An Improved Index Function for (D)FCM Predictors

Martin Burtscher
School of Electrical and Computer Engineering
Computer Systems Laboratory, Cornell University, Ithaca, NY 14853
burtscher@csl.cornell.edu

Abstract

The most promising value predictors to date are the finite context method predictor and a recent improvement thereof, the differential finite context method predictor. Both predictors comprise two levels and the index into the second level is a function of the content of the first level. This index function is crucial for good performance. However, our research shows that the currently used select-fold-shift-xor function performs poorly on range-limited sequences of values. For example, it does not predict the results of byte loads well. The problem with the current function is that it often cannot reach the predictor's entire second-level table. We propose an improved index function that does not suffer from this shortcoming. On the 15 SPECcpu2000 C programs, our new index function improves the average load-value predictability by about 1% to 5% without increase in predictor size. On byte loads, the improvement is over 6% for 4096-entry predictors.

1. Introduction and Motivation

Most high-end microprocessors contain branch predictors to hide the latency associated with determining the direction and/or the target of branch instructions. Some CPUs, such as the Alpha 21264, also contain line, set, and dependence predictors to further improve their performance [3]. We believe it is likely that future CPUs will contain even more predictors. In particular, (load-) value predictors are promising candidates to hide the latency of slow instructions and increase the available instruction-level parallelism (ILP) by breaking dependencies.

One of the most promising value predictors to date is the *finite context method predictor* (FCM) [6, 7]. The *differential finite context method predictor* (DFCM) [2], a variation of the FCM, performs even better because it retains and predicts differences (strides) rather than absolute values. Both predictors contain two tables. The information stored in the first-level table is combined to form an index into the second-level table, which provides the predicted value.

This index calculation is crucial for good performance in both predictors. Currently, most proposed (D)FCMs use the *select-fold-shift-xor* function [7]. However, our research shows that in certain important cases this function performs poorly because it only utilizes a small fraction of the available table space. For instance, Figure 1 shows the average predictability of the byte loads in the 8 SPECcpu2000 C programs that execute a significant number of such loads. The results were obtained using third-order predictors with 2048 lines in the first level. The left-hand-side of the figure shows the FCM predictability for varying second-level table sizes and the right-hand-side the DFCM predictability. Note that for better readability, some of the figures in this paper are not zero based.

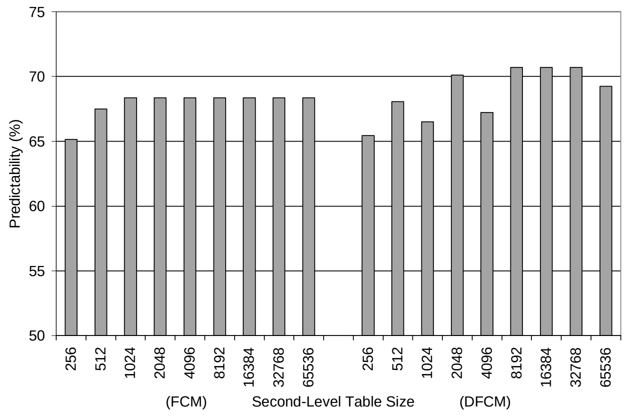

Figure 1: 2048-line, third-order (D)FCM predictability of byte loads.

Figure 1 illustrates the two unexpected results that prompted the work presented in this paper. First, the FCM's predictability is the same for all second-level table sizes of 1024 entries or more. Second, there are several cases where increasing the size of the second level reduces the predictability of the DFCM predictor. Note that this is not an artifact of the averaging as each of the eight programs exhibits these anomalies.

We tracked the problem down to the index function, which cannot reach more than 1024 entries in the FCM and no more than 8192 entries in the DFCM for byte loads (Section 4.1). Moreover, the index function can reach all slots in a DFCM with a 2048- or 8192-entry second-level table, but only 1024 slots with 4096 entries, which explains the low predictability with this table size.

In this paper, we propose an alternate index function that is just as easy to compute but maximizes the utilization of the second-level table. The new index function augments the old one with a *rotate* component. This change does not negatively affect the access time of the predictor because the bit rotation can be done at the time of update (i.e., outside of the critical path) and only amounts to a modification of the wiring in a hardware implementation since the rotate amount is constant. The new index function performs much better on sequences of values of similar magnitude. For 4096-entry predictors, it yields over 6% more predictability on byte loads. When predicting all load instructions, the predictability improvement grows from about 1% for small predictors to over 5% for large predictors.

The remainder of this paper is organized as follows. Section 2 explains the operation of the (differential) finite context method predictor. Section 3 describes the evaluation methods. Section 4 presents our new index function and studies its performance. Section 5 concludes the paper with a summary.

2. The FCM and DFCM Predictors

Both the FCM and the DFCM predictors contain two tables. The first-level table is indexed using PC values. Each line in the first level retains the most recently produced values of the instructions that map to that line. In the DFCM predictor, all values except for the most recently produced value are stored as differences (strides) rather than absolute values. The number of values (or strides in case of a DFCM) per line determines the *order* of the predictor. We indicate a predictor's order by appending the order to the predictor's name, e.g., a third-order FCM would be an FCM3.

In both predictors, the information stored in the first-level table is combined to form an index into the second-level table, which provides the predicted value. Figure 2 illustrates this process for a third-order FCM predictor.

In the DFCM, the predicted value is itself a stride and has to be added to the stored most recently produced value to yield the actual prediction. This makes the DFCM better suited for predicting strided sequences of values [2].

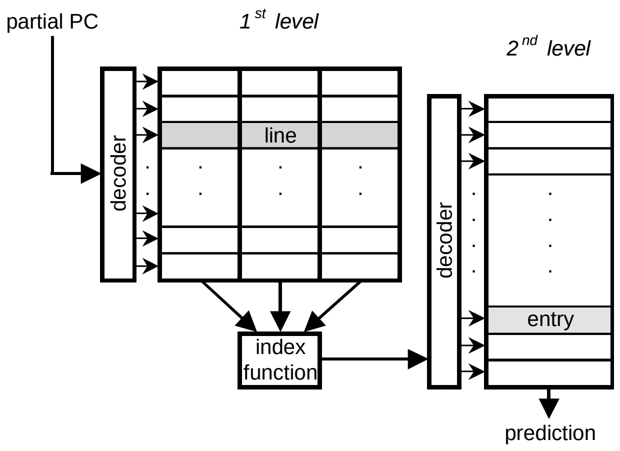

Figure 2: Schematic of a third-order FCM predictor.

Most proposed implementations of (D)FCM predictors use the select-fold-shift-xor index function [2, 4, 5]. The following example illustrates how to compute this function for the three 64-bit values val1, val2, and val3 from a line of an FCM3 with a 4096-entry second level. The symbol "⊕" represents XOR and the subscripts refer to bit positions.

$$\textit{hash}(\text{val}) = \text{val}_{63..60} \oplus \text{val}_{59..50} \oplus \text{val}_{49..40} \oplus \text{val}_{39..30} \oplus \text{val}_{29..20} \oplus \text{val}_{19..10} \oplus \text{val}_{9..0}$$

$$\textit{index}(\text{val1, val2, val3}) = \textit{hash}(\text{val1}) \oplus \textit{hash}(\text{val2}) \ll 1 \oplus \textit{hash}(\text{val3}) \ll 2$$

Each value from the selected line is broken down into n-bit chunks. If the last chunk is shorter than n bits, it is (conceptually) zero padded to n bits. The chunks are then XORed to yield an n-bit hash value. The three hash values are shifted by zero, one, and two bits, respectively, and are then XORed again to form an (n+2)-bit index. Hence, for a second-level table with *size* entries, $n = 1+\log_2(\textit{size})-\textit{order}$. In the above example, $n = 10$.

During the prediction of an instruction, an index is computed using the three values from the line selected by the instruction's PC. This index is then used to access the second-level table, which provides the predicted value.

During updates, the same index calculation is performed, the corresponding second-level entry is overwritten with the update value, and the update value is shifted into the selected line of the first-level table, whereby the oldest value in that line is lost.

One advantage of the select-fold-shift-xor function is that part of it (i.e., *hash*(val)) can be computed before the information is inserted into the predictor. Since *hash*(val) always yields an *n*-bit result, each line in the first level of the predictor only needs to store *order* *n*-bit values rather than *order* 64-bit values, which reduces the predictor size substantially and speeds up the index computation.

3. Evaluation Methods

This study was performed on an Alpha 21264A-based machine. All programs were compiled using Compaq's C compiler V6.3-025 on Tru64 UNIX V5.1 and optimized with "-arch host -non_shared -fast -O2" plus feedback optimization. We used ATOM V2.75 [1, 9], a binary instrumentation toolkit, to evaluate the various predictor configurations. ATOM captures the committed user-mode instructions of the instrumented applications and libraries.

The 15 SPECcpu2000 C programs served as our benchmark suite. Four of them are floating-point intensive (*mesa*, *art*, *equake*, and *ammp*). Table 1 summarizes relevant information about these programs. It shows the static number of load instructions in the binaries (in thousands), the number of load instructions executed when running the train inputs (in millions), and the percentage of the executed load instructions that are byte loads (LDBU).

	load instructions		
program	static	dynamic	byte loads
gzip	11.0k	10,920M	**22.5%**
vpr	20.6k	6,151M	0.1%
gcc	157.2k	813M	0.0%
mcf	11.4k	2,218M	0.0%
crafty	22.8k	6,607M	**12.9%**
parser	22.6k	2,467M	**11.2%**
perlbmk	47.8k	18,686M	**13.7%**
gap	38.4k	2,009M	**9.7%**
vortex	47.7k	3,586M	0.4%
bzip2	11.1k	13,310M	**52.0%**
twolf	27.0k	2,734M	**23.5%**
mesa	38.2k	12,319M	**29.8%**
art	11.7k	819M	0.3%
equake	12.6k	9,164M	0.0%
ammp	17.2k	10,266M	0.4%

Table 1: Information about the 15 benchmark programs.

Table 1 shows that each of the 15 programs contains over ten thousand load sites and executes more than 800 million load instructions. The eight programs in bold print execute over 9.5% byte loads. Only these eight programs are used for the byte-load studies in this paper since the remaining seven programs execute less than 0.5% byte loads.

To keep the simulation time reasonable, we use the SPEC train inputs for running the programs. All programs are executed to completion. Averages refer to the arithmetic mean.

All integer and floating-point load instructions are predicted unless otherwise indicated. Loads to the zero registers (R31 and F31) as well as load-address instructions (LDA and LDAH) are excluded from our study since they are either NOPs, prefetches, or load immediates.

4. Results

The following subsections describe the results. Section 4.1 investigates how many table entries the index functions can reach. Section 4.2 describes our new index function. Section 4.3 compares the performance of the old and the new index functions on byte loads and Section 4.4 on all loads.

4.1 Second-Level Table Reachability

The results from Figure 1 suggested to us that for byte loads, the conventional FCM3 index function might not be able to reach more than 1024 table entries. To test this hypothesis, we counted the number of accessible entries using an exhaustive search. Since we are investigating unsigned byte loads, each of the three numbers in a line of the FCM3's first level can hold a value between 0 and 255. Hence, there are 256^3 or about 17 million possibilities. Figure 3 plots the fraction of the second-level table that is reachable with at least one combination of three values for different table sizes.

Clearly, the conventional index function can only access the entire second level if it is no larger than 1024 entries. With 2048 entries, only 50% of the entries are reachable, with 4096 only 25%, and so on. In absolute numbers, the conventional index function cannot address more than 1024 entries for byte loads, regardless of the actual table size (see explanation below).

The lighter graph in Figure 3 shows the results for our new index function, which is described in more detail in the next section. It can access every table entry up to a table size of 256^3, which is the maximum given three 8-bit numbers ($2^{8*3} = 256^3$).

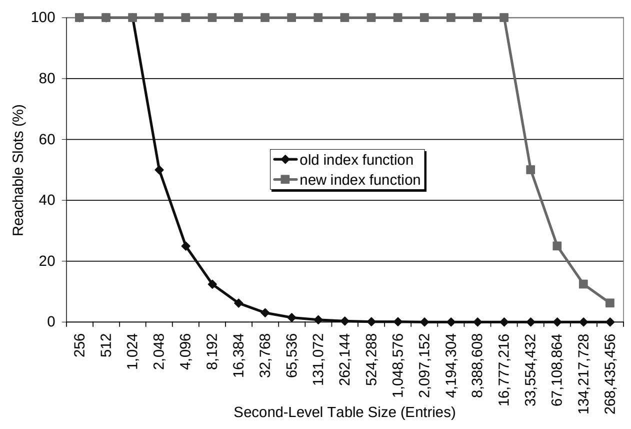

Figure 3: Byte-load reachable fraction of the second-level table of a FCM3 predictor.

Figure 4 is similar to Figure 3 except it is based on a DFCM3. Because this predictor retains differences in its first-level table, the three numbers (strides) in a given line can hold values in the range of -255 to 255 for byte loads. Hence, there are 511^3 (about 133 million) possible combinations that we had to evaluate to obtain the results for Figure 4.

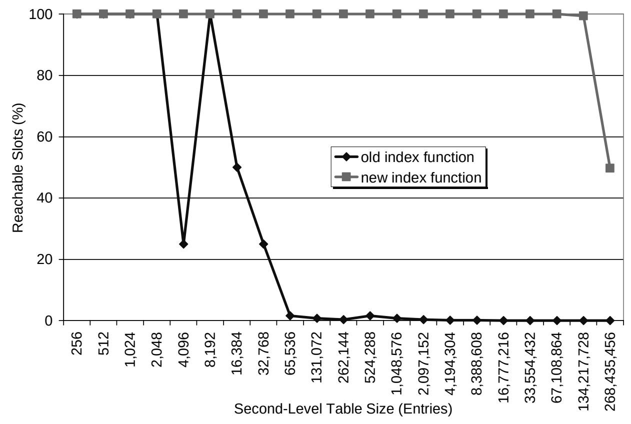

Figure 4: Byte-load reachable fraction of the second-level table of a DFCM3 predictor.

Figure 4 shows that the conventional index function cannot reach all elements in DFCM3 tables with more than 8192 entries. Again, the new index function does not suffer from this problem. It can access every table entry up to a table size of 511^3. Note that the second to last data point (corresponding to a table size of 512^3) is 99.4% and not 100%.

Interestingly, Figure 4 reveals cases where larger tables result in fewer accessible entries than smaller tables. In particular, tables with 4096, 65536, 131072, and 262144 entries result in only 1024 accessible entries. This anomaly occurs whenever the numbers stored in the first level are broken down into an even number of complete blocks before they are XORed into a hash value (Section 2). For example, with a 4096-entry DFCM3, each number is broken down into six 10-bit chunks and one 4-bit chunk. Since the numbers in the first level are between -255 and 255, the two most significant bits in the first (least significant) 10-bit chunk are always identical. Furthermore, they are the same as all the bits in all the other chunks. Since there is an even number of complete chunks (full 10-bit length), the two most significant bits of the XOR result (the hash value) will always be zero. Because this happens with all three values from the selected line, the two most significant bits of the final 12-bit index will also be zero. Hence, there are effectively only 10 useful bits, which is why the index function can only reach 2^{10} or 1024 entries. Similar reasoning shows that all DFCM table sizes that result in an even number of complete chunks and a partial chunk that is no longer than 8 bits suffer from this problem, as well as all table sizes (≥1024 entries) in the FCM.

Unfortunately, the problem is not restricted to byte loads. Any sequence of range-limited values is affected, including sequences of values that are not clustered around zero such as code and data addresses.

4.2 Enhanced Index Function

The problem with the conventional select-fold-shift-xor index function is that it does not provide enough useful bits in the higher bit positions of the index for range-limited values. Fortunately, this shortcoming is not due to the lack of available bits but only because the conventional index function does not spread the (useful) bits evenly when computing the index. As a remedy, our approach rotates the hash values in the first predictor level by $\lfloor 0*n/order \rfloor$, $\lfloor 1*n/order \rfloor$, …, $\lfloor (order-1)*n/order \rfloor$ bits to obtain a more even spread, where $n = 1+\log_2(size)-order$. For any given (D)FCM predictor, the *order* and the second-level table size (*size*) are fixed, meaning that the rotate amounts are constants. Hence, they can be hardwired into the predictor. A simple rewiring would automatically rotate the hash values when they are shifted into the next field during a predictor update. Note that our new index function retains the beneficial property of allowing the computation of the hash values before they are inserted into the first-level table.

The following example shows how to compute the new index function for a third-order FCM with a 4096-entry second-level table (the function *hash* is the same as before).

$$\mathit{index}(\text{val1}, \text{val2}, \text{val3}) = \mathit{hash}(\text{val1}) \oplus \text{rot}(\mathit{hash}(\text{val2}),3) \ll 1 \oplus \text{rot}(\mathit{hash}(\text{val3}),6) \ll 2$$

4.3 Performance on Byte Loads

Figure 5 shows the predictability of byte loads for both the old and the new index function. The results are averages over the 8 byte-load benchmark programs and were obtained using third-order (D)FCMs with 2048 lines in the first level.

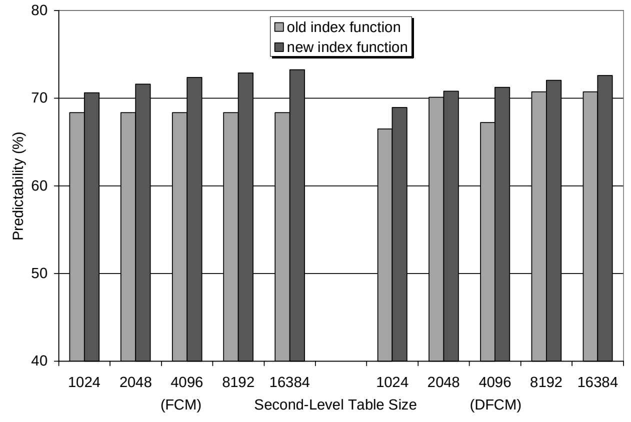

Figure 5: Byte-load predictability of 2048-line (D)FCM3 predictors.

Figure 5 shows that the new index function outperforms the old one for all investigated configurations. Moreover, the predictability continuously improves with increasing second-level table sizes. There is no case where a larger table results in lower performance than a smaller table.

4.4 Performance on All Loads

Thus far, our study has been limited to byte loads to illustrate the problem with the conventional index function. However, many load instructions do not fetch range-limited values. Figure 6 demonstrates that our new index function also outperforms the old function when predicting every load. The results were obtained using third-order predictors with infinite first-level tables to exclude aliasing effects. Note that the results in this subsection are averages over all 15 SPECcpu2000 C programs.

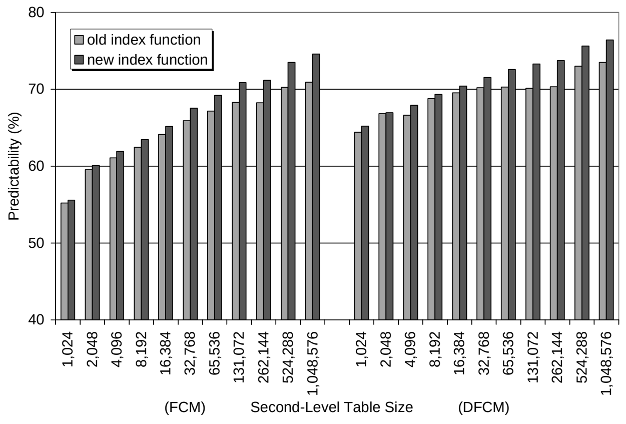

Figure 6: Load-value predictability of infinite-line (D)FCM3 predictors.

From Figure 6 we see that the new index function delivers more predictability than the old one. The performance of the new index function continuously increases with larger table sizes, which is not the case with the old index function for DFCM3. Furthermore, the performance benefit over the old index function grows with the table size as more and more loads fall into the category of range-limited loads. While the larger table sizes presented in Figure 6 are probably beyond what is feasible in hardware, they are certainly in the realm of software predictor implementations [8].

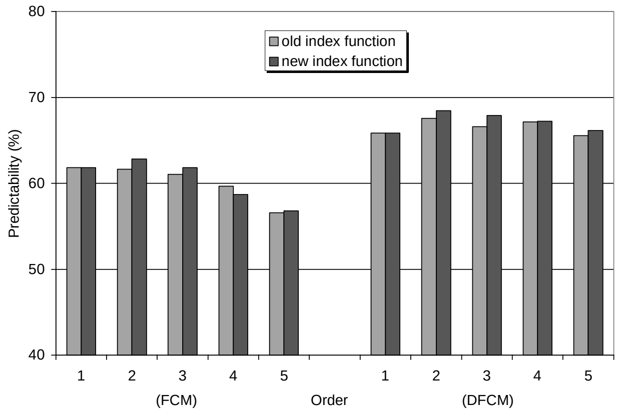

Figure 7: Infinite-line, 4096-entry (D)FCM load-value predictability.

So far, we have only studied third-order (D)FCM predictors. Figure 7 shows results for other orders with infinite first-level and 4096-entry second-level tables.

The advantage of the new index function is not restricted to third-order (D)FCMs. In fact, it is most pronounced for the best-performing second- and third-order predictors. There is no difference in predictability for first-order predictors since the two index functions are identical in this case.

We now see a case where the old index function outperforms the new one. The forth-order FCM favors the old index function for the given predictor configuration, showing that the new index function does not always yield better results.

However, Figure 8 shows that the new index function provides a higher predictability than the old one for implementable (D)FCM3 predictors with realistic sizes, even when all loads are predicted.

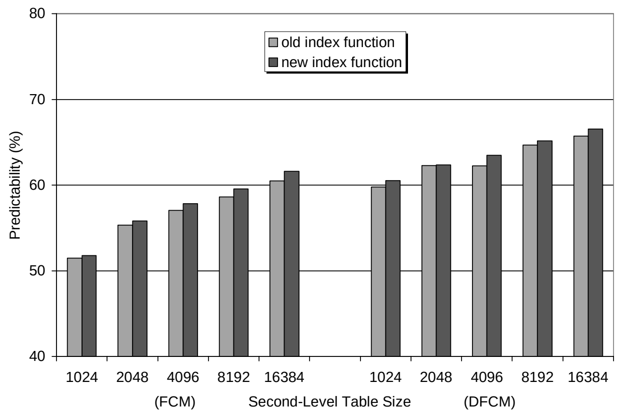

Figure 8: Load-value predictability of 2048-line (D)FCM3 predictors.

5. Summary and Conclusions

This paper proposes a simple change in the wiring of the select-fold-shift-xor index function commonly used in FCM and DFCM value predictors. This change improves the table utilization without negatively affecting the access time. The new index function outperforms the old one in most cases, in particular on sequences of range-limited values. For example, 4096-entry, third-order (D)FCM predictors yield over 6% more predictability on byte loads with the new index function. When predicting all load instructions, the predictability improvement ranges from about 1% for small predictors to over 5% for large predictors with no additional hardware.

In future work we will study the new index function in connection with confidence estimation, investigate other rotate/shift combinations, and evaluate the performance of the new index function on cycle-accurate simulators.

References

[1] A. Eustace, A. Srivastava. "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools." *WRL Technical Note TN-44, Digital Western Research Laboratory, Palo Alto*. July 1994.

[2] B. Goeman, H. Vandierendonck, K. Bosschere. "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency." *7th International Symposium on High Performance Computer Architecture*. January 2001.

[3] R. E. Kessler, E. J. McLellan, D. A. Webb. "The Alpha 21264 Microprocessor Architecture." *1998 International Conference on Computer Design*. October 1998.

[4] G. Reinman, B. Calder. "Predictive Techniques for Aggressive Load Speculation." *31st Annual ACM/IEEE International Symposium on Microarchitecture*. December 1998.

[5] B. Rychlik, J. Faistl, B. Krug, J. P. Shen. "Efficacy and Performance Impact of Value Prediction." *1998 International Conference on Parallel Architectures and Compiler Technology*. October 1998.

[6] Y. Sazeides, J. E. Smith. "The Predictability of Data Values." *30th Annual ACM/IEEE International Symposium on Microarchitecture*. December 1997.

[7] Y. Sazeides, J. E. Smith. "Implementations of Context Based Value Predictors." *Technical Report ECE-97-8, University of Wisconsin-Madison*. December 1997.

[8] E. Speight, M. Burtscher. "Delphi: Prediction-Based Page Prefetching to Improve the Performance of Shared Virtual Memory Systems." To appear in the *2002 International Conference on Parallel and Distributed Processing Techniques and Applications*. June 2002.

[9] A. Srivastava, A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools." *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. June 1994.