Journal of Instruction-Level Parallelism 5(2003) 1-23 Submitted 9/03; published 11/03

Width-Partitioned Load Value Predictors

Gabriel H. Loh LOH@CC.GATECH.EDU
College of Computing
Georgia Institute of Technology
Atlanta, GA, 30332-0280

Abstract

Value prediction has been proposed for breaking data-dependencies and increasing instruction level parallelism. One of the drawbacks of many of the proposed techniques is that the value predictors require very large hardware structures which use up many transistors and can consume a large amount of energy. In this study, we use data-widths to partition the value prediction structures into multiple smaller structures, and then use data-width prediction to select from these tables. A width-partitioned *value predictor requires less space because small bit-width values get allocated to smaller storage locations instead of using a full 64 bits. The total energy consumption of the predictor is also reduced because only a single smaller predictor table needs to be accessed. Width-partitioning provides a 25.7-46.5% reduction in energy consumption for last value predictors and finite context matching (FCM) predictors. For FCM predictors, width partitioning allows the total size of the second-level tables to be reduced by 75% while still maintaining the same prediction rates of a conventional FCM predictor. Our performance studies indicate that width partitioning does not degrade the geometric mean IPC on the SPEC2000 integer benchmarks, thus showing that width partitioning is an effective technique to reduce the size and energy requirements of value predictors.*

1. Introduction

Modern processors are using speculation in increasingly aggressive ways. Branch prediction removes control dependencies to increase the window of instructions that a processor can search for parallelism. Memory dependency prediction removes false dependencies between store and load instructions. Data dependencies through registers have been thought to limit the inherent instruction level parallelism of programs. The idea of data value prediction has been proposed and researched for breaking these so-called "true dependencies" and uncovering more parallelism [14].

Although value prediction may enable speedups by removing critical data dependencies, many of the proposed prediction algorithms require large hardware structures that consume great amounts of space and power [24]. Even though transistor budgets in modern processors continue to increase, overly large value predictors may still not be desirable because the extra transistors could potentially be better used for other resources, such as larger caches. The power consumption of processors is also one of the most critical problems facing processor designers [10, 27]. We are not advocating value prediction for low-power processor designs, but simply observing that a power-hungry solution may not be viable even in a high-performance processor.

In Lipasti et al.'s seminal value prediction study, they make the observation that allocating a full 64 bits for each value predictor entry is suboptimal because many values do not require the full 64

bits [14]. They suggest that the space for a 64-bit entry could be shared among multiple smaller-width entries. Instead of attempting to dynamically pack multiple small-width values into a single larger storage location, we propose to use multiple smaller tables each with different widths and use a *data-width predictor* to choose among the tables [15]. Besides providing a more efficient use of predictor table storage, we also achieve power savings by only accessing one of the smaller tables. In this paper, we only study load value prediction, although the techniques proposed can be easily extended to general data-value predictors.

The rest of this paper is organized as follows. Section 2 describes the different data-width properties that we are interested in and presents the measurements of these properties for our benchmarks. Section 3 explains how to leverage these data-width properties to reduce both the space and power requirements of a several types of value predictors. Section 4 presents our results. Section 5 reviews some of the past research related to our study, and Section 6 concludes the paper.

2. Load Data-Width Properties

Past studies have exploited the fact that the widths of data values in a program are not evenly distributed. For a 64-bit architecture, relatively few values actually need all 64 bits since many of the upper bits are all zero [1]. Furthermore, data-widths are very stable over time; that is if an instruction produces a 16-bit result, it is likely that the next instance of the same instruction will produce another 16-bit result [15]. This past research has shown that these properties tend to hold for integer computation instructions (e.g., arithmetic operations, boolean logic, shifts). In this section, we demonstrate that the values produced by load instructions have similar properties.

2.1 Data-Width Distributions

We first measure the distribution of data-widths in the load-value stream. The data-width of a value is equal to the position of the most significant non-zero bit. Figure 1 shows the cumulative load data-width distribution averaged across the twelve SPEC2000 integer benchmarks (see Section 4 for complete details on the data collection methodology and the benchmarks used). We also considered treating the values as sign-extended instead of zero-extended by measuring the bit-widths of the absolute value of the load values. These results are marked as “signed” in Figure 1.

The distribution of load-value data-widths is not uniform. The curve follows a similar shape to the register data-width distribution from Brooks and Martonosi’s study for the SPEC95 integer benchmarks [1]. The largest fraction of loads is dominated by addresses, which have 33 bit wide values. This is an artifact of the SPEC applications and the compiler. A program with a larger memory footprint may have this spike at a larger bit-width. The next largest group of load values is those that fit in one byte or less. There is also a significant fraction of loads with a value of zero. This has been observed in the past and was exploited for optimizing data-caches as well as for *frequent value prediction* [20, 26]. Treating values as signed or unsigned makes little difference to the overall distribution.

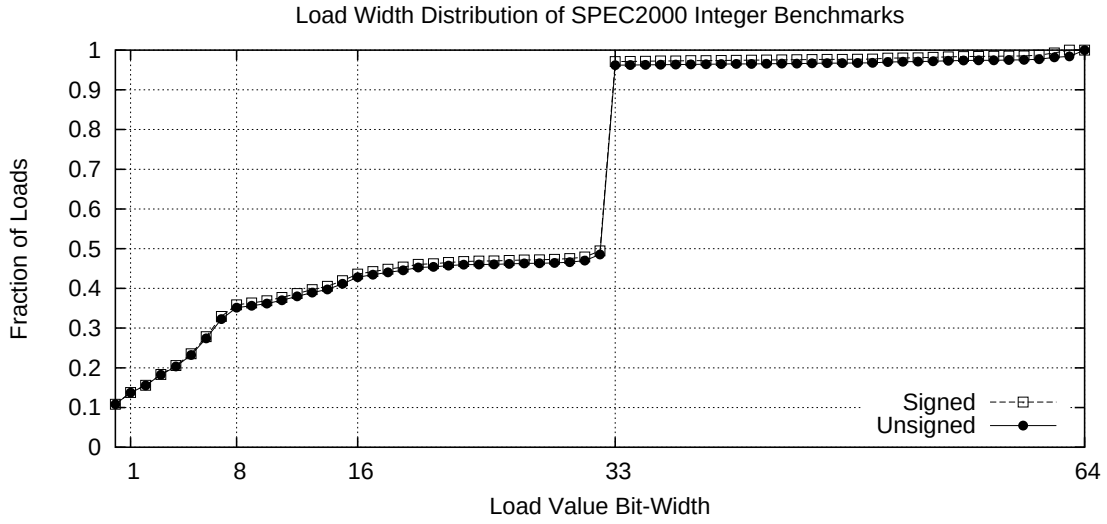

Figure 1: The cumulative distribution of load-value data-widths for the SPEC2000 integer benchmarks. The "signed" distribution corresponds to when the data-width of a value is measured using sign-extension instead of zero-extension.

2.2 Data-Width Locality

For the purposes of optimizing a load-value predictor and many other width-based optimizations, using the exact bit-width is not necessary or perhaps even desirable. Instead, we use ranges of bit-widths where any two values that fall into the same range are considered to have the same bit-width. Based on the data-width distribution of Figure 1 and natural width boundaries, we use a total of six different width classifications: $\mathcal{W}_0, \mathcal{W}_1, \mathcal{W}_8, \mathcal{W}_{16}, \mathcal{W}_{33}$ and $\mathcal{W}_{64}$. The classes $\mathcal{W}_0$ and $\mathcal{W}_1$ contain values with widths of zero and one, respectively. These happen to also be the values of 0 and 1. For the other classes, $\mathcal{W}_i$ includes all values with a bit-width of i down to the next smallest class. For example, the class $\mathcal{W}_{16}$ contains all values with bit-widths from sixteen down to nine.

Our previous study on data-width locality demonstrated that integer computations exhibit strong locality across the $\mathcal{W}_{16}, \mathcal{W}_{33}$ and $\mathcal{W}_{64}$ classes [15], whereas we want to measure the data-width locality of load instructions across six width classes. We used a PC-indexed table that records the last seen data-width ($\mathcal{W}_0$ to $\mathcal{W}_{64}$) of a load value, and measured the number of loads with data-widths equal to the previous instance. Figure 2 shows these last-width prediction results. For each benchmark, Figure 2 shows four sets of data corresponding to table sizes of 256, 512, 1K and 2K entries. Overall, simple last width prediction provides over 92% prediction accuracy. Increasing the size of the prediction table beyond 2K entries does not provide much greater accuracy. We also experimented with a version that uses an extra hysteresis bit, but that also made little difference.

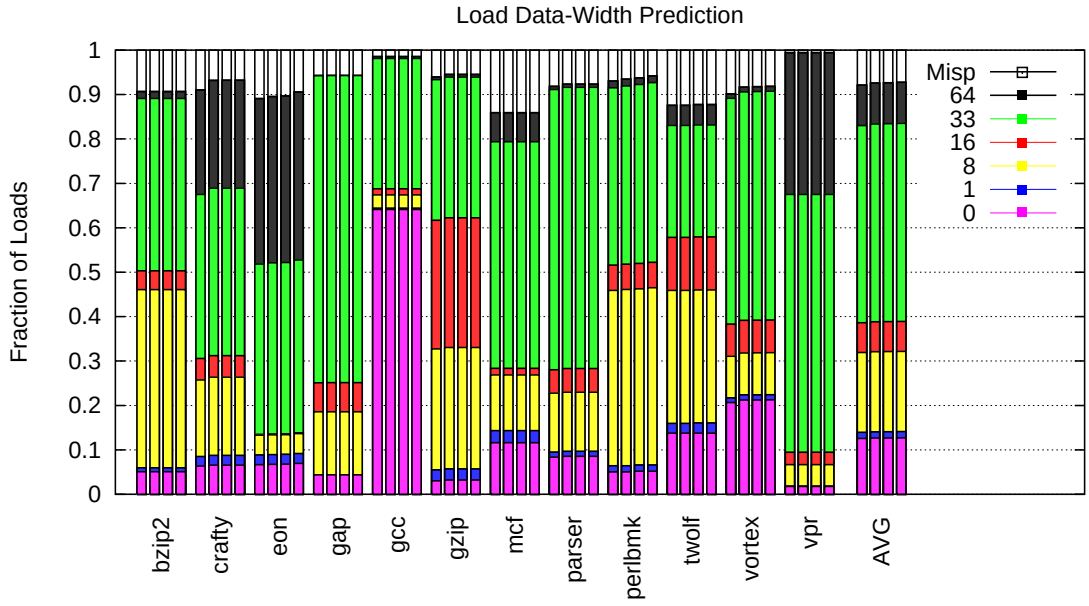

Figure 2: The last width prediction accuracy for the SPEC2000 integer benchmarks. For each benchmark, the four bars correspond to last width predictors with 256, 512, 1024 and 2048 entries (from left to right).

3. Width-Partitioned Predictors

Having demonstrated that load-value data-widths have an uneven distribution and are easily predictable, we now describe how to use these properties to optimize value predictors through a technique called *width partitioning*. Afterward, we will demonstrate these techniques by detailing the designs for width-partitioned Last Value Predictors (LVP) [14], stride-based predictors [9], and Finite Context Matching (FCM) predictors [21]. In general, width partitioning should be applicable to other value predictors beyond those described here.

Value predictors employ a table of values that can be optimized to take advantage of the load width distributions. This table is typically arranged as a direct-mapped cache structure. Depending on the organization, the entries may use tags like traditional caches, partial-tags like BTBs with partial resolution [8], or no tags at all like branch predictor pattern history tables [25]. While tags play an important role in the performance and tuning of value predictors, our optimizations are focused on the value storage of the tables. Therefore we will not be discussing the tags in any great detail.

For any value predictor with a table of values, width partitioning divides the table into multiple sub-tables that each stores values only for a given *width-class*. For all possible data-width (from 0 bits to 64 bits), we define n width classes where each class contains a non-overlapping interval of data-widths, such that the union of all n classes covers the entire range from 0 to 64 bits. In Section 2 we used $\mathcal{W} = \{\mathcal{W}_0, \mathcal{W}_1, \mathcal{W}_8, \mathcal{W}_{16}, \mathcal{W}_{33}, \mathcal{W}_{64}\}$, but the technique of width partitioning can be applied to any set of width-classes. The idea is then to use data-width prediction to choose

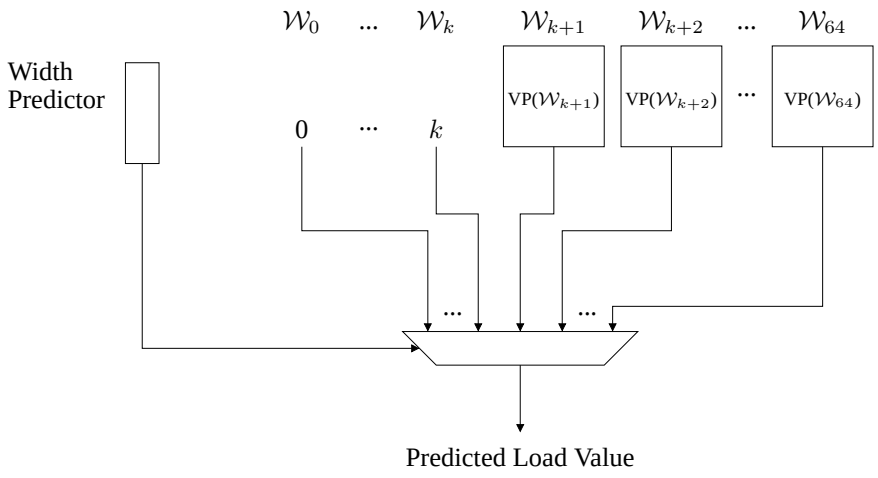

Figure 3: A generic width-partitioned value predictor.

among the n subtables of the predictor. By sizing the individual subtables to suit the data-width distribution of the targeted applications, we improve the capacity of the predictor due to a larger total number of predictor table entries. By only accessing a single subtable, we provide energy savings because only a single smaller table is accessed.

Figure 3 shows the organization of a generic width-partitioned value predictor with a generic width-classification $\mathcal{W} = \{\mathcal{W}_0, ..., \mathcal{W}_k, \mathcal{W}_{k+1}, \mathcal{W}_{k+2}, ..., \mathcal{W}_{64}\}$. The classes $\mathcal{W}_0$ through $\mathcal{W}_k$ each only have a single value per set. For example, $\mathcal{W}_0$ and $\mathcal{W}_1$ each only have the single values zero and one respectively. For such classes, no explicit value predictor is necessary; the value can be directly hard-wired. For the remaining classes $\mathcal{W}_{k+1}$ through $\mathcal{W}_{64}$, each uses its own value predictor VP($\mathcal{W}_{k+1}$) to VP($\mathcal{W}_{64}$) respectively. Each of these value predictors has a different data-width for its stored values, and each predictor can have a different number of entries. Finally, a width predictor chooses among the possible predictions. Note that in this paper we use a last-width predictor, but in general this could be any other prediction algorithm. Techniques used in the area of branch prediction to reduce interference and increase accuracy could potentially be applied to data-width prediction as well.

3.1 Width-Partitioned Last Value Predictor

The Last Value Predictor (LVP) is among the simplest value predictors, but it is very effective for illustrating how width partitioning is used to optimize value predictors in general. A LVP uses a single value prediction table (VPT). A load instruction's address (PC) indexes into the table which stores the last value seen for this load (or any other load that aliases to the same entry if no tags or partial tags are employed).

To width-partition the LVP, we replace the monolithic value prediction table (VPT) with multiple value prediction tables, each with different maximum widths, and then choose between them using a data-width predictor. Figure 4 shows the organization of our *width-partitioned last value predictor* (WP-LVP). The WP-LVP uses four different last value predictor tables VPT_8, VPT_{16},

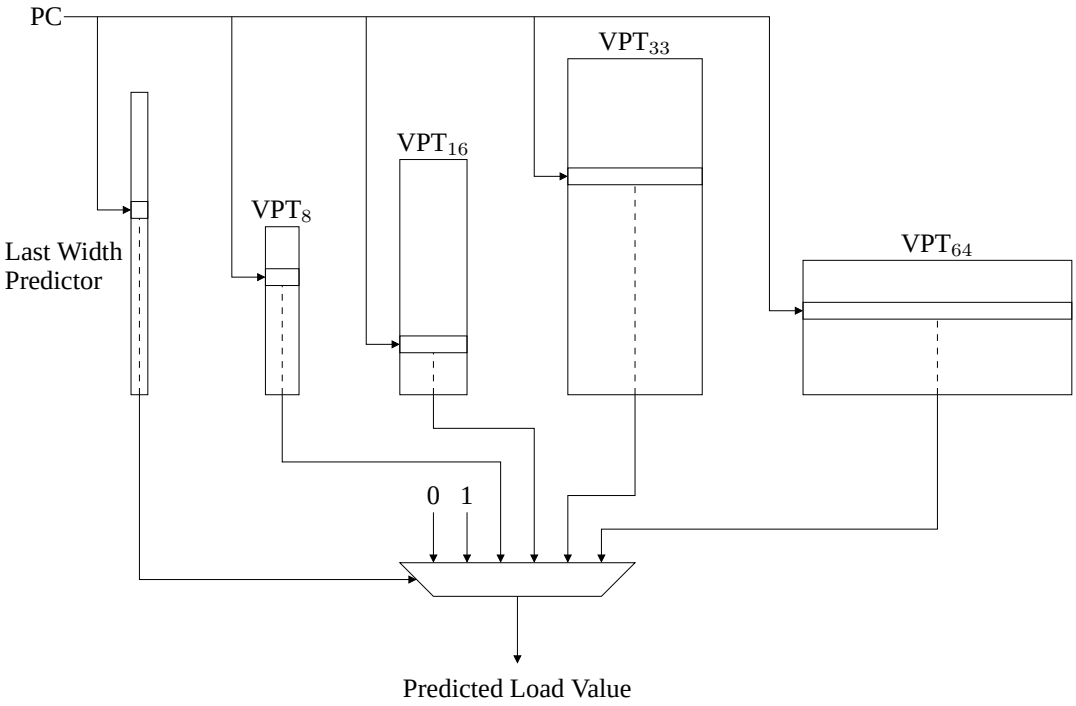

Figure 4: The organization of the width-partitioned last value predictor (WP-LVP). Each table is PC-indexed, and the width predictor selects one of six possible value predictions. Each table may have a different number of entries.

VPT_{33} and VPT_{64}. The notation VPT_i means that each table entry only stores i bits of data. Each of the individual VPTs may have a different number of entries. The exact sizes that we use in this study are based on the distribution of data-widths, which we will discuss with our results in Section 4.

For the predictor lookup, the PC-indexed last width predictor tells the WP-LVP which individual value predictor to use. The predictor then performs the lookup on the corresponding VPT. Note that the WP-LVP does not need any tables for the $\mathcal{W}_0$ or $\mathcal{W}_1$ width classes because each of these classes contain only a single value. A final multiplexer uses the data-width prediction to choose from 0, 1 or the output of one of the VPTs.

For the predictor update, the WP-LVP finds the actual data-width of the load value and stores this width in the last width predictor. Then the WP-LVP stores the value in only the single VPT that corresponds to the actual data-width. For example, if the load's data-width is W_{16}, then only VPT_{16} gets updated. By storing values in structures with a matching size, the WP-LVP greatly reduces the amount of storage wasted on the many upper bits that are usually zero.

Serializing the last width predictor lookup and the VPT lookup increases the overall predictor lookup latency, but this greatly reduces the energy consumption of the value predictor. In a traditional full-width LVP, each lookup (and update) requires accessing a large table which drives 64 bits worth of output data lines. On each lookup for the WP-LVP, we only need to access a single,

smaller VPT. The smaller VPT consumes less energy because it has fewer entries and all VPTs with the exception of VPT_{64} drive fewer output lines. If the data-width prediction is $\mathcal{W}_0$ or $\mathcal{W}_1$, then we save even more power by not accessing any VPTs. The serializing of the last width predictor and the VPT lookups increases the overall latency of the WP-LVP, but the value predictor lookup is not on a critical path because it can be initiated early in the pipeline and the prediction is not needed for many cycles.

We use six data-width classes, which requires three bits to encode each entry of the last width predictor. With three bits, we could theoretically encode two more values or width classes. We experimented with variations that encoded different combinations of the values -1, 2 and 3, but this had almost no impact on the overall results.

As presented, our WP-LVP is a tagless structure. We found that while adding tags reduces the number of mispredictions, the additional hardware cost for storing the tags is better used for increasing the number of entries in the individual predictor tables. *Partial tags* could also be used [7], but examining the tradeoff between taglengths, predictor sizes, width prediction, accuracy and power is beyond the scope of this study.

3.2 Width-Partitioned Strided Value Predictor

The last value predictor can only accurately predict "dynamically constant" load values, that is, values that do not change over some interval of time. There are many values that change frequently, but do so in a regular fashion. Strided value predictors (SVPs) store both the last-seen value as well as a predicted stride or delta to predict values which increase or decrease in a linear fashion [9], such as loop induction variables.

Extending a strided value predictor to incorporate data-width predictions is very similar to the approach used for the simpler last value predictor. Instead of a single SVP that stores 64-bit values, we can again use multiple SVPs that each stores values of specific widths. Figure 5 illustrates the organization of the width-partitioned stride predictor (WP-SVP) using the same width-classes as for the WP-LVP.

A width-partitioned strided value predictor can be further optimized. Depending on the actual distribution of stride-widths, the number of bits used to store the stride may also be sized differently for each value prediction table. One could go so far as to predict the widths of the strides, and store the strides in multiple tables, but the slight decrease in size and power for this minor optimization is probably not worth the additional complexity of implementing this scheme. Due to the strong data-width locality, the width of a value is likely to remain unchanged even after adding its stride. If the width-class does change from adding the stride to the value, then it could be possible to speculatively update the last width predictor such that the next access will perform a lookup on the corresponding SVP subtable.

3.3 Width-Partitioned Finite-Context Matching Value Predictor

The Finite Context Matching (FCM) predictor provides superior value prediction accuracy, but the hardware organization is also more complex. The FCM predictor is a two-level value predictor, analogous to two-level branch predictors such as the PAs or gshare predictors [16, 25]. Instead of a branch history table for tracking past branch outcomes, FCM uses a *value history table* (VHT)

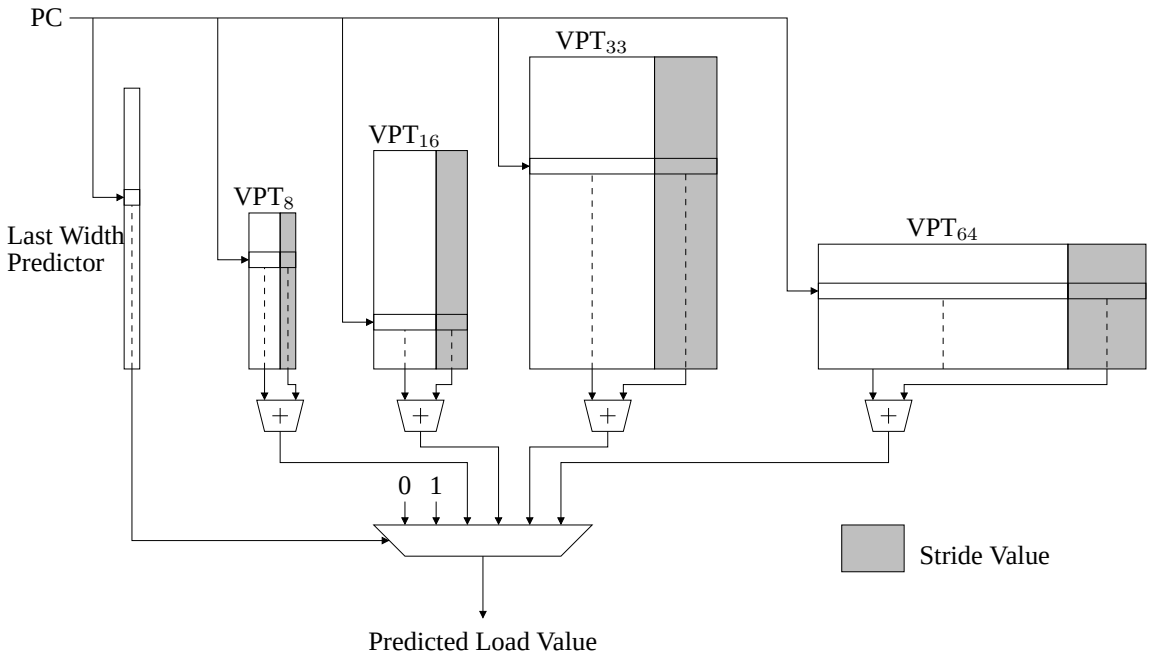

Figure 5: A width-partitioned strided value predictor.

for remembering past load-value patterns. Figure 6 shows the hardware organization of a FCM predictor. For each load, an order-m FCM predictor stores the last m load values in the VHT. The FCM predictor uses a hash of this value history to index into a second-level *value prediction table* (VPT) that stores the actual value prediction.

In a FCM predictor, there are two structures that can potentially benefit from our width partitioning technique: the VHT and the VPT. The problem of partitioning the value history table based on data-widths is complicated by the fact that the last m load values for a given PC may have varying widths. The strong temporal locality of load-value data-widths suggests that in the common case, the widths of all values in the value history will have the same width so long as m is not too large. Our approach is to simply divide the value history stream into multiple substreams based on data-widths. The partitioning of the VPT is analogous to that of the WP-LVP. We use four separate VPTs of different widths, and select from only one based on the width prediction. Figure 7 shows the hardware organization of a FCM predictor with both the VHT and VPT partitioned by width. Note that predictions of zero and one are based solely on the last width predictor and do not incorporate any value history.

We considered two width-partitioned FCM predictors. The first configuration, called the partially width-partitioned FCM (PWP-FCM) predictor uses a conventional VHT combined with a width-partitioned VPT. The PWP-FCM predictor may require different hashing functions to index into the different sized second-level prediction tables. The second configuration (shown in Figure 7) is the fully width-partitioned FCM (FWP-FCM), that uses width partitioning for both the VHT and the VPT.

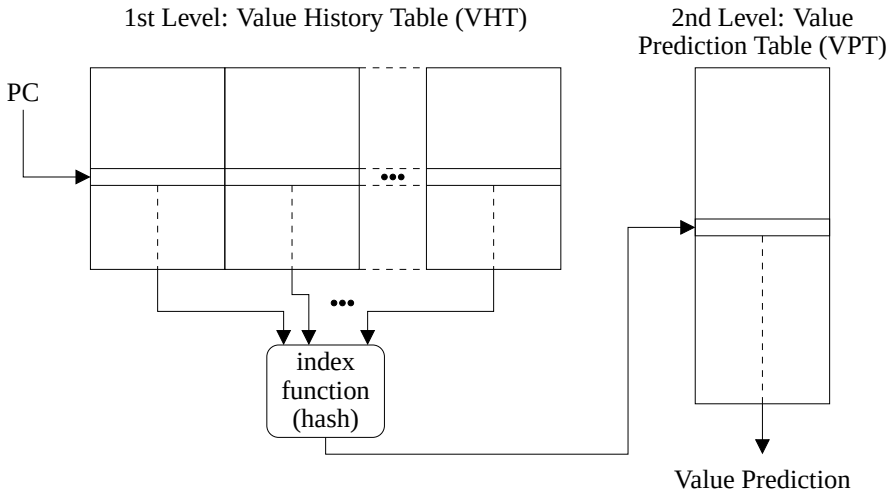

Figure 6: The finite context matching (FCM) predictor uses a hash of the past local value history of a load as an index into the value prediction table.

In the PWP-FCM predictor, we must store the full 64-bit values in the value history table because the hashing functions vary depending on the width prediction. For the FWP-FCM predictor where there are per-width VHTs, each VHT is used to index into only a single VPT, and therefore only one hash function ever gets used. In this case, we can save space by storing the hashed versions of the values directly in the VHTs. This also has the additional benefit of moving some of the hashing latency to the update phase. Note that zeros and ones never get inserted into any of the VHTs.

4. Results

In this section, we present the performance results for the width-partitioned LVP and FCM predictors. We measure prediction accuracy rates and power savings for predictors over a range of hardware budgets, and we measure the performance impact for value predictors with 8K-entry value prediction tables.

4.1 Methodology

In this study, we used the SimpleScalar toolset to collect all of our data except for the power savings results [2]. We used a modified version of the in-order functional simulator sim-safe to collect the load property statistics of Section 2. We wrote our own *sim-vpred* load-value predictor simulator. Sim-vpred is the value-prediction analogue of the in-order branch predictor simulator sim-bpred. We used the twelve integer applications from the SPEC2000 benchmark suite. For each benchmark, we simulated 200 million instructions after skipping the initial setup phase of the program. All benchmarks were compiled with `cc -arch ev6 -non_shared -fast -O4` on an Alpha 21264. Table 1 lists the benchmark input sets and simulation fastforward points.

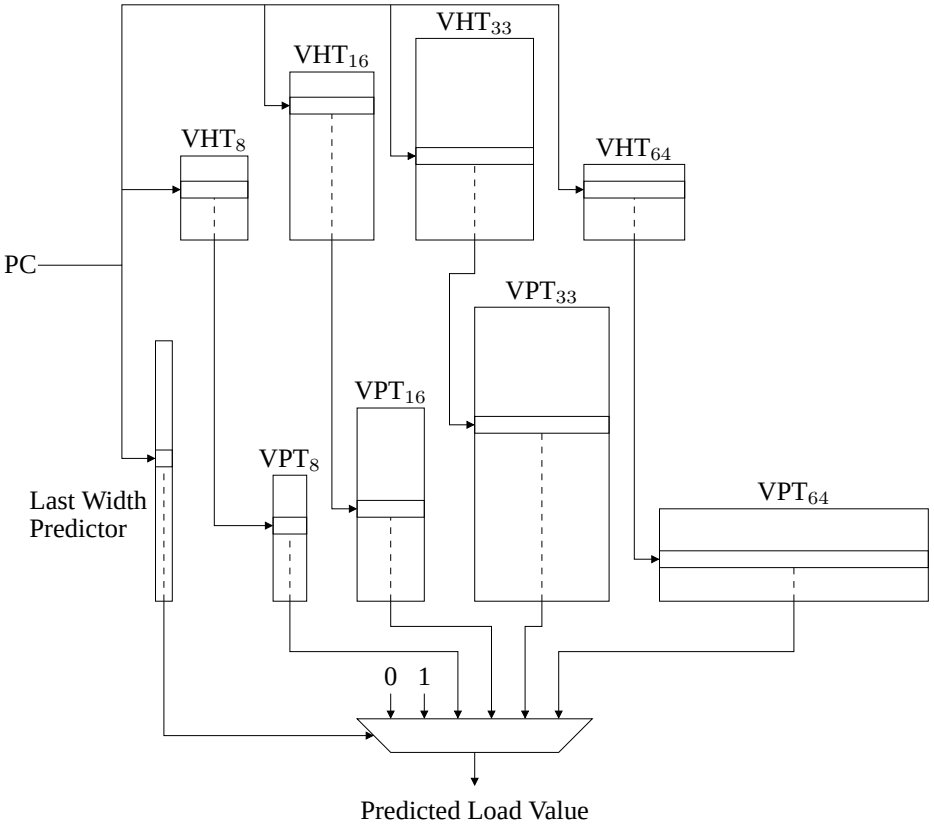

Figure 7: Both the value history table and the value prediction table can be divided into smaller tables based on load-value data-widths.

Benchmark Name (Input Set)	Instructions Skipped	Benchmark Name (Input Set)	Instructions Skipped
bzip2 (program)	45.9B	mcf	20.675B
crafty	44.8B	parser	18.9B
eon (rushmeier)	8.5B	perlbmk (splitmail)	100M
gap	30B	twolf	10.625B
gcc (200)	20B	vortex (2)	13.6B
gzip (graphic)	37.95B	vpr (route)	25.475B

Table 1: The SPEC2000 benchmarks used in this study, along with the input sets and the number of instructions skipped before simulation. All inputs are from the reference set.

For the power savings results, we used CACTI 3.0 [22]. This version of CACTI takes layout considerations into account and provides timing and power results for cache structures. Because we only consider tagless predictor tables, our power results do not include any of the power components due to the tag array and tag matching that CACTI reports. We use the sum of the data array decode, data array wordline and bitline, sense amp, and data output driver power.

Our performance figures were generated using the MASE simulator [11], which is part of the SimpleScalar toolset (pre-release version 4.0). We added support for traditional and width-partitioned last-value and finite context matching value predictors and the associated prediction verification and misprediction recovery. The details of the simulator configuration are explained in Section 4.4.

4.2 WP-LVP

In the rest of this section, we first detail the space and power reduction results for the width-partitioned last value predictor. Then we present similar results for the FCM predictors. Finally, we provide the ILP performance results for both types of predictors.

4.2.1 SPACE SAVINGS

Our first set of results compares our WP-LVP to a conventional LVP. Both do not use tags. The sizing for the individual VPT tables in the WP-LVP use a 4:2:8:1:32 ratio of number of entries in the VPT_8, VPT_{16}, VPT_{33}, VPT_{64} and the last width predictor. For example, our 8KB WP-LVP configuration uses a 512-entry VPT_8, a 256-entry VPT_{16}, a 1K-entry VPT_{33}, a 128-entry VPT_{64}, and a 4K-entry last width predictor. We used the distribution of data-width classes from Figure 1, rounded the number of entries per table to powers-of-two and then further increased some of the tables sizes to make the total storage requirements in kilobytes close to a power-of-two. A corresponding conventional 64 bits-per-entry LVP has eight times the number of entries as the VPT_{64} of the WP-LVP, so the 8KB LVP has 1K entries. Although the last width prediction results from Figure 2 indicate the width predictors with 2K entries are sufficient, we maintain the same table size ratios for all configurations so that comparisons to the conventional LVP are for predictors with close to equal sizes.

Figure 8 shows the value prediction rates for the WP-LVP and the LVP for different hardware budgets. These results are only for last value prediction accuracy and do not make use of any kind of confidence mechanism. Across the range of predictor sizes, a WP-LVP achieves an accuracy that is between a traditional LVP of the same size and a traditional LVP of twice the size. For configurations at 16KB and above, the predictor tables are sufficiently large to contain the working set of load values; further increases in table sizes do not provide many additional correct predictions. Overall, our width partitioning approach can reduce the space requirements of a LVP by approximately one half with a slight loss of accuracy.

We also simulated the load-value predictors in conjunction with a simple counter-based confidence mechanism. We used a PC-indexed table of saturating three-bit counters. On a correct prediction, the counter is incremented by one. On a misprediction, the counter is decremented by three. The counter is in a high-confidence state if its value is greater than or equal to five. Confidence mechanisms are crucial for any practical value predictor implementation because they filter

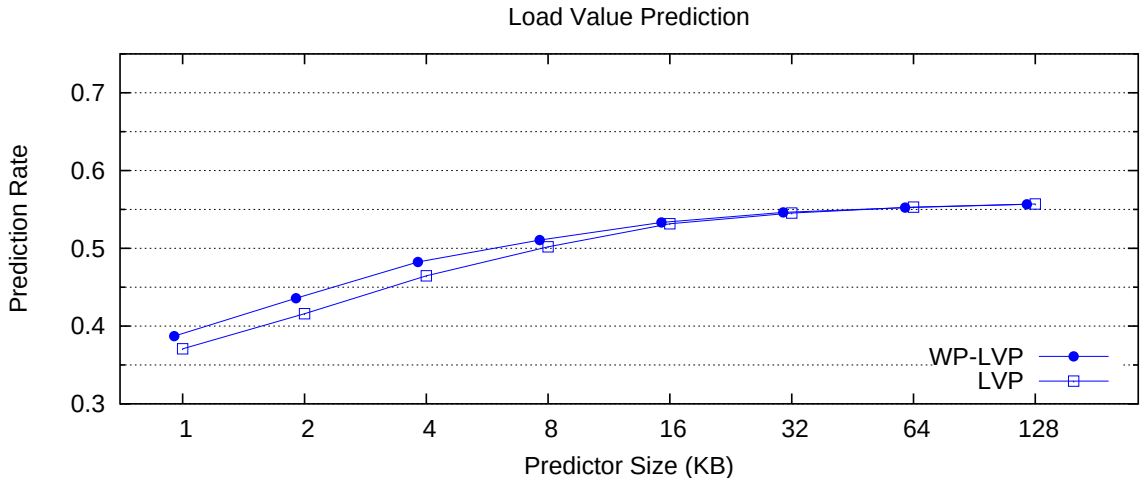

Figure 8: The load-value prediction accuracy for a conventional LVP and the WP-LVP across a range of hardware budgets.

out the difficult to predict loads which would cause many otherwise preventable value mispredictions. The number of entries in the confidence tables simulated are equal to the number of entries in our conventional LVP configuration. Figure 9 shows the value prediction results for the same configurations as in Figure 8, but with these confidence counters. The data bars only show the fraction of loads which are either correctly predicted or mispredicted. All remaining loads have low-confidence and therefore make no prediction. The prediction accuracy trends are similar to the case without the confidence counters, and the fraction of mispredicted loads ranges from 1.33% to 1.52% across all hardware budgets.

4.2.2 POWER SAVINGS

In Section 3, we described how serializing the last width predictor access with the load-value predictor access can reduce the energy consumption of the WP-LVP. The total lookup energy consumption of the serialized WP-LVP is

$$\mathcal{E}_{\text{WP-LVP}} = n_{\text{LWP}} \cdot E_{\text{LWP}} + \sum_{w \in \mathcal{W}} n_w \cdot E_w$$

where n_{LWP} is the number of last width predictor lookups (equal to the total number of load instructions), n_w is the number of loads predicted to be in data-width class w, and E_X is the energy consumed by one access of table X. The set $\mathcal{W}$ is equal to $\{\mathcal{W}_0, \mathcal{W}_1, \mathcal{W}_8, ...\mathcal{W}_{64}\}$. Because width predictions of $\mathcal{W}_0$ and $\mathcal{W}_1$ correspond to the constants zero and one, no additional VPT lookup is needed and therefore $E_0 = E_1 = 0$ pJ. Note that this equation only computes the energy consumption due to predictor lookups. The equation for the energy consumption due to updates is identical, except that n is the number of updates per table. Table 2 lists the energy consumption for a single

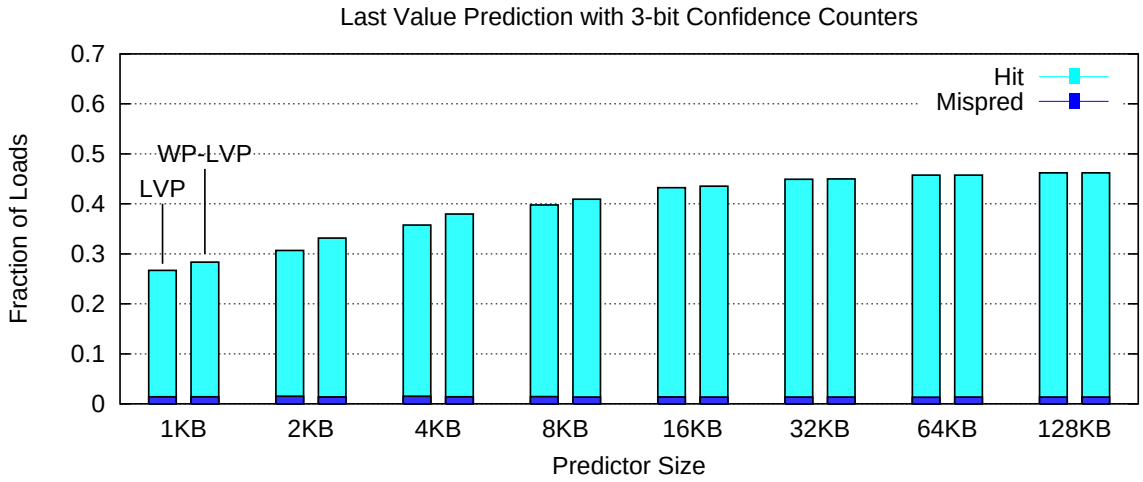

Figure 9: The load-value prediction accuracy for a conventional LVP (left bar) and the WP-LVP (right bar) with three-bit saturating confidence counters. The fraction of loads not classified as "Mispred" or "Hit" in this figure are non-predicted low-confidence loads.

access in each of the prediction tables, as well as for an access in a traditional LVP. The tables are sized for a hardware budget of 8KB.

We measured the numbers of each kind of predictor lookups and updates, with the energy costs of Table 2, computed the overall energy consumption of the value predictors. Figure 10 shows the total value predictor energy consumption for the last value predictors, averaged over the twelve benchmarks. The energy consumption is divided into the energy spent on predictor lookup and predictor update. For the WP-LVP configuration (segmented bars), we provide a further breakdown showing the energy consumed for each individual predictor table. The load width predictor and the VPT_{33} table consume the majority of the energy. The load width predictor must always be accessed, and VPT_{33} has the highest per-access energy cost. There is little difference in energy consumption between the lookup phase and the update phase. The 1KB to 8KB and the 32KB WP-LVP predictors provide a 40.4% to 46.5% reduction in total energy consumption over a conventional last value predictor of the same size. For the 16KB, 64KB and 128KB predictors, the energy reduction ranges from 25.7% to 35.0%.

The power savings results in Figure 10 do not take any sort of confidence mechanism into account. With a confidence mechanism, even greater power savings can be achieved (for either the conventional LVP or the WP-LVP) by serializing the confidence lookup with the value prediction. Any load initially predicted as having low-confidence need not waste any additional energy to perform the value prediction lookup.

Table Name	Number of Entries	Actual Size (KB)	Energy per Access (pJ)
	1KB Predictor		
Last Width Predictor	512	0.1875	14.5
VPT_8	64	0.0625	15.6
VPT_{16}	32	0.0625	25.9
VPT_{33}	128	0.5156	57.1
VPT_{64}	16	0.125	90.9
Conventional LVP	128	1.0	104.4
	2KB Predictor		
Last Width Predictor	1024	0.375	16.7
VPT_8	128	0.125	17.3
VPT_{16}	64	0.125	27.7
VPT_{33}	256	1.0313	61.8
VPT_{64}	32	0.25	93.7
Conventional LVP	256	2.0	112.3
	4KB Predictor		
Last Width Predictor	2048	0.75	24.7
VPT_8	256	0.25	20.3
VPT_{16}	128	0.25	30.7
VPT_{33}	512	2.0625	70.0
VPT_{64}	64	0.5	98.7
Conventional LVP	512	4.0	125.7
	8KB Predictor		
Last Width Predictor	4096	1.5	29.0
VPT_8	512	0.5	24.4
VPT_{16}	256	0.5	34.7
VPT_{33}	1024	4.125	82.4
VPT_{64}	128	1.0	104.5
Conventional LVP	1024	8.0	162.4
	16KB Predictor		
Last Width Predictor	8192	3.0	55.6
VPT_8	1024	1.0	30.1
VPT_{16}	512	1.0	40.6
VPT_{33}	2048	8.25	119.0
VPT_{64}	256	2.0	112.3
Conventional LVP	2048	16.0	205.3
	32KB Predictor		
Last Width Predictor	16384	6.0	70.4
VPT_8	2048	2.0	38.0
VPT_{16}	1024	2.0	48.9
VPT_{33}	4096	16.5	158.7
VPT_{64}	512	4.0	125.7
Conventional LVP	4096	32.0	296.6
	64KB Predictor		
Last Width Predictor	32768	12.0	132.5
VPT_8	4096	4.0	49.0
VPT_{16}	2048	4.0	60.1
VPT_{33}	8192	33.0	250.0
VPT_{64}	1024	8.0	162.4
Conventional LVP	8192	64.0	394.8
	128KB Predictor		
Last Width Predictor	65536	24.0	211.4
VPT_8	8192	8.0	85.6
VPT_{16}	4096	8.0	96.7
VPT_{33}	16384	66.0	343.9
VPT_{64}	2048	16.0	205.3
Conventional LVP	16384	128.0	567.8

Table 2: The number of entries, physical size, and energy cost for each of the predictor tables.

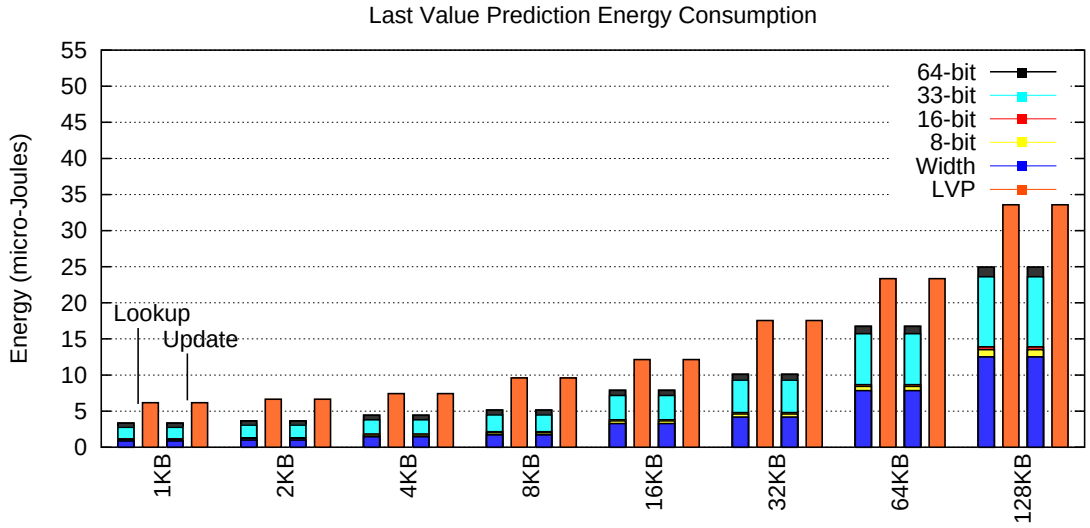

Figure 10: The predictor energy consumption for the WP-LVP (segmented bars) and a conventional LVP (solid bars). For the WP-LVP, the energy measurements are further divided based on the contribution of each component predictor table.

Table Name	1024-entry		4096-entry	
	Entries	Size (KB)	Entries	Size (KB)
VHT_8	1024	3.0	4096	12.0
VHT_{16}	512	3.0	2048	12.0
VHT_{33}	1024	12.4	4096	49.5
VHT_{64}	256	6.0	1024	24.0
VHT (total)	2816	24.4	11264	97.5
FCM VHT	1024	24.0	4096	96.0

Table 3: The number of entries and size (in KB) of the different value history tables.

4.3 WP-FCM

In this section, we present the prediction accuracy results of the PWP-FCM and the FWP-FCM predictors. We compare these predictors to a traditional FCM predictor. All predictors are third-order predictors, and the index and hashing functions used are the improved functions described by Burtscher [3]. For the width-partitioned VPTs, we use the same sizing ratios as for the WP-LVP tables.

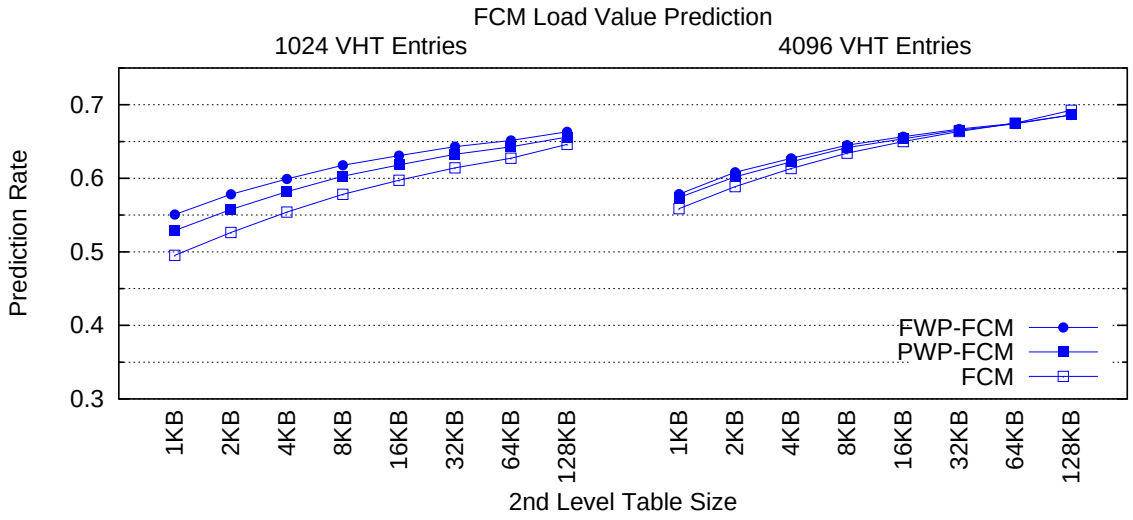

Figure 11: The value prediction accuracy of the FCM predictors for two different sized first-level VHTs. For the fully width-partitioned FCM predictor, the first-level VHTs use the sizings listed in Table 3.

4.3.1 SPACE SAVINGS

We evaluate two classes of predictors. The first class uses a 24KB (1024-entry) VHT for the FCM and PWP-FCM predictors, and the FWP-FCM predictor uses a collection of VHTs with a comparable hardware cost of 24.4KB. The second class uses VHTs with four times as many entries in each table. The exact sizes and number of entries are listed in Table 3. For each class we varied the second-level VPT hardware budget from 1KB to 128KB. Figure 11 shows the prediction accuracy of the FCM, PWP-FCM and FWP-FCM predictors. For the 1K-entry VHT configurations, the width-partitioned VPTs of the PWP-FCM predictor provide some additional accuracy due to the larger effective size of the prediction tables. Using a width-partitioned history table provides a much greater benefit as a 1K-entry VHT still suffers from considerable inter-load interference. Overall, the FWP-FCM predictor provides a factor of four reduction in space requirement over a traditional FCM predictor for approximately the same prediction accuracy.

With a 4K-entry VHT, there is far less interference between unrelated load histories, and so the benefits of the width-partitioned VHT are less pronounced. The FWP-FCM predictor still reduces the space requirements by roughly one half with about the same accuracy as the FCM predictor. As the number of entries in the second level tables increases, the prediction accuracy of the PWP-FCM predictor catches up with and then surpasses that of the FWP-FCM predictor. We believe that while partitioning the load-value history into separate substreams provides space-savings and energy benefits, there are a small number of loads that do require multi-width contexts to be accurately predicted.

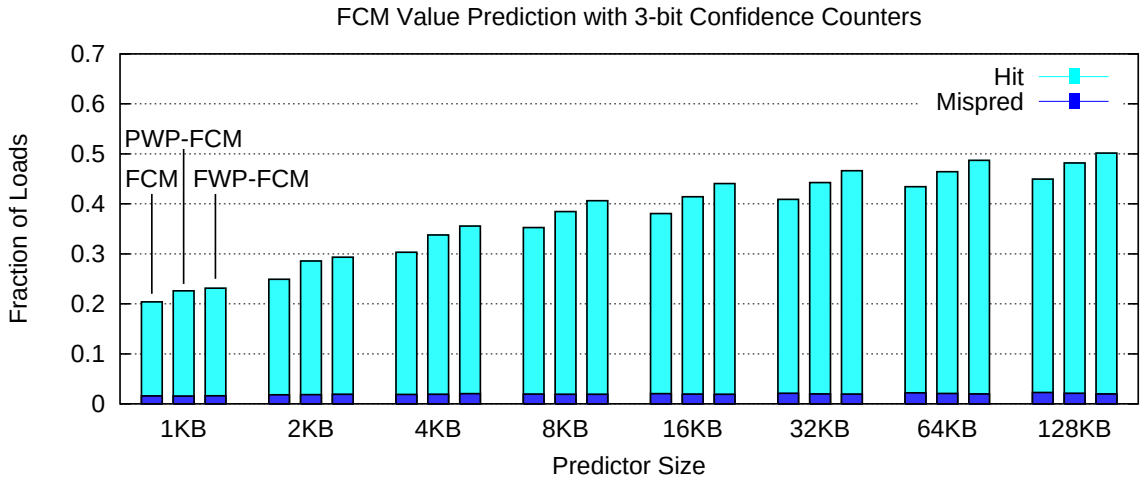

Figure 12: The load-value prediction accuracy for conventional FCM (left bar), PWP-FCM (middle bar) and FWP-FCM (right bar) predictors with three-bit saturating confidence counters. The fraction of loads not classified as "Mispred" or "Hit" in this figure are non-predicted low-confidence loads.

We also simulated the FCM predictor in conjunction with the same three-bit confidence mechanism used for the LVPs in Section 4.2.1. Figure 12 shows the average misprediction rates and the average correct prediction rates. Low-confidence non-predictions make up all remaining loads. Overall, the confidence mechanism keeps the misprediction rates between 1.54% and 2.26%. The misprediction rates for the LVP configurations are lower because the greater number of mispredictions for the LVPs causes the confidence mechanism to label more value predictions as being low-confidence. Labeling more value predictions as low-confidence means that more mispredictions are converted to low-confidence non-predictions.

4.3.2 POWER SAVINGS

For the FCM predictors, we performed an energy consumption analysis similar to that for the last value predictors. We only considered the energy reduction in the FCM second-level value prediction tables. The second-level tables sizes and energy costs are identical to the LVP tables sizes listed in Table 2. Figure 13 shows the energy consumption for both the conventional FCM predictor as well as the width-partitioned FCM predictor over a range of sizes. The second-level energy consumption for the PWP-FCM and the FWP-FCM are identical, and therefore Figure 13 does not display the data separately for the two predictor types. Overall, the trends are very similar to the last value predictors, with the smaller configurations achieving energy reductions from 40.4% to 46.5%, and the larger configurations attaining reductions in the 25.8-35.0% range.

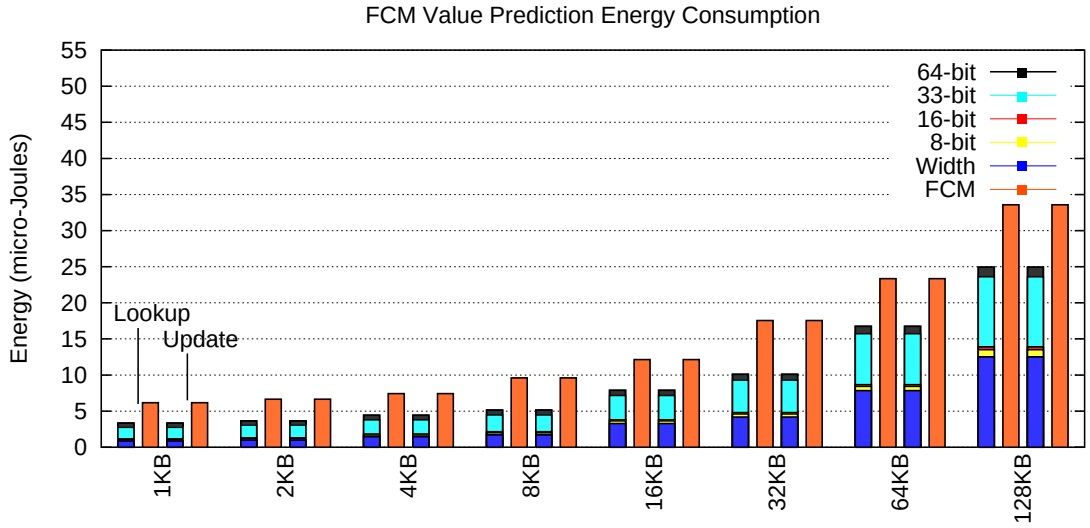

Figure 13: The second-level table predictor energy consumption for the WP-FCM (segmented bars) and a conventional FCM (solid bars). For the WP-FCM, the energy measurements are further divided based on the contribution of each component predictor table.

4.4 ILP Impact

Achieving equal prediction rates for less power does not necessarily mean that the overall performance of the value-predicting processor will be unaffected. A correct value prediction of a load on a program's critical path will provide a greater performance benefit than the correct prediction of a non-critical load. Therefore the prediction rate alone is not sufficient to fully judge the effects of width partitioning on a value predicting processor. To quantify the performance impact of our width-partitioned value predictors, we simulated a superscalar processor supporting value prediction and observed the attained IPC.

We modeled a four-way out-of-order superscalar processor. The configuration parameters are listed in Table 4. The processor supports selective replay of only data-dependent instructions following a value-mispredicted load Although an oracle memory dependence predictor is not implementable, the store sets predictor attains performance levels very close to that of an oracle predictor [6], and so we chose to use an oracle predictor as an approximation of store sets to simplify our simulator. We simulated the same twelve SPEC2000 integer benchmarks with the same inputs and sample windows as have been used throughout the rest of this study.

We simulated the LVP and FCM predictors with 8K-entry (or width-partitioned equivalent) value prediction tables. Figure 14 shows the IPC performance improvement relative to a base processor configuration that performs no value prediction. Overall, FCM predictors perform better than last value predictors, which is not surprising due to the higher prediction rates of the FCM predictors. For either the LVP or FCM predictors, width partitioning has very little effect on the

Parameter	Value
Fetch, Decode, Issue, Commit Widths	4 instructions per cycle
Fetch Queue	32 instructions
Issue Queue	32 instructions
Reorder Buffer	128 instructions
Load Store Queue	64 instructions
Front-end Pipeline Length	6 stages
Scheduler Latency	3 cycles
L1 instruction and data caches	16KB, 4-way, 64 byte line, 3 cycles each
L2 unified cache	512KB, 8-way, 128 byte line, 12 cycles
Memory Latency	100 cycles

Table 4: The simulated processor configuration.

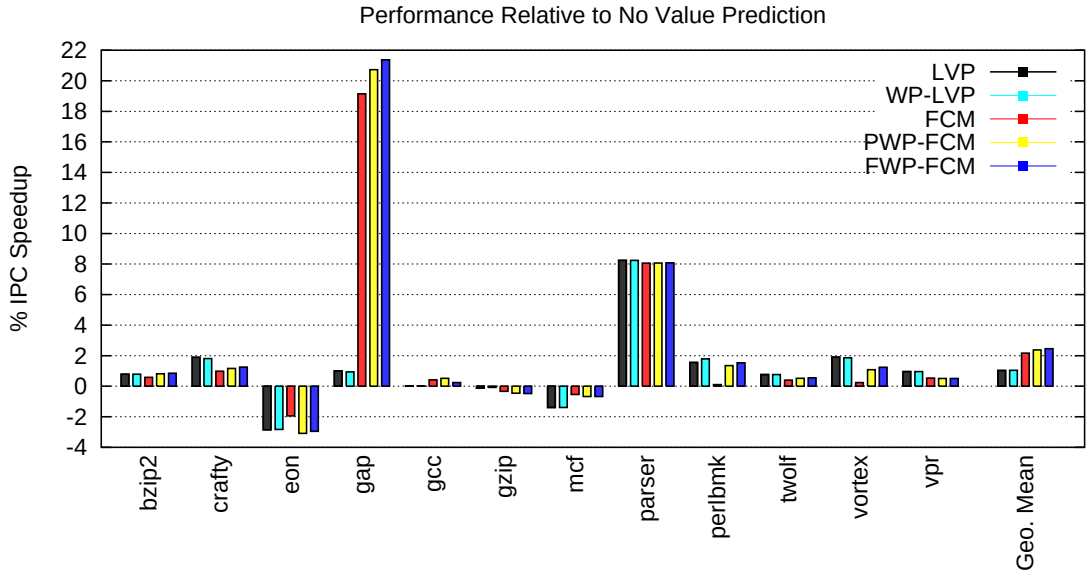

Figure 14: The IPC speedup of each value predictor over a base processor configuration with no load value prediction.

geometric mean IPC. This means that width partitioning is an effective means for reducing the size and energy consumption of value predictors without any negative impact on overall performance. In fact, there is actually a very slight performance improvement of 0.01% for the WP-LVP over the LVP, and 0.29% for the FWP-FCM over the FCM configuration. On a per-benchmark basis, the performance benefits of value prediction has a greater variance, but these differences are mostly independent from width partitioning.

5. Related Work

This research follows from a large body of related work. The most similar research to this paper is the 2-mode predictor of Sato and Arita. The other studies which are most relevant fall into three categories. The first is the existing research on value prediction. The second is on width-based optimizations. The last group is in partitioned hardware structures.

Sato and Arita proposed the *2-mode* value predictor which is the most similar scheme to our width partitioning [19]. The 2-mode predictor uses two prediction tables, one for 8-bit values and another for 32-bit values (the study was performed for a 32-bit architecture). Instead of using a width prediction, all predictor tables use tags and a load can only have a predicted value stored in a single location. Our approach generalizes this idea to any number of width classifications. During the lookup phase, the 2-mode predictor must access both tables in parallel to search for an entry that has a matching tag. Width partitioning saves energy because it only accesses one of the individual LVP tables. Our update procedure is also much simpler because in addition to updating one of the tables with the data value, the 2-mode predictor must also check the other table and invalidate any matching entries. Our width partitioning also allows for cheap prediction of common zero and one valued loads, similar to the 0/1 frequent value predictor [20].

After Lipasti et al.'s seminal load-value prediction study [14], Lipasti and Shen extended the concept to general value prediction of any register value producing instruction [13]. More accurate value prediction algorithms have been proposed such as stride-based predictors, two-level predictors and hybrid predictors [24]. The finite context matching predictor provides even better accuracy by being able to detect and predict repeating patterns in the value history [21].

Morancho et al. proposed using a dynamic load classification scheme to avoid unpredictable loads thus reducing contention in the value prediction tables [17]. Calder et al. proposed filtering out non-critical instructions, thus using the value prediction resources to only target instructions that lie on a program's critical path of execution [5]. Note that our proposed width partitioning approach is orthogonal to these techniques and can be used to further reduce the space and power requirements of any of these more sophisticated prediction schemes.

Many past studies have identified and exploited the fact that data-widths are not uniformly distributed. SIMD instruction set extensions allow the processor to increase the effective execution bandwidth by allowing a program to perform a single operation on multiple sets of narrow-width data in parallel [12, 18]. The Dynamic Zero Compression (DZC) cache reduces energy consumption by using a single bit to indicate that a full byte is zero [23]. Brooks and Martonosi proposed width-based optimizations for operation packing and power savings [1]. Loh introduced data-width prediction to solve the problem of providing width information early enough in the processor pipeline for the dynamic instruction scheduler to perform width-based scheduling decisions [15].

Our value predictor organization uses data-widths to partition the load-value predictor into several smaller but more efficient structures. The idea of partitioning hardware structures also comes up in many other contexts. Hybrid predictors for branch prediction and value prediction partition their predictors into multiple structures, each of which predicts certain classes of instructions better than the others [16, 24]. Our width-partitioned predictors also have a similar structure to way-predicted data caches [4]. A way-predicted cache has multiple direct mapped cache structures (the ways), and a prediction specifies which of these should be accessed. This provides a reduction in power similar

to our WP-LVP by accessing only a single structure. Of great importance to caches, having smaller individual structures can also reduce the access latency for memory operations, although it is much less important for our value predictors.

6. Conclusions

Although value prediction shows some promise for increasing the performance of future processors, the extensive hardware budgets and high power consumption of many proposed prediction algorithms need to be taken into consideration. By dividing a value predictor into several smaller predictors of different widths, we can achieve a more efficient use of the predictor tables by allocating appropriately sized entries based on the actual widths of the values. Furthermore, by only accessing the one table corresponding to the width of the value, we can achieve a significant power savings for the predictor. Our width partitioning technique can potentially be applied to other load-value or general data-value predictors for space reduction and power savings without impacting overall performance.

References

[1] David Brooks and Margaret Martonosi. Value-Based Clock Gating and Operation Packing: Dynamic Strategies for Improving Processor Power and Performance. *ACM Transactions on Computer Systems*, 18(2):89–126, May 2000.

[2] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, University of Wisconsin, June 1997.

[3] Martin Burtscher. An Improved Index Function for (D)FCM Predictors. *Computer Architecture News*, 30(3):19–24, June 2002.

[4] Brad Calder, Dirk Grunwald, and Joel Emer. Predictive Sequential Associative Cache. In *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, pages 244–253, San Jose, CA, USA, February 1996.

[5] Brad Calder, Glenn Reinmann, and Dean Tullsen. Selective Value Prediction. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 64–74, Atlanta, GA, USA, June 1999.

[6] George Z. Chrysos and Joel S. Emer. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 142–153, Barcelona, Spain, June 1998.

[7] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. Pointer Cache Assisted Prefetching. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 62–73, Istanbul, Turkey, November 2002.

[8] Barry Fagin. Partial Resolution in Branch Target Buffers. *IEEE Transaction on Computers*, 46(10):1142–1145, 1997.

[9] Freddy Gabbay and Avi Mendelson. Can Program Profiling Support Value Prediction? In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 270–280, Research Triangle Park, NC, USA, December 1997.

[10] Ricardo Gonzalez and Mark Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

[11] Eric Larson, Saugata Chatterjee, and Todd Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, Tucson, AZ, USA, November 2001.

[12] Ruby Lee. Accelerating Multimedia with Enhanced Microprocessors. *IEEE Micro Magazine*, 15(2):22–32, April 1995.

[13] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 226–237, Paris, France, December 1996.

[14] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value Locality and Load Value Prediction. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Cambridge, MA, USA, October 1996.

[15] Gabriel H. Loh. Exploiting Data-Width Locality to Increase Superscalar Execution Bandwidth. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 395–405, Istanbul, Turkey, November 2002.

[16] Scott McFarling. Combining Branch Predictors. TN 36, Compaq Computer Corporation Western Research Laboratory, June 1993.

[17] Enric Morancho, José María Llabería, and Angel Olivé. Split Last-Address Predictor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 230–239, Paris, France, October 1998.

[18] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture. *IEEE Micro Magazine*, 16(4):51–59, August 1996.

[19] Toshinori Sato and Itsujiro Arita. Table Size Reduction for Data Value Predictors by Exploiting Narrow Width Values. In *Proceedings of the 14th International Conference on Supercomputing*, pages 196–205, Santa Fe, New Mexico, USA, May 2000.

[20] Toshinori Sato and Itsujiro Arita. Low-Cost Value Predictors Using Frequent Value Locality. In *Proceedings of the 4th International Symposium on High Performance Computing*, pages 106–119, Kansei Science City, Japan, May 2002.

[21] Yiannakis Sazeides and James E. Smith. Implementations of Context Based Value Predictors. ECE 97-8, University of Wisconsin-Madison, December 1997.

[22] Premkishore Shivakumar and Norman P. Jouppi. CACTI 3.0: An Integrated Timing, Power, and Area Model. TR 2001/2, Compaq Computer Corporation Western Research Laboratory, August 2001.

[23] Luis Villa, Michael Zhang, and Krste Asanović. Dynamic Zero Compression for Cache Energy Reduction. In *Proceedings of the 33rd International Symposium on Microarchitecture*, Monterey, CA, USA, December 2000.

[24] Kai Wang and Manoj Franklin. Highly Accurate Data Value Prediction Using Hybrid Predictors. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 281–290, 1997.

[25] Tse-Yu Yeh and Yale N. Patt. Two-Level Adaptive Branch Prediction. In *Proceedings of the 24th International Symposium on Microarchitecture*, pages 51–61, Albuqueque, NM, USA, November 1991.

[26] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent Value Locality and Value-Centric Data Cache Design. In *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, Cambridge, MA, USA, November 2000.

[27] Victor V. Zyuban and Peter M. Kogge. Optimization of High-Performance Superscalar Architectures for Energy Efficiency. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 196–205, Rapallo, Italy, July 2000.