DFCM++: Augmenting DFCM with Early Update and Data Dependence-driven Value Estimation

Nayan Deshmukh*, Snehil Verma*, Prakhar Agrawal*, Biswabandan Panda, Mainak Chaudhuri
Indian Institute of Technology Kanpur,
{ndesh, snehilv, pkhrag, biswap, mainakc}@iitk.ac.in

ABSTRACT

Value prediction is one of the promising micro-architectural techniques to improve the processor performance. Through this paper, we provide a series of four enhancements that we apply on top of Differential Finite Context-Method (DFCM) value predictor and call it DFCM++. Our design achieves a geomean IPC of 4.11 whereas the baseline system, without any value predictor, provides a geomean IPC of 3.21 (an improvement of 28.1%). In comparison to the baseline DFCM, which provides a geomean IPC of 2.93, DFCM++ delivers an improvement of 40.2%. Additionally, we show the effectiveness of our enhancements on some of the state-of-the-art value predictors such as VTAGE and DVTAGE.

1. INTRODUCTION

True data dependencies cause frequent stalls in the processor pipeline, resulting in significant performance degradation. Value prediction is one of the techniques that minimizes the impact of data dependencies and improve Instruction Level Parallelism (ILP), hence the system performance. For example, a perfect value predictor (as per the CVP framework) provides a geomean IPC of 6.96 whereas a baseline with no value prediction provides a geomean IPC of 3.21. This gap in performance magnifies the importance of a value predictor.

In this paper, we propose a series of enhancements on an existing value predictor named DFCM (Differential Finite Context-Method) predictor [3] and call it as DFCM++. The enhancements are as follows:

1. An early update policy where we update the value predictor before the instruction commits (Section 3.1).
2. A data-dependence driven value estimator that estimates the value (does not predict) (Section 3.2).
3. A PC blacklister that selects a PC for which we should not predict (Section 3.3).
4. We introduce the notion of dynamic context length for the DFCM predictor (Section 3.4).

In a nutshell, compared to a system with no value prediction, DFCM++ improves the performance (in terms of geomean of normalized IPCs) by 28.1%, attaining a geomean IPC of 4.11. Compared to an unlimited baseline DFCM (that has a geomean IPC of 2.93), DFCM++ improves the performance (geomean of normalized IPCs) by 40.2%.

*The first three authors contributed equally to this paper

2. BACKGROUND

Lipasti et al. [4][5] and Gabbay et al. [2] independently introduced value prediction in the year 1996. Sazeides and Smith defined two different types of predictors [9] based on the criteria that they use to predict values:

- Computational predictors: The predictors that predict the next values based on the output of the operations performed on the previous value(s). E.g: last value [4], stride [2], and 2-delta stride [1] predictors.
- Context-based predictors: The predictors that predict the next value by matching the recent value history of the instruction Program Counter (PC) with the value history based on the observed patterns. E.g: Finite Context-Method (FCM) [8] and last n-values [4] predictor.

We design our predictor based on DFCM predictor as DFCM is a simple and yet effective predictor. We compare the performance of FCM (geomean IPC of 2.3) with DFCM (geomean IPC of 2.93) for 135 traces and find that DFCM outperforms FCM.

2.1 DFCM

The DFCM predictor is derived from the FCM predictor and is more efficient than FCM in terms of performance gain. Unlike FCM, that has two hardware tables, DFCM has three hardware tables.

For the sake of simplicity, we name the three hardware tables of the DFCM predictor as follows: (i) Stride History Table (SHT, 1st level), (ii) Stride Prediction Table (SPT, 2nd level), and (iii) Last Value Table (LVT). SHT is indexed with the instruction PC and stores the *local history* of strides corresponding to the instruction. Each entry is tagged with a hashed PC. SPT contains the actual prediction, in the form of stride, and its corresponding confidence. To index into SPT, we hash the stride history stored in the SHT. This hash effectively represents the stride pattern that the values follow. LVT stores the last value corresponding to the instruction PCs. *Note that, in our implementation we use only two tables* *(i) SHT (1st level, where the LVT is combined with the SHT) and (ii) SPT (2nd level)*

In contrast, the FCM uses two tables: Value History Table (VHT) and Value Prediction Table (VPT). VHT and VPT of the FCM correspond to SHT and SPT of the DFCM.

In [8], an n^{th} order FCM is defined as the one that keeps n values for determining the context of a particular instruction PC. Similarly, an n^{th} order DFCM contains last n strides with respect to an instruction PC. Figure 1 shows the basic DFCM predictor that we use for our enhancements.

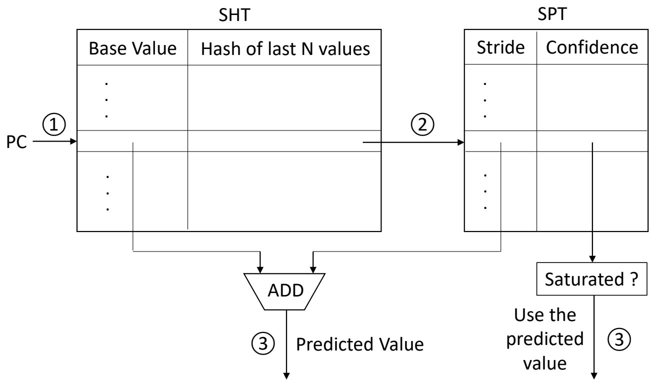

Figure 1: Order N Differential FCM predictor with SHT combined with LVT (1st level) and SPT (2nd level).

We find that the DFCM predictor learns the value/stride pattern better than other predictors such as FCM and 2-delta stride. 2-delta stride predictor provides a geomean IPC of 2.83, FCM predictor provides a geomean IPC of 2.30 and DFCM predictor provides a geomean IPC of 2.93. All the predictors use unlimited hardware structures. As DFCM outperforms FCM and 2-delta stride, we choose DFCM predictor for our enhancements. We elaborate on our enhancements in the following section.

3. FROM DFCM TO DFCM++

We extend the DFCM predictor with four enhancements. These enhancements can be augmented with any value predictor to further boost the performance. An obvious enhancement comes with the unlimited storage track, which is to use unlimited size SHT and SPT. This is a simple enhancement mainly targeted for the unlimited track of this championship. We start off with a basic DFCM predictor and make each table (SHT and SPT) unbounded (based on the requirement all the tables grow). This enhancement reduces the number of conflicts in the SHT and SPT. This makes the tag and the replacement policy within the tables unnecessary.

3.1 Enhancement 1: Early Update (EU)

The early update policy provides us an opportunity to update the predictor even before the instruction gets committed, which helps in back-to-back predictions. In this policy, we add a *predictive state* for each PC that contains a base value and a hash of last N strides. For a normal instruction, the predictive state is same as its corresponding SHT entry. However, for the in-flight instructions, the predictive state contains the speculative value per PC (per SHT entry).

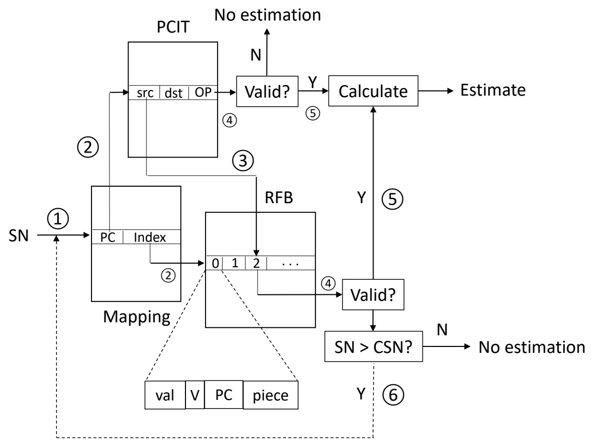

Figure 2: Schematic of Value Estimator. SN: sequence number, CSN: sequence number of the last committed instruction, val: value, V: valid bit, PC: program counter.

Implementation of early update: We simply update the predictive state speculatively using the predicted value. For each entry, if there are no in-flight instructions, we restore the predictive state to the corresponding SHT entry at the commit time. Since we update the SPT entries only at the commit time, we do not pollute the SPT table in case of unknown predictions. Note that, we use the first level table (SHT) just to index into the second level table (SPT) and update the SPT table only at the commit time, once we get the final value.

3.2 Enhancement 2: Value Estimator (VE)

One of the primary enhancements of DFCM++ is the value estimator which is driven by data dependencies. Our value estimator contains four hardware structures: (i) PC information table (PCIT), (ii) Register File Buffer (RFB), (iii) sequence-no-to-PC mapper (here no is used as an alias for number), and (iv) RFB-index-to-sequence-no mapper. PCIT (unlimited in size) stores the relevant information, such as source and destination registers involved with the PC, operation, values of previous occurrences. This table is indexed by the PC and the piece. RFB is a limited size buffer (500 entries), which is indexed by the output of the *sequence-no-to-PC mapper*, which itself is indexed by the sequence number of the dynamic instruction (say I1). RFB stores the state (value of each register, and the valid/invalid state of each register; in case of invalid, it stores the sequence number of the instruction that has overwritten the value) of the register file before the execution of the instruction I1. Sequence-no-to-PC mapper maps a sequence number to a <PC,index> and the index field is used for indexing into the RFB. RFB-index-to-sequence-no mapper maps an RFB entry to a sequence number. We use the mappers to maintain the RFB entries.

Through the PCIT, we try to infer the operation (example an ADD operation) based on the past instances of of the source and destination values corresponding to the instruction PC. Once we determine the operation, given the source register values, we estimate the destination register value.

Figure 2 shows the working of our value estimator. In (①), the sequence-no-to-PC mapper maps a sequence number to a <PC,index>, where the index field is used to index into the RFB (②) and the PC is used to index into the PCIT. In ③, we extract the source registers from a PCIT entry and get their corresponding values from RFB. (④) If the states of the registers are valid, we get the operation field from the PCIT and check for its validity. In case of a valid entry in the operation field, we estimate the value for the destination register (⑤); however, we do not estimate if the operation field is invalid. If the state of the registers is invalid (⑥), we check if the sequence number, corresponding to the instruction that has overwritten the register, is greater than the last committed sequence number. If so, then we repeat the steps from ① to ⑤ till we get the source register value. This is recursive process, and through this the value estimator exploits the data dependence between a set of dynamic instructions. If at any point, we fail to retrieve the value of the source register, we do not estimate. In case the sequence number is less than the last committed sequence number, we do not estimate the value as every committed instruction updates its register values and we need not go beyond this sequence number to estimate.

Allocation of new RFB entry: We implement a FIFO policy to replace the old RFB entry and use the RFB-index-to-sequence-no mapper to invalidate the entry in the sequence-no-to-PC mapper. Also we update the RFB-index-to-sequence-no mapper for the current index with the new sequence number.

Prioritization: While predicting the value, we prioritize the estimator over the speculative predictor (base DFCM) i.e., first we attempt to estimate the value using the estimator and if we are unable to do so then we use the speculative predictor, based on its confidence. There are cases where we may have to use just the DFCM predictor, as we may not have the behavior of the PC all the time or the value in the register file may be stale (for example, if the previous instruction was not the part of prediction or if the value prediction was wrong then we do not use the estimator).

3.3 Enhancement 3: PC Blacklister (PCB)

We observe that even a little increase in mis-predictions deteriorates the performance significantly. We also record some instruction PCs on which we mis-predict quite often. To overcome this issue, we use a simple but effective method to improve performance. The idea is to blacklist an instruction PC if it is mis-predicted consecutively for more than a threshold value. We tune this threshold to select the best one (details in the Section 4.3).

3.4 Enhancement 4: Dynamic Context Length (DCL)

Our last enhancement is to dynamically determine the context length (number of instances of previous strides). Intuitively, longer context length should imply more chance of correct prediction. But we find that some patterns repeat in short periods for short interval of time. To predict these patterns we must reduce the context length. But if we reduce our context length significantly to values like 16 and 8, we observe that mis-predictions increase drastically. This is because longer patters are captured partially, which when mis-predicted, decreases the IPC. So, we introduce the concept of dynamic context length to exploit this behavior.

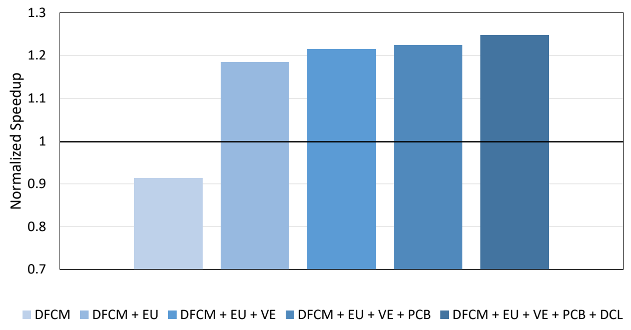

Figure 3: Normalized speedup with the enhancements on DFCM (early update, value estimator, PC blacklister, dynamic context length). Note that additional 4% gain is achieved by fine tuning the enhancements and the associated policies.

Implementation: In addition to an SHT with context length of 64 (SHT-64) in DFCM predictor, we have a SHT with context length of 32 (SHT-32) and a separate SPT for each SHTs. We empirically validate that this implementation works better than any other combination. To select the best context length (between 32 and 64), we use two saturating counters (one for SHT-64 and one for SHT-32) for each entry in SHT-64 and SHT-32 and store them in a table called preference counter table. We increment the preference counter for a particular entry on a correct prediction and decrement it on a mis-prediction. We simultaneously train for both context lengths to select the best context length based on the preference counter value (if > 0 then we choose the higher one).

Figure 4 shows the schematic of DFCM++, i.e. after adding all the enhancements to DFCM.

4. PARAMETER TUNING

4.1 Indexing into SPT tables

We index into the SPT table with the help of "hash of last N values". We define our hash function as follows: the values stored in the SHT are of width 64 bits. For each value, we extract the 64 bits, we fold it by 18 bits and then shift the existing hash by 1 bit and then do the XOR with the folded value and shifted hash. This is similar to the hash function named select-fold-shift-xor used in [8].

4.2 Dynamic confidence threshold in SPT

Threshold confidence is the confidence above which we start predicting values. It is a critical parameter to consider while designing DFCM++ as SPT uses it to decide whether to use the predicted value. This is very sensitive as it directly impacts the number of correct and incorrect predictions. We observed that certain instructions are “less predictable” and work better with high confidence threshold, while others lose precious opportunities to predict correctly with high confidence threshold. Hence we add a per PC dynamic confidence saturation that updates depending on the behavior of the instruction. The confidence threshold is incremented for instructions that cause a mis-prediction.

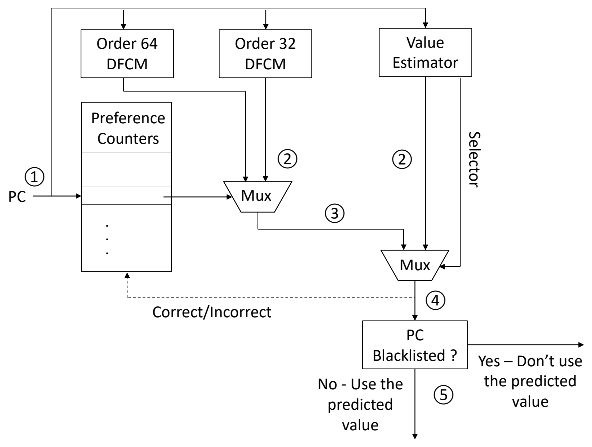

Figure 4: Schematic of DFCM++ including preference counter table and estimator.

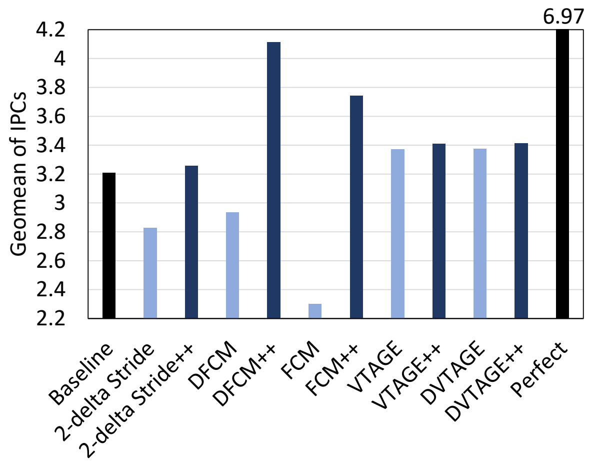

Figure 5: Comparison of geomean IPC for the existing predictors with and without our enhancements.

4.3 Blacklisting Threshold

As mentioned in the Section 3.3, we study the variation of IPC with DFCM++ with respect to the blacklist threshold. We tune this parameter and observe a performance peak at 10, meaning we blacklist a PC after 10 consecutive mispredictions.

5. PERFORMANCE EVALUATION

We use the CVP framework to quantify the effectiveness of our enhancements on DFCM. With our final design (Figure 4), we achieve a normalized speedup of 28.1% over baseline system that does not employ any value prediction, with a maximum performance improvement of 11X (an IPC jump from 0.1 to 1.1) for the trace compute_int_45.

We also apply our enhancements on other value predictors such as FCM, 2-delta-stride, VTAGE [6], and DVTAGE [7] and call it FCM++, 2-delta-stride++, VTAGE++, and DVTAGE++. Note that we implement an unlimited version of all these predictors. Figure 5 shows that our enhancements are effective across all the predictors and especially with 2-delta-stride, FCM , and DFCM based predictors.

Table 1 shows the hardware overhead of per entry for the structures used in DFCM++. Details of the field of this entries are available in the source file `mypredcitor.h`.

6. PRACTICAL IMPLEMENTATION

Questions regarding the complexity of the hardware design of the value estimator are legitimate. Initially, the predictor was designed keeping in mind the infinite track of the competition, but the ideas proposed can be revised suitably to get a practical implementation e.g. The complexity of the value predictor can be dealt by limiting the number of operations that could be executed by this unit. We limit this to only ADD, SUBTRACT and SHIFT operations. Hence introducing a basic ALU may solve the problem.

Since we weren't allowed to change any other files except the predictor's *.h* and *.cc* files, we were restricted to implement the design in the aforementioned manner. There might be other better and efficient ways of implementing the same.

7. CONCLUSION

This paper proposed DFCM++, a series of four enhancements on top of DFCM predictor that improves the effectiveness of DFCM. We show the effectiveness of each enhancement and the final combination that contains all the enhancements. We also show the effectiveness of our enhancements on other value predictors, such as FCM, 2-delta stride, VTAGE, and DVTAGE. On average, DFCM++ provides a geomean IPC (geomean over 135 traces) of 4.11 whereas a base DFCM provides 2.93 and the baseline system (without any value predictor) provides 3.21.

8. REFERENCES

[1] R. J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. *IBM J. Res. Dev.*, 37(4):547–564, July 1993.

[2] Freddy Gabbay and Freddy Gabbay. Speculative execution based on value prediction. Technical report, EE Department TR 1080, Technion - Israel Institue of Technology, 1996.

[3] Bart Goeman, Hans Vandierendonck, and Koenraad De Bosschere. Differential fcm: Increasing value prediction accuracy by improving table usage efficiency. In *HPCA*, pages 207–216. IEEE Computer Society, 2001.

[4] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 226–237, Washington, DC, USA, 1996. IEEE Computer Society.

[5] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 138–147, New York, NY, USA, 1996. ACM.

[6] Arthur Perais and André Seznec. Practical Data Value Speculation for Future High-end Processors. Research Report RR-8395, INRIA, November 2013. A fait l'objet d'une publication à "High Performance Computer Architecture (HPCA) 2014" Lien : http://people.irisa.fr/Arthur.Perais/data/HPCA

[7] Arthur Perais and André Seznec. Bebop: A cost effective predictor infrastructure for superscalar value prediction. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pages 13–25, 2015.

[8] Y. Sazeides and J. E. Smith. Implementations of context based value predictors. Technical report, University of Wisconsin at Madison, 1998.

[9] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *Proceedings of the 30th Annual ACM/IEEE International*

Symposium on Microarchitecture, MICRO 30, pages 248–258, Washington, DC, USA, 1997. IEEE Computer Society.

Acknowledgement

We would like extend our gratitude and thankfulness to Microsoft Research Labs, India and Research-I foundation for providing us with the financial support to attend and present the paper at the workshop, 1st Championship Value Prediction, held in conjunction with ISCA'18 at Los Angeles, USA.

Appendix

Component	**Hardware Tables**	**Hardware overhead per entry**
DFCM	SHT	460 bits
	SPT	68 bits
Value Estimator	PCIT	1112 bits
	Register entry (for each register)	193 bits
	Register file entry (for each inflight inst)	1586 bytes
	seq-no-to-pc-mapper	73 bits
	RFB-index-to-seq-no mapper	64 bits
PC Blacklister	Blacklisted PC table	64 bit

Table 1: Hardware overhead with DFCM++. Details of these hardware structures are available in the source file `mypredictor.h`