0
\end{aligned}
$$

The *threshold*, *top*, *penalty*, and m values are parameters of the *SAg* confidence estimator. The best setting for these parameters depends with varying degrees on the load-value predictor, the programs, and the recovery mechanism used.

Note that the first level of the *SAg* confidence estimator is normally merged with the first level of the load-value predictor. Therefore, load-value predictors with confidence estimators have an additional field in each line, which is why the Reg predictor is not completely storageless.

We can now explain the *select* function used in the last four value predictor (Section 2.4). It simply picks the component that reports the highest confidence, prioritizing younger values in case of a tie.

TABLE 2
Functional Unit and Memory Latencies (In Cycles)

Operation	Latency
integer multiply	8-14
conditional move	2
other int and logical	1
floating point multiply	4
floating point divide	16
other floating point	4
L1 load-to-use	1
L2 load-to-use	12
memory load-to-use	80

3 Evaluation Methods

All measurements pertaining to this study are based on the Alpha AXP architecture [9]. The performance of the various load-value predictors is evaluated using the AINT simulator [19] with a cycle-accurate, superscalar back-end that runs native Alpha binaries. The simulator is configured to emulate a high-performance microprocessor similar to the Alpha 21264 [14]. It accurately models the processor's internal timing behavior, resource constraints, and speculative execution, as well as the memory hierarchy and latencies. Bus-contention is not modeled.

The simulated CPU is four-way superscalar, issues instructions out-of-order from a 128-entry instruction window, has a 32-entry load/store buffer, four integer and two floating-point units, a 64KB two-way set associative L1 instruction-cache, a 64KB two-way set associative L1 data-cache, a 4MB unified direct-mapped L2 cache, a 4,096-entry branch target buffer (BTB), and a 2,048-line hybrid gshare-bimodal branch predictor. The caches have a block size of 32 bytes. Table 2 summarizes the modeled latencies.

The six functional units are fully pipelined and each unit can execute all operations in its class. Operating system calls are executed but not simulated, which should not be a problem since the benchmark programs we use perform few operating system calls [26]. Loads can only execute when all prior store addresses are known. Up to four load instructions are able to issue per cycle. This CPU represents the baseline processor. All reported speedups are relative to this CPU, which does not contain a load-value predictor.

In the CPUs that include a load-value predictor, predictions are initiated in the rename stage of the instruction pipeline and have a two-cycle latency. Because of the two levels of the *SAg* confidence estimator, all investigated predictors are pipelined over two stages. To support up to four predictor accesses per cycle, the predictors are divided into four independent banks that operate in parallel [3]. Each bank represents an independent predictor one quarter the total size. Since the modeled CPU fetches naturally aligned four-tuples of instructions, it is not possible to fetch two load instructions during the same cycle that go to the same predictor bank. Only one access per bank is allowed per cycle. Updates have a lower priority than predictions and are queued in an eight-entry FIFO buffer (one per bank). When the buffer is full, further updates are dropped, which rarely happens [3].

The predictors are updated as soon as the true load value becomes available (i.e., when the verification memory access completes), predictions do not speculatively update the predictor's state, out-of-date predictions are made as long as there are pending updates (for the same predictor line), and out-of-order and wrong-path updates of the predictor are accurately modeled in the simulator. Conditional branches whose outcome depends on a predicted load value may select the wrong path or be mispredicted due to a load-value misprediction. All predictor updates are final.

We study the performance of load-value predictors in the presence of two different misprediction-recovery mechanisms. The simpler but less powerful *refetch* mechanism is the one used for recovering from branch mispredictions [10]. When a misprediction is detected in this scheme, all the instructions that follow the mispredicted instruction are purged from the instruction window and the processor state is reset to the point of the last nonspeculative instruction. The CPU then continues processing instructions by fetching the next instruction, that is, the instruction that immediately follows the mispredicted load. Refetch recovery incurs a cycle-penalty because it takes time to purge instructions from the instruction window, to restore the CPU's state, to refetch instructions, and to drain functional units.

Unfortunately, in this scheme, instructions whose results are correct are sometimes purged. For example, if instruction X is independent of an earlier load instruction L, an out-of-order processor may execute X before the load has completed. Because X is independent of L, its result does not depend on the load value and will therefore be correct. Consequently, purging X is not necessary even if L is mispredicted.

In fact, mispredicting L does not even invalidate the instructions that do depend on L (up to the first conditional branch instruction whose target depends on L). In the worst case, these instructions have executed with an incorrect input value. If the affected instructions remain in the instruction window, it suffices to reexecute them with the correct input value [16]. Hence, after a misprediction, the directly and indirectly dependent instructions only need to be issued again. The second (or subsequent) execution will produce the correct result because the input operands are now correct. We refer to this misprediction recovery mechanism as *reexecute* recovery.

While the reexecute mechanism avoids the unnecessary purging of independent instructions and the overhead of refetching already fetched instructions, it still incurs a penalty for identifying the dependent instructions, changing their state, reexecuting them, and draining functional units. However, the penalty is considerably smaller than the refetch penalty.

3.1 Benchmarks

We use the eight SPECint95 programs [26] as our benchmark suite. These programs are well-understood, nonsynthetic, and compute-intensive, which is ideal for processor performance evaluations. The SPECint95 programs are written in C and perform the following tasks:

TABLE 3
Information about the SPECint95 Benchmark Suite

Information about the SPECint95 Benchmark Suite													
	static			dynamic			skipped	simulated			base	load miss-rate	
program	insts	loads	%lds	insts	loads	%lds	instrs	instrs	loads	%lds	IPC	%L1	%L2
compress	22 k	4 k	(17.9)	60,156 M	10,537 M	(17.5)	5.6 G	300.0 M	53.5 M	(17.8)	1.338	11.72	6.17
gcc	337 k	73 k	(21.6)	334 M	80 M	(23.9)	0.0 G	334.1 M	79.7 M	(23.9)	1.510	2.39	6.44
go	81 k	16 k	(20.1)	35,971 M	8,764 M	(24.4)	7.0 G	300.0 M	72.1 M	(24.0)	1.414	1.62	15.72
ijpeg	70 k	14 k	(19.8)	41,579 M	7,141 M	(17.2)	2.0 G	300.0 M	49.5 M	(16.5)	1.498	2.31	65.20
li	37 k	7 k	(18.2)	66,613 M	17,792 M	(26.7)	5.0 G	300.0 M	86.4 M	(28.8)	1.911	4.13	0.67
m88ksim	51 k	9 k	(17.4)	82,810 M	14,849 M	(17.9)	2.0 G	300.0 M	62.1 M	(20.7)	1.258	0.13	11.21
perl	105 k	21 k	(20.3)	19,934 M	6,207 M	(31.1)	1.0 G	300.0 M	93.5 M	(31.2)	1.567	0.00	46.87
vortex	161 k	32 k	(20.0)	95,791 M	22,471 M	(23.5)	7.0 G	300.0 M	71.0 M	(23.7)	2.922	2.16	11.99
average	108 k	22 k	(20.4)	50,399 M	10,980 M	(21.8)	3.7 G	304.3 M	71.0 M	(23.3)	1.677	3.06	20.53

- **compress**: compresses and decompresses a file in memory.
- **gcc**: C compiler that generates SPARC instructions.
- **go**: artificial intelligence, plays the game of Go.
- **ijpeg**: graphic compression and decompression.
- **li**: Lisp interpreter.
- **m88ksim**: Motorola 88000 chip simulator, runs a test program.
- **perl**: manipulates strings (anagrams) and prime numbers in Perl.
- **vortex**: an object-oriented database program.

Except for *gcc*, we use the reference inputs for all programs. Only the *varasm* input is used with *gcc* because our simulation infrastructure only supports one input per program. To avoid possible side effects that may be attributed to poor code quality, the peak versions of the programs are utilized which were compiled with DEC GEM-CC on a DEC Alpha 21164 using the highest optimization level "*-migrate -O5 -ifo*." The optimizations include common subexpression elimination, split-lifetime analysis, code scheduling, NOP insertion, code motion and replication, loop unrolling, software pipelining, local and global inlining, interfile optimizations, etc. In addition, the binaries are statically linked to combine the global segments, which reduces the number of runtime constants that are loaded.

The few floating-point load instructions contained in the binaries are also predicted and loads to the zero-registers (R31 and F31) as well as load-address instructions (LDA and LDAH) are ignored since they do not read from the memory.

Table 3 summarizes relevant information about the SPECint95 programs. It shows the static number of instructions and load instructions contained in the binaries, the number of instructions and load instructions executed during complete runs, the number of skipped instructions before the detailed simulations are started (in billions), the number of simulated (committed) instructions, and the instructions per cycle (IPC) and the cache read miss-rates of the baseline processor on the simulated segments. The numbers in parentheses indicate the percentage of instructions that are loads. The static counts are in thousands and the dynamic counts in millions. The averages are arithmetic means.

Table 3 shows that all eight binaries contain several thousand load instructions. Despite the high optimization level, the percentage of load instructions is quite high. About every fifth static instruction and every fifth executed instruction is a load.

Each benchmark program is executed for about 300 million instructions on the cycle-accurate simulator to keep the simulation time reasonable. Before the detailed measurements commence, the simulator skips over the initialization code of each program. Doing so is important when only a fraction of a program's execution can be simulated because the initialization is not usually representative of the general program behavior [21]. No instructions are skipped with *gcc* and it is executed for 334 million instructions since this amounts to the complete compilation of the *varasm* input-file. Each simulated segment contains more than 49 million executed load instructions, which should be sufficient to render any warm-up effects in the load-value predictors negligible.

The fast-forward points were carefully hand-selected to make the simulated segments as representative of the whole programs as possible. We chose a segment length of 300 million instructions since this appears to be enough to capture the "average" program behavior. Longer segments do not yield significantly different results. Care was taken to match the percentage of executed instructions that are loads and, particularly, the predictability of the eight segments with the respective numbers for the whole program executions [3]. Only for *li* and *m88ksim* was the search for a representative segment was not very successful. Fortunately, *li*'s segment exhibits too low a predictability and *m88ksim*'s too high a predictability, making the average over the eight programs very close to the average over the complete execution of the entire benchmark suite.

With the exception of *compress*, the benchmark programs do not have high L1 data-cache load miss-rates, making it hard for a load-value predictor to be effective. Some of the L2 load miss-rates are, on the other hand, quite large. However, since the corresponding number of cache accesses is very small (not shown), the large L2 miss-rates do not have a significant impact on the performance.

TABLE 4
SAg Configurations Yielding
the Highest Harmonic-Mean Speedup

		SAg confidence estimator			
		hist bits	cntr top	threshold	penalty
re-fetch	FCM	10	16	15	11
	L4V	10	16	15	8
	LV	10	16	13	5
	Reg	10	16	15	7
	St2d	10	16	12	5
re-execute	FCM	10	8	6	3
	L4V	10	8	7	3
	LV	10	8	5	2
	Reg	10	8	4	1
	St2d	10	8	5	1

4 Results

To determine the performance of the five basic predictors from Section 2, we outfitted them with *SAg* confidence estimators (Section 2.6) and measured how much they speed up the simulated CPU (Section 3). Note that we use the harmonic-mean speedup over the eight SPECint95 programs as the performance metric throughout this paper. For the sake of brevity, we cannot show the performance of individual programs. However, it should be noted that the speedup of a single program can be quite different from the mean over the benchmark suite [3].

Based on previous studies [3], [5], we decided to use 10-bit histories in the confidence estimators and a top value for the saturating counters of 16 for refetch recovery and eight for reexecute. A global parameter-space search was performed to find the optimal threshold and penalty values for each predictor. Table 4 lists the resulting configurations.

Note that the penalties yielding the best performance with a reexecute misprediction recovery mechanism are lower than those for refetch, even when accounting for the wider refetch counters. This is a direct reflection of the lower misprediction penalty with reexecute.

Fig. 2 shows the speedups delivered by the five predictors for both recovery mechanisms. Each predictor comprises 2,048 lines divided into four banks. The FCM predictor has an additional 2,048 lines in the second level, which is also divided into four banks. Note that the five predictors differ considerably in their sizes.

As expected, all five predictors perform better with reexecute than with refetch. The difference in speedup is very small for the Reg predictor because it exhibits the most regular predictability patterns of the five predictors, which results in the most accurate confidence estimations and, therefore, the smallest number of mispredictions.

The FCM predictor exhibits the largest difference between refetch and reexecute. This predictor makes the most mispredictions with refetch, resulting in a substantial recovery cost that keeps the speedup low.

4.1 Metric Anomalies

In measuring the performance of the predictors, we noticed several anomalies that suggest that the prediction rate does not always characterize a predictor's quality. While some of the five predictors predict up to 45 percent of the dynamically executed loads correctly, the Reg predictor only correctly predicts between 11 and 15 percent of all the loads. In spite of this very low prediction rate, it yields a respectable speedup. One might correctly speculate that the loads the Reg predictor is able to predict are somehow more important than the loads the other four basic predictors predict. The loads predicted by Reg have a substantially longer average latency than the ones predicted by the other predictors, as the shaded columns in Table 5 illustrate. For example, the loads predicted by the Reg predictor have an average latency of over 20 cycles both with refetch and reexecute recovery, whereas the St2d's loads only have a latency of 12.5 cycles for refetch and about 15 for reexecute.

Evidently, the number of correctly predicted loads does not adequately predict the delivered performance. Rather, the latency of the predicted loads needs to be taken into account.

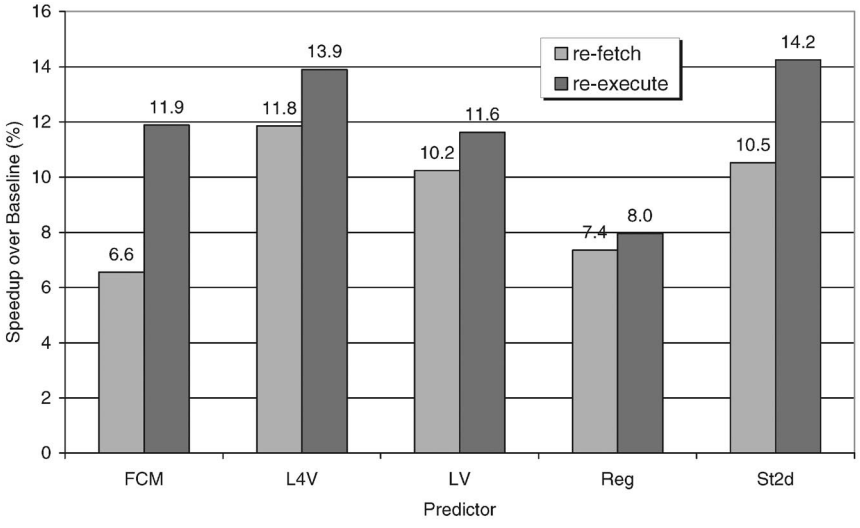


Fig. 2. Refetch and reexecute speedup of five predictors.

TABLE 5
Latency and Cycles to First Usage of the Predicted Load Values

	re-fetch		re-execute	
	latency	usage	latency	usage
FCM	14.6	3.8	16.6	4.0
L4V	14.6	4.9	16.8	5.7
LV	15.2	5.0	17.2	5.7
Reg	20.4	3.9	20.2	4.9
St2d	12.7	3.0	15.4	3.2
average	15.5	4.1	17.2	4.7

Table 6 demonstrates a more striking example of a metric anomaly. It shows that, while the FCM makes fewer correct predictions, more incorrect predictions, attempts fewer predictions, and has a lower prediction rate and accuracy than LV with reexecute, the FCM still outperforms the LV speedup-wise. To ensure that this result is not an artifact of the averaging of the speedups, we show the harmonic, geometric, and arithmetic mean speedup as well as the average IPC (instructions per cycle) improvement over the eight SPECint95 programs. All four ways of averaging the measured speedups yield the same result, i.e., FCM performs better than LV. Since even the average load latency (Table 5) is in favor of the LV, there has to be another factor that influences the performance.

The nonshaded columns of Table 5 offer a possible explanation. The average time to the first usage of a predicted load value is much lower for the FCM (4.0 cycles) than it is for the LV (5.7 cycles), meaning that the FCM's predictions are needed sooner and are therefore more important than the LV's. Hence, it looks like the time to the first use of a predicted load value also needs to be accounted for to properly establish a load-value predictor's performance.

4.2 Hybrid Performance

Combining multiple load-value predictors does not always result in a predictor that can predict more load instructions or make more accurate predictions. For example, the stride 2-delta predictor can make last-value predictions. Consequently, combining it with a last-value predictor will probably not yield a hybrid that is more effective than the stride 2-delta predictor by itself.

Another reason why a combined predictor may not perform better than its components is the selector, which has to choose one of the multiple component predictors for making a prediction and therefore represents a new source of errors. We use the confidence estimators to guide the selection process by selecting the component with the highest confidence [21], [23]. (Note that the selected component is only allowed to make a prediction if its confidence is above the preset threshold.)

The components in the hybrid predictors discussed in this section are prioritized to resolve selector ties. If only one component reports the highest confidence, then that component is selected, regardless of its priority. Since changing the priority among the components of a hybrid does not appear to affect the performance considerably [3], we only investigate hybrids in which the components are prioritized in the following, arbitrary order (from high priority to low priority): Reg, LV, St2d, L4V, FCM.

To determine which predictors complement each other well and yield good hybrids, we tested every possible combination between a register value, last value, stride 2-delta, last four value, and finite context method predictor. However, because the last four value predictor is a strict superset of the last value predictor (assuming the same number of lines and the same CE configuration), we exclude hybrid combinations that include both an LV and an L4V predictor. The performance of the excluded hybrids is identical to the performance of the same predictor without the (redundant) LV component.

Since our goal is to study which predictors complement each other well, all components comprise 2,048 lines regardless of the resulting hybrid's overall size. We chose this height because it results in reasonable predictor sizes and performance. While the resulting size of some of the hybrids is rather large, they can frequently be made smaller by sharing state between their components [3], [4], [20]. Nevertheless, due to the varying predictor sizes, care must be taken when using the performance numbers shown in this section for interhybrid comparisons.

Fig. 3 shows the performance of all hybrid combinations with a refetch misprediction recovery mechanism. The predictors are sorted by increasing performance. The hybrid's names are character combinations in which each letter or digit represents one component: r stands for register value, l for last value, s for stride 2-delta, 4 for last four value, and f for finite context method predictor.

It is important to optimize the threshold and penalty for each predictor and recovery mechanism individually [21]. We did this for the five basic predictors, but it is not practical to separately optimize the two parameters for every hybrid. Instead, the threshold and penalty values that yield the highest average speedup over the included components are used as an approximation. They are computed as follows: We evaluated the speedup of the five basic predictors for a large number of threshold and penalty pairs and recorded the results in speedup maps [3]. A speedup map is a matrix with different thresholds in one dimension and different penalties in the other dimension. The matrix elements

TABLE 6
Various Metrics Showing Anomaly

	% correct predictions	% no predictions	% wrong predictions	prediction rate (%)	accuracy (%)	mean speedup over baseline (%) harmonic	geometric	arithmetic	IPC
FCM	34.71	61.34	3.95	38.66	89.78	11.88	14.89	18.55	14.60
LV	40.28	57.26	2.46	42.74	94.24	11.63	13.36	15.59	12.47

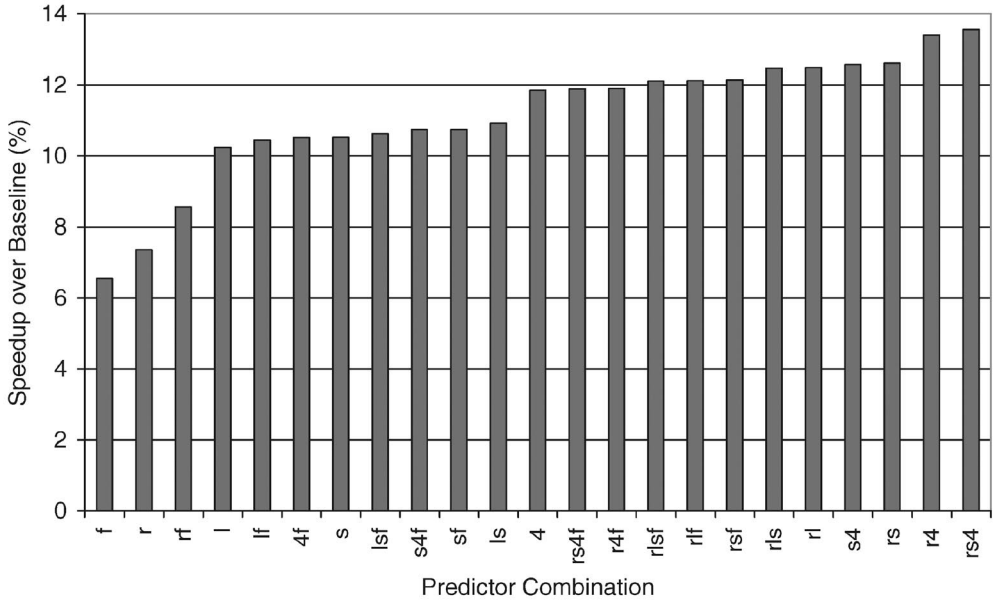


Fig. 3. Hybrid performance with refetch recovery.

are the speedups measured for the threshold and penalty that intersect at that element. We then computed an average map by forming the arithmetic mean of the entries in the individual maps of each component in a given hybrid (e.g., the register value predictor's map and the last value predictor's map for the $\mathrm{Reg} + \mathrm{LV}$ hybrid). The highest speedup in the averaged map determined the threshold and penalty value we used for each hybrid. Note that this approach does not always yield the best performance but is usually close. For example, the $\mathrm{St2d} + \mathrm{FCM}$ hybrid yields a speedup of 9.99 percent with refetch and 13.09 percent with reexecute when using the parameters from the averaged speedup map, whereas truly optimizing the threshold and penalty results in a speedup of 10.01 percent for refetch and 13.94 percent for reexecute. Table 7 shows the confidence-estimator configurations derived from the averaged speedup maps. All histories are 10 bits long and the counters' top values are fixed at 16 for refetch and eight for reexecute.

Hybrids with more components tend to yield a higher speedup than the ones with fewer components. However, there are many notable exceptions. For instance, the speedup of the best performing predictor with refetch $(\mathrm{Reg} + \mathrm{St2d} + \mathrm{L4V})$ decreases when adding an FCM component to it. Likewise, $\mathrm{Reg} + \mathrm{L4V}$, $\mathrm{Reg} + \mathrm{St2d}$, $\mathrm{St2d} + \mathrm{L4V}$, $\mathrm{Reg} + \mathrm{LV}$, $\mathrm{Reg} + \mathrm{LV} + \mathrm{St2d}$, L4V, and $\mathrm{LV} + \mathrm{St2d}$ all suffer when an FCM component is included. Only the Reg, the LV, and the St2d predictors benefit from an FCM. This is clearly a result of the poor performance of the *SAg* confidence estimator in connection with the FCM predictor. It is possible that hybridizing an FCM with a different confidence estimator would result in better performance. Such an investigation is left for future work.

Aside from the FCM component, there also exist other irregularities in the refetch speedups. For example, the $\mathrm{Reg} + \mathrm{St2d}$ predictor outperforms the $\mathrm{Reg} + \mathrm{LV} + \mathrm{St2d}$ predictor. The $\mathrm{Reg} + \mathrm{L4V} + \mathrm{FCM}$, the $\mathrm{Reg} + \mathrm{LV} + \mathrm{FCM}$, and the $\mathrm{Reg} + \mathrm{LV}$ predictors do not benefit from having an St2d component added to them. Furthermore, the speedups of the $\mathrm{Reg} + \mathrm{St2d} + \mathrm{FCM}$ and the $\mathrm{St2d} + \mathrm{FCM}$ predictors decrease when adding an L4V component. The reason for this counterintuitive behavior is *negative interference*.

Because adding a component to a hybrid makes the task of the selector harder (there are more choices), it can happen that the added predictability provided by the new component is unable to offset the increased selector-related losses. When this situation occurs, the hybrid's components interfere negatively with one another and lower the overall performance. In previous papers, we provided performance results for perfect selectors which show that the selector-related losses can be substantial [3], [4], [6].

Note that some of the most effective hybrids are small and have only two components ($\mathrm{Reg} + \mathrm{LV}$ and $\mathrm{Reg} + \mathrm{St2d}$). The remaining three of the five best combinations are significantly larger because they include an L4V component. However,

TABLE 7
The Confidence-Estimator Parameters of the Hybrid Predictors

		4	4f	f	l	lf	ls	lsf	r	r4	r4f	rf	rl	rlf	rls	rlsf	rs	rs4	rs4f	rsf	s	s4	s4f	sf
re-fetch	threshold	15	15	15	13	15	12	14	15	15	15	15	13	15	14	15	11	15	15	15	12	15	15	15
	penalty	8	11	11	5	9	5	6	7	8	10	10	8	9	5	7	7	7	7	7	5	7	7	7
re-exec	threshold	7	7	6	5	6	5	5	4	7	6	6	6	5	5	5	5	6	5	5	5	6	6	5
	penalty	3	3	3	2	3	1	2	1	3	3	3	1	2	2	2	1	1	2	2	1	1	2	2

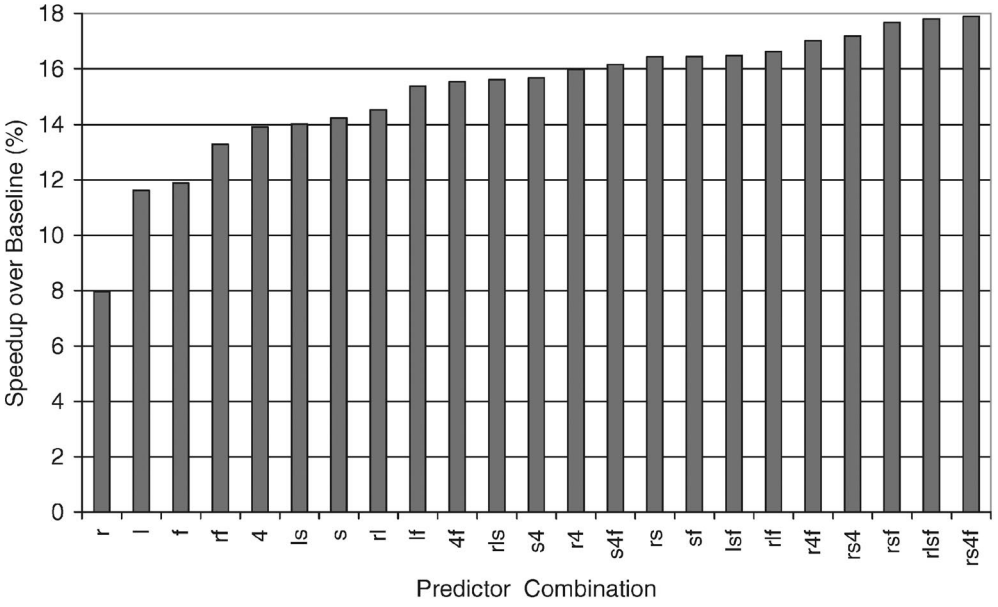


Fig. 4. Hybrid performance with reexecute recovery.

previous work by the authors illustrates how the size of the $\mathrm{Reg} + \mathrm{St} + \mathrm{L4V}$ predictor can be reduced to only slightly more than that of the $\mathrm{Reg} + \mathrm{St2d}$ hybrid essentially without loss of performance [4].

Eleven of the 12 best-performing hybrids include the small register value predictor, indicating that the Reg predictor is a very important component in a hybrid. This result is particularly surprising because the Reg predictor by itself performs rather poorly. Note that no profiling was used to improve the register allocation, which can significantly enhance the performance of this predictor [27], yet the benefit from including a register value predictor is already substantial.

Fig. 4 shows the performance of all hybrid combinations with a reexecute misprediction recovery mechanism. Again, the predictors are sorted by increasing performance.

Clearly, hybrid load-value predictors outperform even the best single-component predictors. Nevertheless, it should be noted that an oracle predictor that predicts every executed load instruction correctly yields a harmonic-mean speedup of 55.7 percent over SPECint95 (independent of the misprediction-recovery method) [4]. The speedup of the oracle is three times higher than the speedup of the best reexecute hybrid we investigated and suggests significant opportunity for novel prediction methods that exploit as of yet undetected load-value locality. It is, however, unclear how much of the remaining performance potential can be realized because the fraction of truly unpredictable loads is unknown.

There are also instances of negative interference with the reexecute misprediction-recovery mechanism. Again, the $\mathrm{L4V}$ component diminishes the performance when it is included in the $\mathrm{St2d} + \mathrm{FCM}$ predictor and the $\mathrm{Reg} + \mathrm{St2d}$ predictor suffers when an LV component is added. There is one new case of negative interference that was not present with refetch. Specifically, the $\mathrm{St2d}$ predictor outperforms the $\mathrm{LV} + \mathrm{St2d}$ hybrid with reexecute.

The negative interference related to the FCM component does not exist with reexecute. In fact, seven of the eight best performing hybrids include an FCM, six of them include an $\mathrm{St2d}$, and six include a Reg component. The best performing refetch hybrid $(\mathrm{Reg} + \mathrm{St2d} + \mathrm{L4V})$ is among the four best performing reexecute hybrids and is the only one of them that does not include an FCM, which may be significant because the FCM is a two-level predictor, whereas the other four basic predictors comprise only one level (the *SAg* confidence estimator, however, requires two levels).

When averaging the refetch and the reexecute speedups, the $\mathrm{Reg} + \mathrm{St2d} + \mathrm{L4V}$ hybrid performs best by a considerable margin. The most effective two-component hybrid is the $\mathrm{Reg} + \mathrm{L4V}$, which is closely followed by the $\mathrm{Reg} + \mathrm{St2d}$ hybrid. Finally, the best single-component predictor is $\mathrm{L4V}$ trailed by the $\mathrm{St2d}$ predictor. Surprisingly, neither of the four-component hybrids outperforms the best three-component hybrid.

Table 8 and Table 9 show the results from Fig. 3 and Fig. 4, respectively, in a different form. Both tables list the hybrids and their speedups on the left. The numbers on the right represent the speedup increase in percentage points when adding the given components to the listed hybrids.

Evidently, both with refetch and reexecute, all the hybrids that do not include a Reg component would benefit considerably from having one. This is particularly surprising because the Reg predictor does not perform well in isolation. Similarly, the Reg predictor benefits from any other component. Only Reg, FCM, and the $\mathrm{Reg} + \mathrm{FCM}$ hybrid benefit significantly from an LV component. These three predictors also profit the most from having an $\mathrm{St2d}$ or an $\mathrm{L4V}$ component added to them. As mentioned earlier, most hybrids are slowed down by an FCM component with refetch, whereas it is advantageous for most hybrids to have an FCM with reexecute. Several predictors benefit from an $\mathrm{L4V}$ component.

TABLE 8
Refetch Speedup Benefit from Adding Components

Re-fetch Speedup Benefit of Adding Components to SAg Hybrids						
hybrid	speedup	+r	+l	+s	+4	+f
r	7.4		5.1	5.3	6.0	1.2
l	10.2	2.2		0.7	1.6	0.2
rl	12.5			0.0	0.9	-0.4
s	10.5	2.1	0.4		2.0	0.2
rs	12.6		-0.1		0.9	-0.5
ls	10.9	1.5			1.6	1.2
rls	12.5				1.1	-0.4
4	11.8	1.5	0.0	0.7		-1.3
r4	13.4		0.0	0.2		-1.5
s4	12.6	1.0	0.0			-1.8
rs4	13.6		0.0			-1.7
f	6.6	2.0	3.9	4.2	4.0	
rf	8.6		3.6	3.6	3.4	
lf	10.4	1.7		0.2	0.1	
rlf	12.1			0.0	-0.2	
sf	10.7	1.4	-0.1		0.0	
rsf	12.1		0.0		-0.3	
lsf	10.6	1.5			0.1	
rlsf	12.1				-0.2	
4f	10.5	1.4	0.0	0.2		
r4f	11.9		0.0	0.0		
s4f	10.7	1.2	0.0			
rs4f	11.9		0.0			

TABLE 9
Reexecute Speedup Benefit from Adding Components

Re-execute Speedup Benefit of Adding Components to SAg Hybrids						
hybrid	speedup	+r	+l	+s	+4	+f
r	8.0		6.6	8.5	8.0	5.3
l	11.6	2.9		2.4	2.3	3.8
rl	14.5			1.1	1.4	2.1
s	14.2	2.2	-0.2		1.5	2.2
rs	16.4		-0.8		0.7	1.2
ls	14.0	1.6			1.7	3.7
rls	15.6				1.6	2.2
4	13.9	2.1	0.0	1.8		1.7
r4	16.0		0.0	1.2		1.0
s4	15.7	1.5	0.0			0.5
rs4	17.2		0.0			0.7
f	11.9	1.4	3.5	4.6	3.7	
rf	13.3		3.3	4.4	3.7	
lf	15.4	1.2		1.1	0.2	
rlf	16.6			1.2	0.4	
sf	16.4	1.2	0.0		-0.3	
rsf	17.7		0.1		0.2	
lsf	16.5	1.3			-0.3	
rlsf	17.8				0.1	
4f	15.6	1.5	0.0	0.6		
r4f	17.0		0.0	0.9		
s4f	16.2	1.8	0.0			
rs4f	17.9		0.0			

4.3 Performance Analysis

In an effort to determine why the register value predictor is such a valuable addition to all hybrids while, for example, the LV component generally is not, we investigated how frequently each component in a hybrid can predict a load value that none of the other components can, how often the predictions from different components overlap, and how often they interfere with one another. As pointed out in Section 4.1, not every prediction is equally important (e.g., predicting a load that hits in the L1 data-cache is not as important as predicting a load that has to go all the way to main memory). Consequently, we study the speedup contributions of the hybrids' components rather than the actual set of load instructions that each component can predict to account for the dynamic importance of predicting a specific load at a specific time.

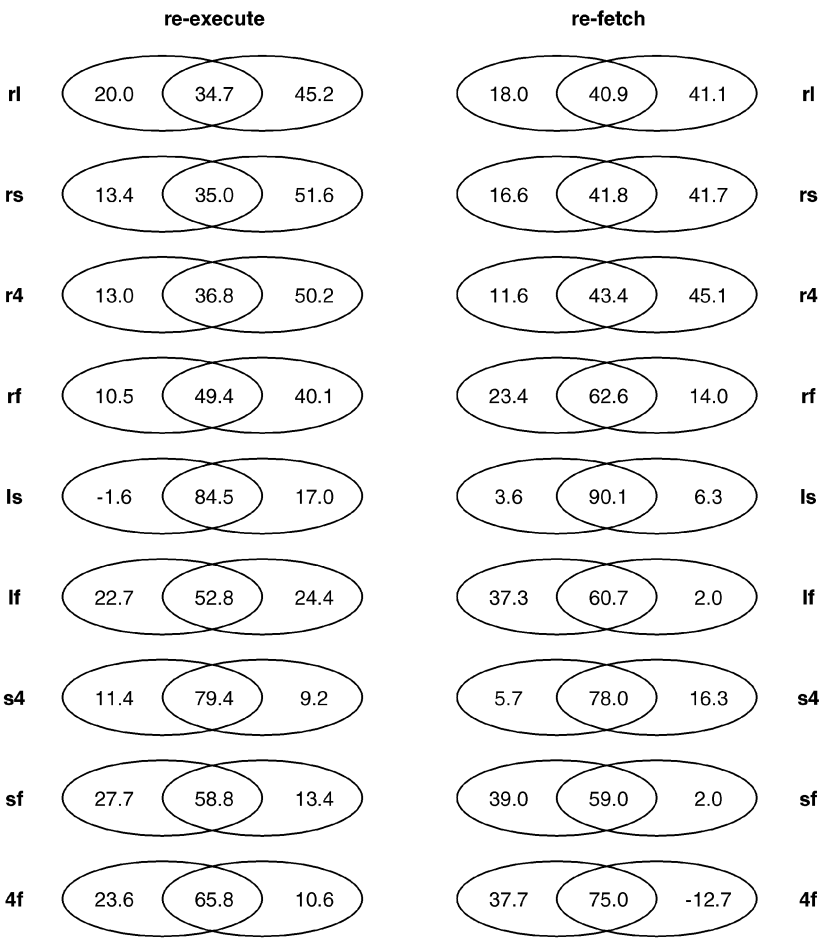


Fig. 5. Reexecute and refetch speedup contributions of two-component hybrids in percent.

4.3.1 Two-Component Hybrids

A hybrid component's *unique* speedup contribution is the part of the overall performance that is lost when the component is removed. In other words, the component must be present to deliver its unique performance contribution. Conversely, in a two-component hybrid, the *shared* contribution is common to both components, meaning that either one is able to provide this contribution, but the contribution does not increase if both components are used together. Hence, only one of the two components is necessary to deliver the shared performance contribution.

We compute the unique and shared speedup contributions in two-component hybrids as follows: Assuming that predictor A yields a speedup of a, predictor B yields a speedup of b, and the hybrid predictor $A+B$ yields a speedup of c, then A contributes $c-b$ unique speedup, B contributes $c-a$ unique speedup, and the shared contribution is $a+b-c$ in the $A+B$ hybrid.

We use Venn diagrams to visualize the different contributions. For example, the top left Venn diagram in Fig. 5 shows that with reexecute recovery, 20 percent of the $\mathrm{Reg}+\mathrm{LV}$ hybrid's speedup stems uniquely from the Reg component, 45.2 percent from the LV component, and the shared contribution is 34.7 percent. (The true sum of the three contributions is, of course, 100 percent, but sometimes the rounding to one digit after the decimal point makes it appear otherwise.) Fig. 5 shows the results for all two-component hybrids (the $\mathrm{L4V}$ predictor is treated as a single component).

The $\mathrm{Reg}+\mathrm{LV}$ hybrid exhibits the smallest shared contribution of any two-component hybrid with both misprediction recovery mechanisms. Clearly, the Reg component complements the LV component well and vice versa, implying that each of them is able to predict important loads that the other cannot. In fact, Reg complements any predictor well. The three predictors with the smallest overlap all include a Reg component and are all among the best performing refetch hybrids. The $\mathrm{Reg}+\mathrm{St2d}$ predictor, the most effective two-component hybrid, has the second smallest overlap both with refetch and reexecute. However, this inverse correlation between overlap and speedup does not hold for reexecute. Furthermore, for both recovery mechanisms, there are hybrids that perform well in spite of a large shared contribution, for instance, the $\mathrm{St2d}+\mathrm{L4V}$ and $\mathrm{St2d}+\mathrm{FCM}$ predictors. This is an artifact of the relative numbers given in Fig. 5. For well-performing components, even a small relative contribution can be large in absolute terms.

The hybrids $\mathrm{LV}+\mathrm{St2d}$, $\mathrm{St2d}+\mathrm{L4V}$, and $\mathrm{L4V}+\mathrm{FCM}$ exhibit large shared contributions and their components therefore complement each other only poorly. The LV component in the $\mathrm{LV}+\mathrm{St2d}$ hybrid with reexecute

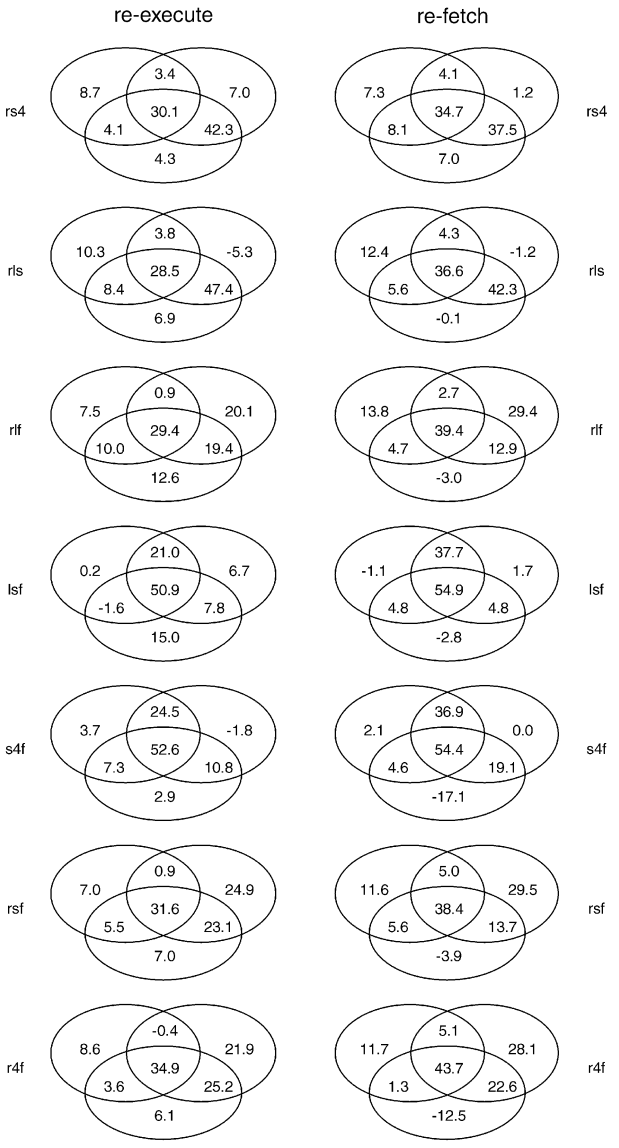


Fig. 6. Reexecute and refetch speedup contributions of three-component hybrids in percent.

and the FCM component in the L4V + FCM hybrid with refetch show a negative individual contribution. This is because, with reexecute, the St2d component performs better than the LV + St2d hybrid and with refetch the L4V component outperforms the L4V + FCM hybrid. Hybridizing actually lowers the performance in these cases and, thus, results in a negative unique contribution. The reason is the aforementioned negative interference between the involved components. Since single-component predictors do not require a selector, whereas hybrids do, the culprit for the lower performance must be the imperfect selector. After all, the St2d component in the LV + St2d hybrid is identical to the single-component St2d predictor that outperforms the hybrid.

With only one exception (St2d + L4V), refetch recovery results in larger shared contributions than reexecute for the same predictors. This probably means that the easily predictable loads (i.e., loads that have very high confidences associated with them) tend to be the loads that both components can predict. Such loads most likely fetch runtime constants that are always last-value predictable [7].

Overall, the Reg predictor complements the other four predictors exceptionally well, indicating that it can predict a rather distinct set of load instructions. The next best "partner" is the FCM predictor. The St2d predictor does not complement the LV or the L4V predictor well because all three predictors mostly predict last-value-predictable loads.

The St2d + L4V hybrid is similar to the *last distinct four value + stride* predictor proposed by Wang and Franklin [28] and St2d + FCM is the hybrid proposed by Rychlik et al. [23], except it is not set-associative and uses a different confidence estimator. Fig. 5 illustrates that the two predictors exhibit significant shared speedup. Together with the performance results from Fig. 3 and Fig. 4, we find that, for the studied predictor sizes, the smaller and simpler Reg + St2d hybrid outperforms both the St2d + L4V and the St2d + FCM predictors, illustrating the importance of component analyses when designing hybrid load-value predictors.

4.3.2 Three-Component Hybrids

Fig. 6 shows the shared and unique speedup contributions (in percent of total predictor performance) of the three-component hybrids. A set of seven equations has to be solved to compute the seven values shown in each Venn diagram. The numbers in the center of each diagram denote the contribution that is shared between all three predictor components, the other three overlapping regions represent the shared contribution of pairs of components, and the nonoverlapping sections list the unique speedup contributions. For example, in the top left Venn diagram (rs4), the upper left oval lists the contribution of the Reg component, the upper right oval the contribution of the St2d component, and the oval at the bottom the contribution of the L4V component.

Fig. 6 illustrates that the three components in the LV + St2d + FCM and the St2d + L4V + FCM hybrids exhibit large shared speedups. In both hybrids, over half of the performance is shared among all three components. The Reg + St2d + L4V and the Reg + LV + St2d predictors have somewhat large shared contributions and the remaining hybrids exhibit relatively little sharing in at least one of their components. Note that Reg's unique contribution is at least seven percent in every case.

Again, the amount of sharing correlates reasonably with the refetch performance, but there is no significant correlation with the reexecute performance. Nevertheless, the Venn diagrams expose components that do not contribute any performance and components that hurt the performance. Such components should be removed, which will *not* lower the predictor's performance but will make it smaller, faster, and reduce the power consumption.

Because the four-component hybrids do not outperform the best three-component hybrids with refetch and do not significantly outperform the best three-component hybrids with reexecute, we refrain from studying the speedup contributions of the two four-component hybrids in detail.

5 Related Work

Two independent research efforts [10], [17] first recognized that load instructions exhibit *value locality* and concluded that there is potential for prediction. Lipasti et al. [17] propose the last value predictor. Gabbay [10] proposes four predictor schemes: a last value predictor, a stride predictor, a register file predictor, and a sign-exponent-fraction (SEF) predictor. The SEF predictor is only useful for predicting IEEE floating-point loads. Tullsen and Seng [27] present the register value predictor as we use it in this study. We found their predictor to be a great complement for any other component in hybrid predictors.

In their next paper, Lipasti and Shen [16] suggest making predictions based on the last n values instead of just the last value. Wang and Franklin [28] propose a last distinct four value predictor as well as the first hybrid predictor, a combination of their last distinct four value predictor and a stride predictor. In previous work [3], [6], we show the last four value predictor to be simpler but about as effective as the last distinct four value predictor.

Sazeides and Smith [25] describe the finite context method predictor. They found that this predictor performs very well with large table sizes. Since we use a relatively small FCM component in our hybrids, it is quite possible that a larger such component would further improve the performance. Goeman et al. [13] were able to substantially reduce the FCM's storage requirement by retaining strides instead of full values in the first and second level. Rychlik et al. [23] use a hybrid between a finite context method predictor and a stride 2-delta predictor in their study. Like Reinman and Calder [21], they use the confidence estimators in the components of their hybrids as selector, thus eliminating the need for extra storage to guide the selection process. We use the same approach.

Later, Rychlik et al. [22] augment their predictor with a popular last value predictor and study updating only one component at a time to increase the predictor's capacity. We tackled the capacity issue in previous work by investigating approaches to shrink the predictor size without loss of performance [4]. By compressing values and sharing information between predictor components, we were able to reduce the size of the $\mathrm{Reg} + \mathrm{St2d} + \mathrm{L4V}$ to only about twice the size of an LV predictor with the same number of lines without negatively affecting the hybrid's performance. Pinuel et al. [20] present a hybrid between a last value, a stride, and a finite context method predictor in which information is also shared between components to keep the predictor size small.

Reinman and Calder [21] examine a different kind of hybrid that combines dependence prediction, value prediction, address prediction, and memory renaming. They conclude that, due to the small hardware requirement, dependence prediction should be added to new processors first even though value prediction provides a larger performance improvement. Address prediction and memory renaming are shown to be inferior to dependence and value prediction.

Lee et al. [15], [18] integrate their value predictor into the trace cache and decouple the predictions from the instruction fetch stage to reduce the number of accesses to each value prediction table. All predictors used in this study are banked to limit the number of accesses to one per cycle and table. Similarly to their predictors, we buffer predictor updates in queues [3].

6 Summary and Conclusions

This paper studies the performance of all hybrid load-value predictors that can be built out of a register value, a last value, a stride 2-delta, a last four value, and a finite context method predictor. Our analysis shows that hybrids are able to deliver substantially more speedup than the best single-component predictors and that different components contribute independently to the overall performance. We conclude that multicomponent predictors, in which each component is tailored to a different kind of load-value locality, are necessary to effectively exploit the existing value locality.

An investigation of the speedup contributions of individual components revealed that the register value predictor, which by itself performs only poorly, represents the most valuable addition to any other studied component. Conversely, combining well-performing predictors often does not result in an effective hybrid. In fact, we found that some predictor combinations perform worse than the individual predictors. This happens when the extra component causes more selector-related losses than the added predictability can compensate for.

Our analysis shows that the prediction rate, accuracy, and related metrics can sometimes be inversely correlated with the true speedup, indicating that such metrics are poor performance indicators. In particular, we found the latency and the time to first use of the predicted loads to be of great importance, emphasizing the need for cycle-accurate simulations.

Our hybridization analysis identified the *register value + stride 2-delta* predictor as one of the best two-component hybrids. In spite of its substantially smaller and simpler design, it matches or exceeds the speedup of two-component hybrids from the literature. Of all the studied predictors, the *register value + stride 2-delta + last four value* hybrid performs best with refetch as well as when averaging the refetch and reexecute speedups.

Among predictors with 2,048 lines, the best hybrids yield harmonic-mean speedups over the eight SPECint95 programs of close to 18 percent and outperform the best single-component predictors by over 25 percent. These performance improvements are obtained with transparent load-value predictors that require no changes to the instruction-set architecture and can therefore be added to existing CPU families as well as future processors. Furthermore, the speedups stem from programs that were not compiled with load-value prediction in mind. In future work, we intend to study compiler optimizations to further improve the performance of hybrid- and single-component load-value predictors.

Acknowledgments

This work was supported in part by the Hewlett Packard University Grants Program (including Gift No. 31041.1) and the Colorado Advanced Software Institute. The authors would like to especially thank Tom Christian for his support of this project and Dirk Grunwald and Abhijit Paithankar for providing and helping with the cycle-accurate simulator. The simulations were performed on

Alpha machines, which were sponsored in part by Digital Equipment Corporation (now Compaq). A large part of this work was performed at the University of Colorado at Boulder.

REFERENCES

[1] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated Load-Address Predictors," *Proc. 26th Int'l Symp. Computer Architecture,* May 1999.

[2] M. Burtscher, A. Diwan, and M. Hauswirth, "Static Load Classification for Improving the Value Predictability of Data-Cache Misses," to appear in *ACM SIGPLAN Conf. Programming Language Design and Implementation,* June 2002.

[3] M. Burtscher, "Improving Context-Based Load Value Prediction," PhD dissertation, Dept. Computer Science, Univ. of Colorado at Boulder, Apr. 2000.

[4] M. Burtscher and B.G. Zorn, "Hybridizing and Coalescing Load-Value Predictors," *Proc. Int'l Conf. Computer Design,* pp. 81-92, Sept. 2000.

[5] M. Burtscher and B.G. Zorn, "Load Value Prediction Using Prediction Outcome Histories," Technical Report CU-CS-873-98, Univ. of Colorado at Boulder, Oct. 1998.

[6] M. Burtscher and B.G. Zorn, "Exploring Last n Value Prediction," *Proc. 1999 Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 66-76, Oct. 1999.

[7] B. Calder, P. Feller, and A. Eustace, "Value Profiling," *Proc. 30th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* Dec. 1997.

[8] B. Calder, G. Reinmann, and D.M. Tullsen, "Selective Value Prediction," *Proc. 26th Int'l Symp. Computer Architecture,* May 1999.

[9] *Alpha Architecture Handbook.* Digital Equipment Corp., 1992.

[10] F. Gabbay, "Speculative Execution Based on Value Prediction," Technical Report #1080, EE Dept., Technion-Israel Inst. of Technology, Nov. 1996.

[11] F. Gabbay and A. Mendelson, "The Effect of Instruction Fetch Bandwidth on Value Prediction," *Proc. 25th Int'l Symp. Computer Architecture,* June 1998.

[12] J. González and A. González, "Speculative Execution via Address Prediction and Data Prefetching," *Proc. 11th Int'l Conf. Supercomputing,* pp. 196-203, 1997.

[13] B. Goeman, H. Vandierendonck, and K. Bosschere, "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency," *Proc. Seventh Int'l Symp. High Performance Computer Architecture,* Jan. 2001.

[14] R.E. Kessler, E.J. McLellan, and D.A. Webb, "The Alpha 21264 Microprocessor Architecture," *Proc. 1998 Int'l Conf. Computer Design,* Oct. 1998.

[15] S.-J. Lee and P.-C. Yew, "On Table Bandwidth and Its Update Delay for Value Prediction on Wide-Issue ILP Processors," *IEEE Trans. Computers,* vol. 50, no. 8, pp. 847-852, Aug. 2001.

[16] M.H. Lipasti and J.P. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proc. 29th Int'l Symp. Microarchitecture,* Dec. 1996.

[17] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen, "Value Locality and Load Value Prediction," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 138-147, Oct. 1996

[18] S.-J. Lee, Y. Wang, and P.-C. Yew, "Decoupled Value Prediction on Trace Processors," *Proc. Sixth Int'l Symp. High Performance Computer Architecture,* Jan. 2000.

[19] A. Paithankar, "AINT: A Tool for Simulation of Shared-Memory Multiprocessors," master's thesis, Univ. of Colorado at Boulder, 1996.

[20] L. Pinuel, R.A. Moreno, and F. Tirado, "Implementation of Hybrid Context Based Value Predictors Using Value Sequence Classification," *Proc. Euro-Par,* Aug. 1999.

[21] G. Reinman and B. Calder, "Predictive Techniques for Aggressive Load Speculation," *Proc. 31st Ann. ACM/IEEE Int'l Symp. Microarchitecture,* Dec. 1998.

[22] B. Rychlik, J.W. Faistl, B.P. Krug, A.Y. Kurland, J.J. Sung, M.N. Velev, and J.P. Shen, "Efficient and Accurate Value Prediction Using Dynamic Classification," Technical Report CMμART-1998-01, Carnegie Mellon Univ., 1998.

[23] B. Rychlik, J. Faistl, B. Krug, and J.P. Shen, "Efficacy and Performance Impact of Value Prediction," *Proc. 1998 Int'l Conf. Parallel Architectures and Compiler Technology,* Oct. 1998.

[24] Y. Sazeides and J.E. Smith, "The Predictability of Data Values," *Proc. 30th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* Dec. 1997.

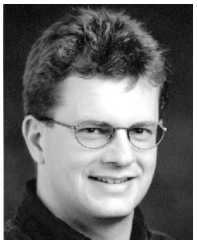
[25] Y. Sazeides and J.E. Smith, "Implementations of Context Based Value Predictors," Technical Report ECE-97-8, Univ. of Wisconsin-Madison, Dec. 1997.

[26] *SPEC CPU'95,* Aug. 1995.

[27] D. Tullsen and J. Seng, "Storageless Value Prediction Using Prior Register Values," *Proc. 26th Int'l Symp. Computer Architecture,* May 1999.

[28] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction Using Hybrid Predictors," *Proc. 30th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* Dec. 1997.

[29] T.Y. Yeh and Y.N. Patt, "A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History," *Proc. 20th Ann. Int'l Symp. Computer Architecture,* pp. 257-266, May 1993.



Martin Burtscher received the Dipl. Informatik-Ing. degree from the Swiss Federal Institute of Technology (ETH) Zurich in 1996 and the PhD degree in computer science from the University of Colorado at Boulder in 2000. He is an assistant professor in the School of Electrical and Computer Engineering at Cornell University. His research focuses on high-performance microprocessor architecture, instruction-level parallelism, and compiler optimizations. His current work includes hardware and software-based value prediction, data compression, and memory latency reduction. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Benjamin G. Zorn received the BS degree from Rensselaer Polytechnic Institute (RPI) in 1982 and the MS and PhD degrees from the University of California at Berkeley in 1984 and 1989, respectively. He is a senior researcher at Microsoft Research and currently leads the Performance Monitoring and Analysis Group. His research interests include performance optimization and measurement, the design and implementation of programming languages, and programming language runtime systems. Prior to joining Microsoft Research, Dr. Zorn was an associate professor of Computer Science at the University of Colorado at Boulder, where he worked from 1990 to 1998. He is a member of the ACM and the IEEE Computer Society.