

Width Prediction for Reducing Value Predictor Size and Power

Gabriel H. Loh
Georgia Institute of Technology
College of Computing
loh @ cc.gatech.edu

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 table into several smaller structures, and then use data-width prediction to select from these multiple tables. This 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. An 8KB width-partitioned last value predictor achieves nearly the same load-value prediction rates as a conventional 16KB last value predictor, while simultaneously consuming 41.1% less energy. A width partitioned finite context matching (FCM) predictor achieves the same prediction accuracy as a regular FCM predictor with two- to four-times the number of second-level table entries.

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 [11].

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 [21]. 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 [8, 24]. 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 [11]. 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 [12]. 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 simple last value predictor. Section 4 presents our results. Section 5 briefly explains how to extend our technique to strided predictors, and details a width-partitioned finite context matching predictor. Section 6 reviews some of the past research related to our study, and Section 7 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 [12]. 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.

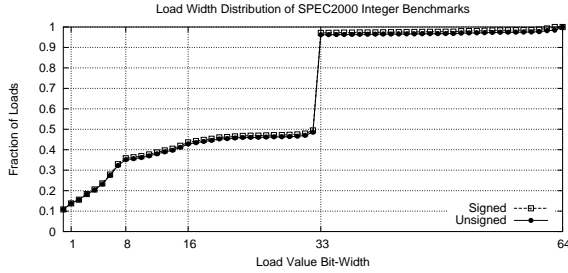


Figure 1: The cumulative distribution of load-value data-widths for the SPEC2000 integer benchmarks. The “signed” distribution corresponds to when the data-width of a value is measured using sign-extension instead of zero-extension.

2.1. 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* [17, 23]. Treating values as signed or unsigned makes little difference to the overall distribution.

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 bound-

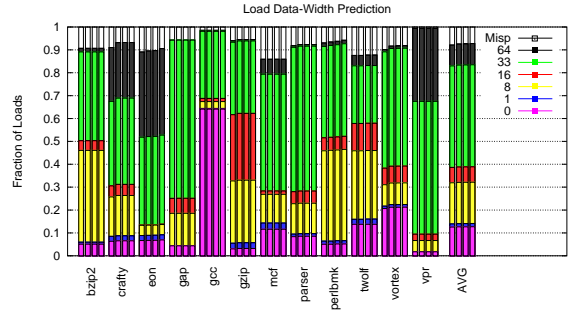


Figure 2: The last width prediction accuracy for the SPEC2000 integer benchmarks. For each benchmark, the four bars correspond to last width predictors with 256, 512, 1024 and 2048 entries (from left to right).

aries, 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 [12], whereas we want to measure the data-width locality of load instructions across six width classes. We use 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.

3. Width Partitioned Predictors

Having demonstrated that load-value data-widths have an uneven distribution and are easily predictable, we now describe one way of using these properties to optimize a simple last value predictor (LVP) [11]. We present the basic concepts in this section with the LVP, and then extend these ideas to a more complex finite context matching predictor in Section 5.

The idea is to employ multiple value prediction tables (VPTs), each with different maximum widths, and then choose between them using the data-width prediction. Figure 3 shows the organization of our *width parti-*

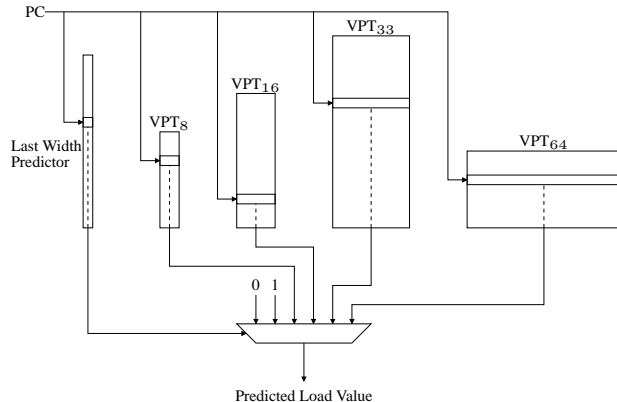


Figure 3: The organization of the width-partitioned last value predictor (WP-LVP). Each table is PC-indexed, and the width predictor selects one of six possible value predictions. Each table may have a different number of entries.

tioned last value predictor (WP-LVP). The WP-LVP uses four different last value predictor tables VPT_8 , VPT_{16} , 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. In parallel, the WP-LVP also performs the individual VPT lookups. 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 one of the VPT outputs.

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 zeros.

The WP-LVP only uses a single value prediction from the multiple VPT outputs. By serializing the last width predictor lookup and the VPT lookup, we can greatly reduce the energy consumption of our 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 mis-predictions, 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 [6], but examining the tradeoff between taglength, predictor size, accuracy and power is beyond the scope of this paper.

4. Results

In this section, we present the performance results for our WP-LVP. We measure prediction accuracy rates for predictors over a range of hardware budgets, and we measure the power savings for a predictor at an 8KB budget.

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.

For the power savings results, we used CACTI 3.0 [19]. 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.

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.

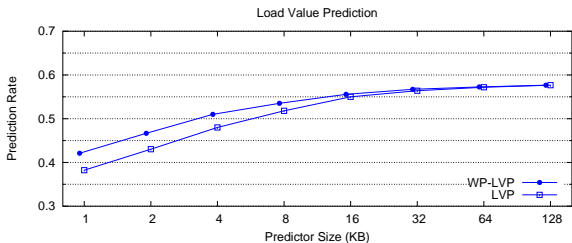


Figure 4: The load-value prediction accuracy for a conventional LVP and the WP-LVP across a range of hardware budgets.

4.2. 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 VPT₈, VPT₁₆, VPT₃₃, VPT₆₄ and the last width predictor. For example, our 8KB WP-LVP configuration uses a 512-entry VPT₈, a 256-entry VPT₁₆, a 1K-entry VPT₃₃, a 128-entry VPT₆₄, 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 VPT₆₄ of the WP-LVP, so the 8KB LVP has 1K entries.

Figure 4 shows the value prediction rates for the WP-LVP and the LVP for different hardware budgets. Note that 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 almost as high as a traditional LVP of twice the size. For configurations at 16-32KB, 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.

We also simulated the load-value predictors in conjunction with a simple counter-based confidence mechanism.

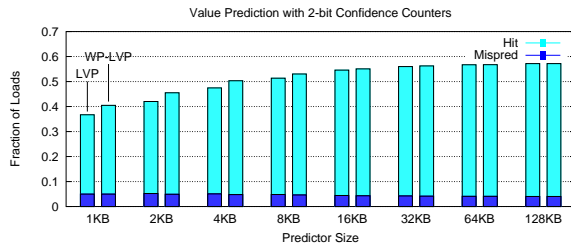


Figure 5: The load-value prediction accuracy for a conventional LVP (left bar) and the WP-LVP (right bar) with two-bit saturating confidence counters. The fraction of loads not classified as “Mispred” or “Hit” in this figure are non-predicted low-confidence loads.

We used a PC-indexed table of saturating two-bit counters. If the counter is in a low-confidence state (zero or one), then no prediction is made. Confidence mechanisms are crucial for any practical value predictor implementation because they filter 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 5 shows the value prediction results for the same configurations as in Figure 4, 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 is about 5% across all hardware budgets. In general, an n -KB WP-LVP predictor still performs nearly as well as a $2n$ -KB conventional LVP.

4.3. Power Savings

In Section 3, we described how serializing the last width predictor lookup with the load-value predictor lookup 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 $\{\mathcal{W}_0, \mathcal{W}_1, \mathcal{W}_8, \dots, \mathcal{W}_{64}\}$. 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 relative energy savings for predictor updates

Table Name	Number of Entries	Actual Size (KB)	Energy per Access (pJ)
Last Width Predictor	4096	1.5	38.37
VPT ₈	512	0.5	24.44
VPT ₁₆	256	0.5	34.69
VPT ₃₃	1024	4.125	82.35
VPT ₆₄	128	1.0	104.45
Conventional LVP	1024	8.0	162.35

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

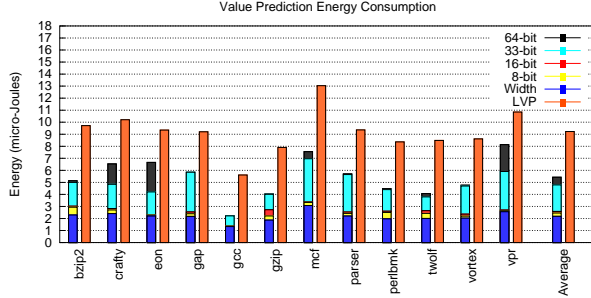


Figure 6: The total predictor lookup energy consumption for the WP-LVP (left bar) and a conventional LVP (right bar). For the WP-LVP, the energy measurements are further divided based on the contribution of each component predictor table.

is similar. Table 2 lists the energy consumption for a single 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 number of each kind of predictor lookup and, with the energy costs of Table 2, computed the overall energy consumption of the value predictors. Figure 6 shows the total value predictor energy consumption for the 8KB predictors. Results for individual benchmarks as well as the overall average are presented. For the WP-LVP configuration (left bar), we provide a further breakdown showing the energy consumed for each individual predictor table. The load width predictor and the VPT₃₃ table consume the majority of the energy. On average, the 8KB WP-LVP provides a 41.1% reduction in total energy consumption compared to a 8KB conventional LVP.

The power savings results in Figure 6 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.

5. Other Value Predictors

In this paper, we have shown how to apply data-width prediction to optimize one of the simplest value predictors. In this section, we briefly explain how to extend these optimizations to a stride-based value predictor, and then we describe in detail a width-partitioned finite context matching (FCM) load-value predictor.

5.1. Stride Predictors

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. Stride predictors 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 [7], such as loop induction variables.

Extending a stride predictor to incorporate data-width predictions is very similar to the approach used for the simpler last value predictor. Instead of a single stride predictor that stores 64-bit values, we can again use multiple stride predictors that each stores values of specific widths. Due to the strong data-width locality, the width of a value is likely to remain unchanged even after adding its stride. Depending on the actual distribution of stride-widths, the number of bits used to store the stride may also be optimized. Because the width-partitioned stride predictor is structurally very similar to the WP-LVP, we will not discuss it any further.

5.2. Width Predicted FCM 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 [13, 22]. Instead of a branch history table for tracking past branch outcomes, FCM uses a *value history table* (VHT) for remembering past load-value patterns. Figure 7 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

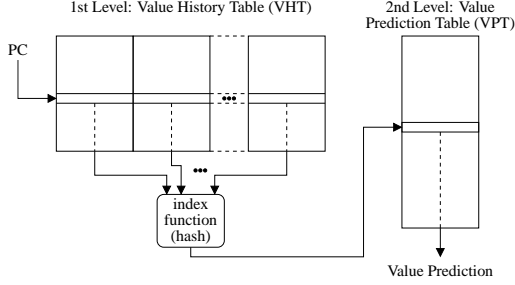


Figure 7: The finite context matching (FCM) predictor uses a hash of the past local value history of a load as an index into the value prediction table.

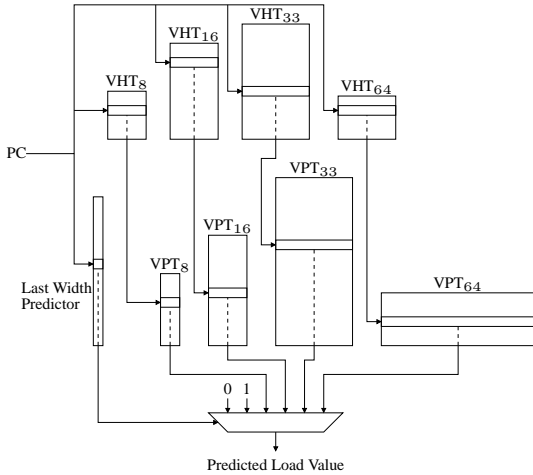


Figure 8: Both the value history table and the value prediction table can be divided into smaller tables based on load-value data-widths.

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 8 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) uses a conventional VHT combined with a width-partitioned VPT. The PWP-FCM may require different hashing functions to index into the different sized second-level prediction tables. The second configuration (shown in Figure 8) is the fully width-partitioned FCM (FWP-FCM), that uses width partitioning for both the VHT and the VPT.

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

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.

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.

5.3. Results

In this section, we present the prediction accuracy results of the PWP-FCM and the FWP-FCM predictors. We also 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. Table 3 lists the VHT sizes for both classes of the FWP-FCM predictor. Note that using the same sizings as the LVP may be suboptimal for the FCM predictors, and this may understate the benefits of width-partitioning the predictor.

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. For each class we varied the number of entries in the second-level VPTs from 1024 entries to 64K entries (or the width-partitioned equivalent). Figure 9 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-

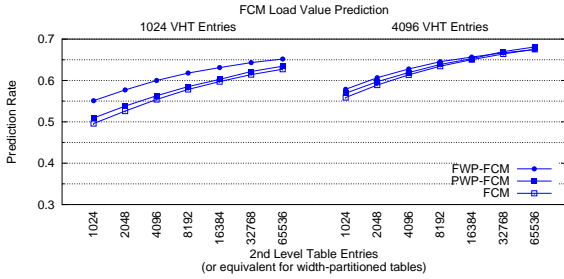


Figure 9: The value prediction accuracy of the FCM predictors for two different sized first-level VHTs. For the fully width-partitioned FCM predictor, the first-level VHTs use the sizings listed in Table 3.

FCM predictor still reduces the space requirements by roughly one half with about the same accuracy as the FCM predictor. In fact, 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 sub-streams provides space-savings and energy benefits, there are a small number of loads that do require multi-width contexts to be accurately predicted.

Due to space constraints, we do not include an energy reduction analysis of the FWP-FCM predictor. Because the table sizings of the FWP-FCM predictor use similar ratios as the WP-LVP predictor, we expect a comparable reduction in energy requirements.

6. 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 [16]. 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 still saves more 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 [17].

After Lipasti et al.’s seminal load-value prediction study [11], Lipasti and Shen extended the concept to general value prediction of any register value producing instruction [10]. More accurate value prediction algorithms have been proposed such as stride-based predictors, two-level predictors and hybrid predictors [21]. The finite context matching predictor provides even better accuracy by being able to detect and predict repeating patterns in the value history [18].

Morancho et al. proposed using a dynamic load classification scheme to avoid unpredictable loads thus reducing contention in the value prediction tables [14]. 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 [9, 15]. The Dynamic Zero Compression (DZC) cache reduces energy consumption by using a single bit to indicate that a full byte is zero [20]. 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 [12].

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 [13, 21]. 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.

7. Conclusions

Although value prediction shows 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.

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] 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.
- [7] 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.
- [8] Ricardo Gonzalez and Mark Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.
- [9] Ruby Lee. Accelerating Multimedia with Enhanced Microprocessors. *IEEE Micro Magazine*, 15(2):22–32, April 1995.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] Scott McFarling. Combining Branch Predictors. TN 36, Compaq Computer Corporation Western Research Laboratory, June 1993.
- [14] Enric Morancho, José María Llbería, 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.
- [15] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture. *IEEE Micro Magazine*, 16(4):51–59, August 1996.
- [16] 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.
- [17] 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.
- [18] Yiannakis Sazeides and James E. Smith. Implementations of Context Based Value Predictors. ECE 97-8, University of Wisconsin-Madison, December 1997.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] Tse-Yu Yeh and Yale N. Patt. Two-Level Adaptive Branch Prediction. In *Proceedings of the 24th International Symposium on Microarchitecture*, pages 51–61, Albuquerque, NM, USA, November 1991.
- [23] 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.
- [24] 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.