HW 1 Report (Shreyas Ravishankar)

Three cache replacement algorithms were simulated-

a) Belady's OPT

b) SRRIP- Static re-reference interval prediction

c)LRU- Least recently used

Configuration-

The replacement policies were meant for the LLC whose base configuration was 2MB with 2048 sets and 16 ways. Each cache line was 64B long. The other caches in the system used LRU. The system had a perceptron-based branch predictor, and prefetching was turned off. 100 million instructions were simulated from various benchmarks with no warmup instructions.

Expectation: Belady's OPT should be the most optimal solution (lowest MPKI) that is possible. SRRIP should perform better than LRU for scan workloads. LRU might perform better for cache friendly workloads. Neither should perform well for thrashing workloads.

Results: The first thing we notice is that Belady's OPT is sometimes performing worse as compared to SRRIP & LRU. Ideally the trace which we used to construct Belady's OPT should contain all LLC accesses, but it seems that there were some unseen accesses during execution. This may be the reason we are not able to perfectly simulate Belady's OPT.

There is a huge difference in ideal performance of sphinx, as compared to SRRIP & LRU, so there is a scope of improvement there.

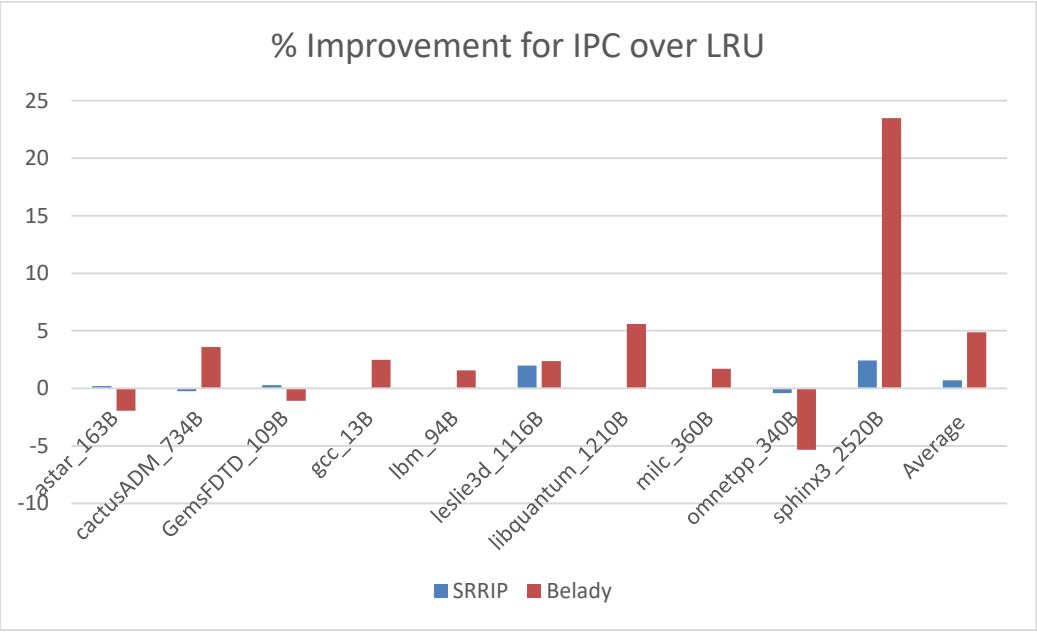


Figure 1: % Improvement in Performance compared to LRU (IPC)

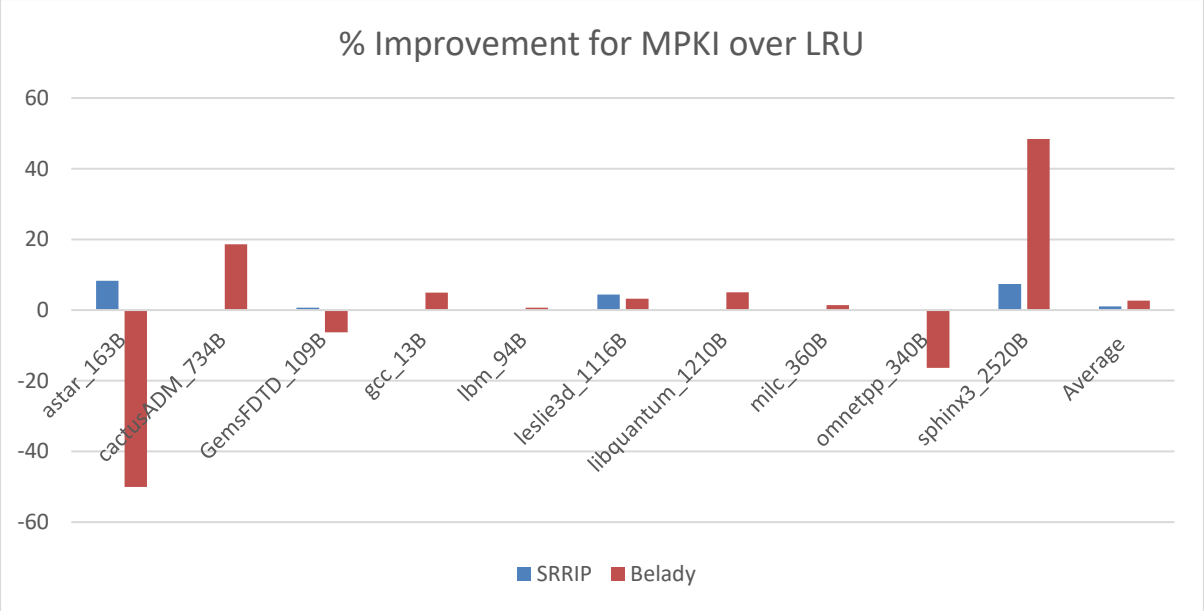


Figure 2: %improvement in MPKI compared to LRU

Variations of cache size, associativity and # of RRPV bits –

Various parameters like cache size, associativity and # of RRPV bits were sweeped across to understand the effectiveness of SRRIP.

Variation in cache size- We experimented with LLC cache sizes of 512KB, 1MB, 2MB (default), and 4MB. The associativity is kept the same (16), so the no. of sets are reduced to do these experiments.

Expectation: As we go for higher cache sizes, if there was a thrashing set, it might fit inside the cache now, which means LRU should be able to do better as size increases. This may not be true for all workloads, as the reasons for lower IPC might not be thrashing alone.

Results**:** It seems that most workloads don't benefit enough from the increasing cache size expect sphinx benchmark. So the knee of the working set (the point at which working set fits into the cache) seems to have reached b/w 2 and 4MB for sphinx.

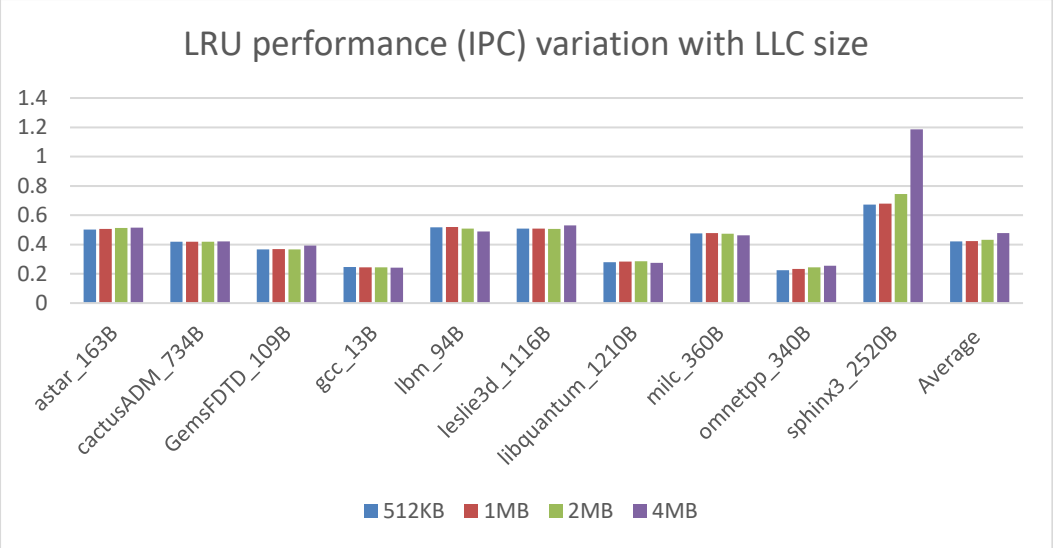


Figure 3: LRU performance variation with last level cache size.

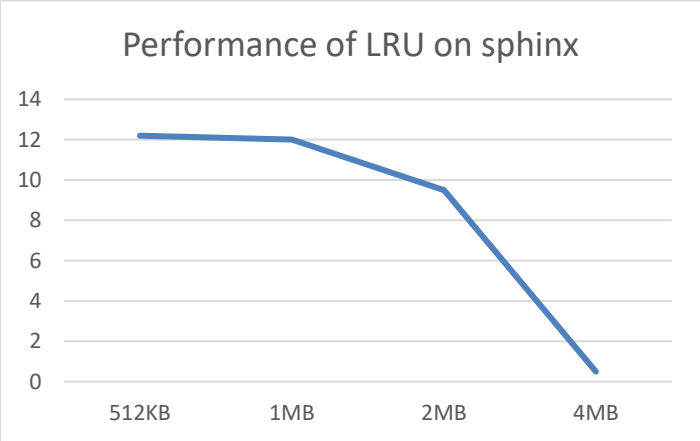


Figure 4: Performance of LRU cache replacement on the sphinx benchmark as cache size is varied.

As figure 4 points out, we can see the knee occurring at 2MB, when the MPKI reduces drastically from 9 to 0.5.

Coming to SRRIP: I expect it will be harder to get gains (as compared to LRU) as we go for larger cache sizes. Some workloads don't show improvement until we reach 4MB (type A), some show improvements but get worse as cache size increases (Type B), some workloads don't show any benefits of SRRIP at across all cache sizes (libquantum and milc).

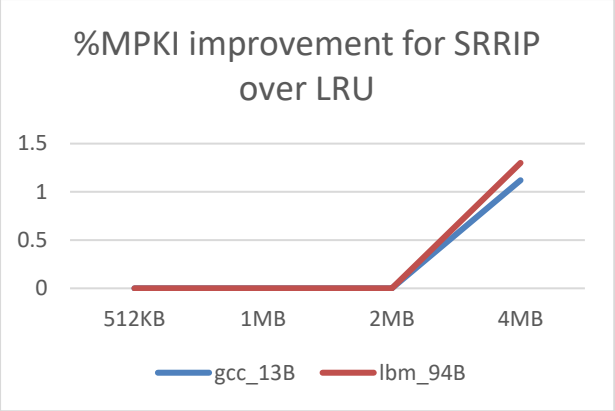


Figure 5: Type A, Show improvement only for larger sizes.

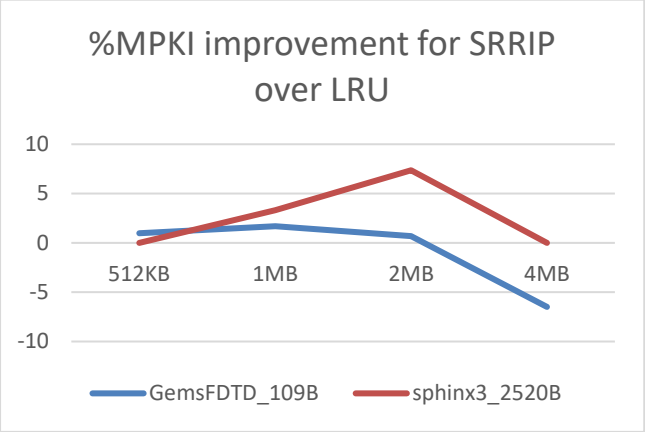
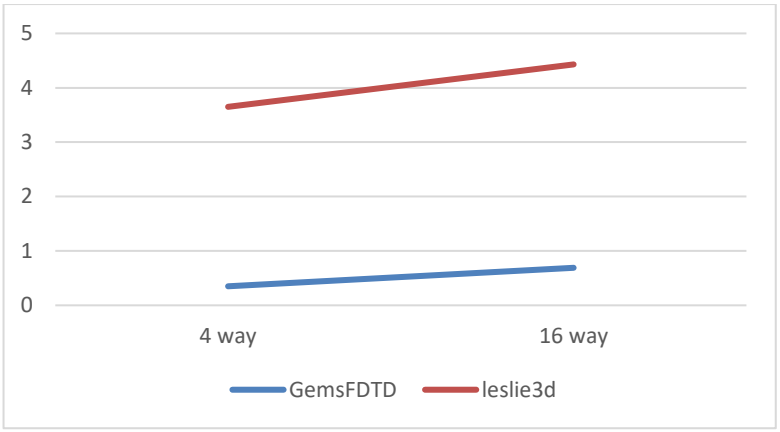


Figure 6: Type B, Show a mixed trend.

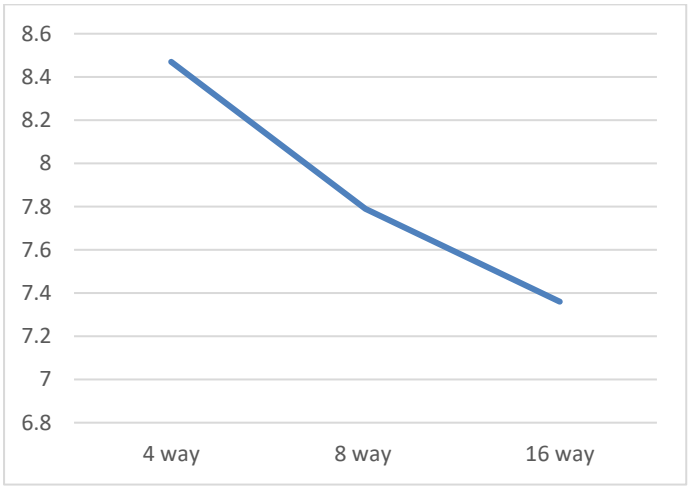
Variation of associativity- Associativity/ no. of ways for each set is varied. Experiments were carried out with 4 way, 8 way and 16 way. The size was kept constant at 2MB.

Expectation: As associativity is increased for most workloads there is less chance of conflict in a set associative cache, so the replacement decisions might play a lesser role. Hence LRU is expected to perform better for increased associativity. SRRIP also will have the same trend.

Results: Unfortunately, we don't see any trend by changing the associativity. Some workloads benefit from this (Type A). The workloads that don't benefit (Type B) might have access patterns such that they are accessing the same set repeatedly thus causing some sets to become full quickly. Again there are some workloads which are not sensitive to associativity and show zero benefits (libquantum, milc, gcc, and lbm).



Type A: workloads which benefit from increasing associativity (GemsFDTD and Leslie3D)



Type B: workloads which worsen from increasing associativity (Sphinx)

Figure 7: % improvement MPKI for SRRIP as size is varied

Variation of RRPV bits (For SRRIP only)- Changed the no. of RRPV bits that are stored in the cache. Experiments were performed with the default configuration (2MB, 2048 sets, 16 ways) while # of RRPV bits were 2, 3 and 4.

Expectation: According to the paper more no. of RRPV bits means more scan resistance, so longer scans be tolerated. So it should show a decreasing trend with respect to MPKI as we increase the no. of RRPV bits.

Result**:** We see this trend for some workloads, both interms of IPC and MPKI.

# of rrpv bits	2	3	4
gcc	0.243	0.244	0.246
lbm	0.509	0.511	0.511
MPKI			
	2	3	4
gcc	18.1	18	17.6
lbm	31	30.8	30.8

Figure 8: Variation of SRRIP with no. of RRPV bits.

HW2 – Prefetching

Shreyas Ravishankar

A markov table based prefetcher based on address correlation was implemented. Initially there was no restriction on table size. This was done to explore the upper bound of what is possible with longer histories so that the size factor does not affect our results. The prefetch degree was also variable to get maximum benefits from aggressive prefetching. The number of values that are prefetched are based on the current no. of values learnt for that address index. Ex- If the state of the Markov table->

Index Previous miss address

A C,D,E

B M,N

Then if we get a trigger A, it will prefetch 3 values, and if we get a trigger B, we will prefetch 2 values. This approach will henceforth be called **custom degree** prefetcher. The effectiveness of this approach was also confirmed from preliminary experiments to use this kind of prefetch degree. As we can see in **table 1**, custom degree performs better.

Prefetcher/Metric %	IPC improvement %	MPKI reduction %	Accuracy %	Coverage %
Custom degree (described above)	18.52	20.61	85.83	51.63
Degree=1	6.2	7.33	40.14	14.97

Table 1. Preliminary experiments on the astar_313B benchmark to choose degree for further experiments.

We also consider different triggers (**table 2**)- only miss addresses and (miss and hit) addresses. The former leads to less aggressive prefetching, and lesser cache pollution. Ex- The accuracy and coverage is comparable for both but the no. of prefetches is 6.8e6 (miss triggered) as opposed to 9.5e6 (hit &miss triggered), so the latter leads to more cache pollution and lower IPC improvement.

Prefetcher/Metric %	IPC improvement %	MPKI reduction %	Accuracy %	Coverage %
Miss address trigger	18.52	20.61	85.83	51.63
All address trigger	13.98	20.09	84.97	53.54

Table 2. Preliminary experiments comparing triggers. The prefetcher with custom degree, having miss address and all addresses (hits and misses) as triggers respectively. The benchmark used was astar_313B benchmark.

Experiments with Increasing History

With an initial degree as 1 for the markov prefetcher, the no. of trigger addresses was varied across 1,2,3 and 4.

Expectation- The prefetcher might be able to detect more complicated address patterns. So we expect the accuracy to increase resulting in IPC improvement.

Results-

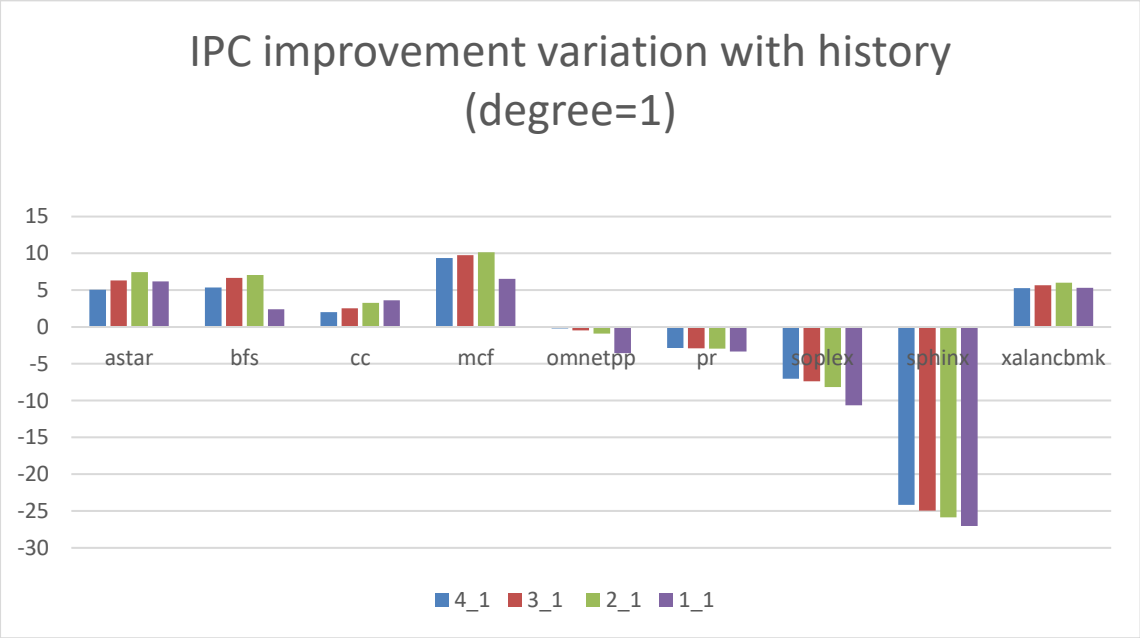


Figure 1. IPC improvement for prefetcher with degree 1 as number of trigger addresses (history) in increased from 1 to 4. (Notation- 4_1 means 4 trigger addresses with a degree 1)

For the benchmarks with positive improvement compared to baseline, IPC improvement reduces with increasing history. These results are understandable since

a)Each prefetch index takes more time to train

b)Its hard for the addresses to match the longer address pattern which might lead to more markov table misses for longer histories.

Therefore there is **coverage loss (figure 2)** due to the misses that could be prevented using a smaller history.

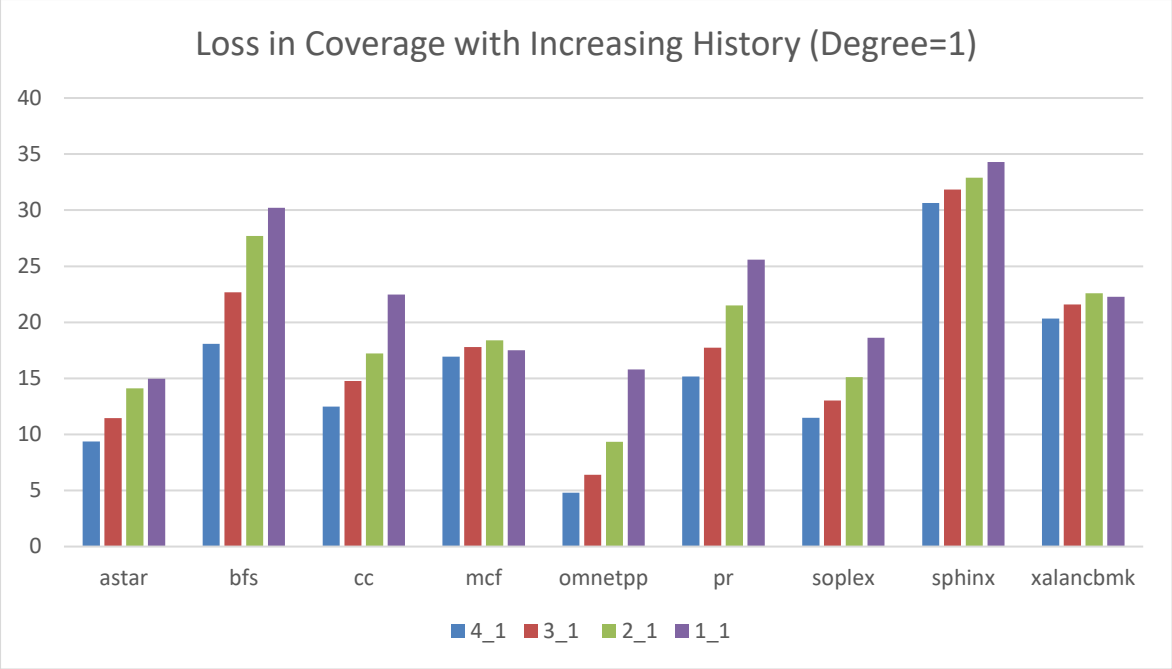


Figure 2. Coverage loss as we go for longer histories for a degree 1 prefetcher. (Notation- 3_1 means 3 trigger addresses with a degree 1).

From table 3, The accuracy does not show a strong increasing trend although it increases initially for shorter histories (1→ 2 addresses). In conclusion, the overall effect is that accuracy changes negligibly while the coverage reduces significantly as we increase history leading to an overall reduction in %IPC improvement.

	4_1	3_1	2_1	1_1
Avg Accuracy (%)	34.71	35.76	36.84	34.43
Avg Coverage (%)	15.47	17.47	19.88	22.42

Table 3. Average accuracy and coverage as history length is varied from 1 to 4. (Notation- 4_1 means a history length of 4 and a prefetch degree of 1)

Can we improve performance with longer history? - Increasing the prefetch degree and by using hybrid prefetch approaches.

One way we can improve the effectiveness of longer history by **increasing the prefetch degree.** For a particular history length, we expect a greater IPC reduction if we increase the degree (as seen in Table 1). So, we now perform our experiments with custom degree.

Result- **Figure 3** shows our results on all the benchmarks. Many benchmarks like mcf, omnetpp, pr show a huge loss in % IPC improvement especially at lower history lengths.

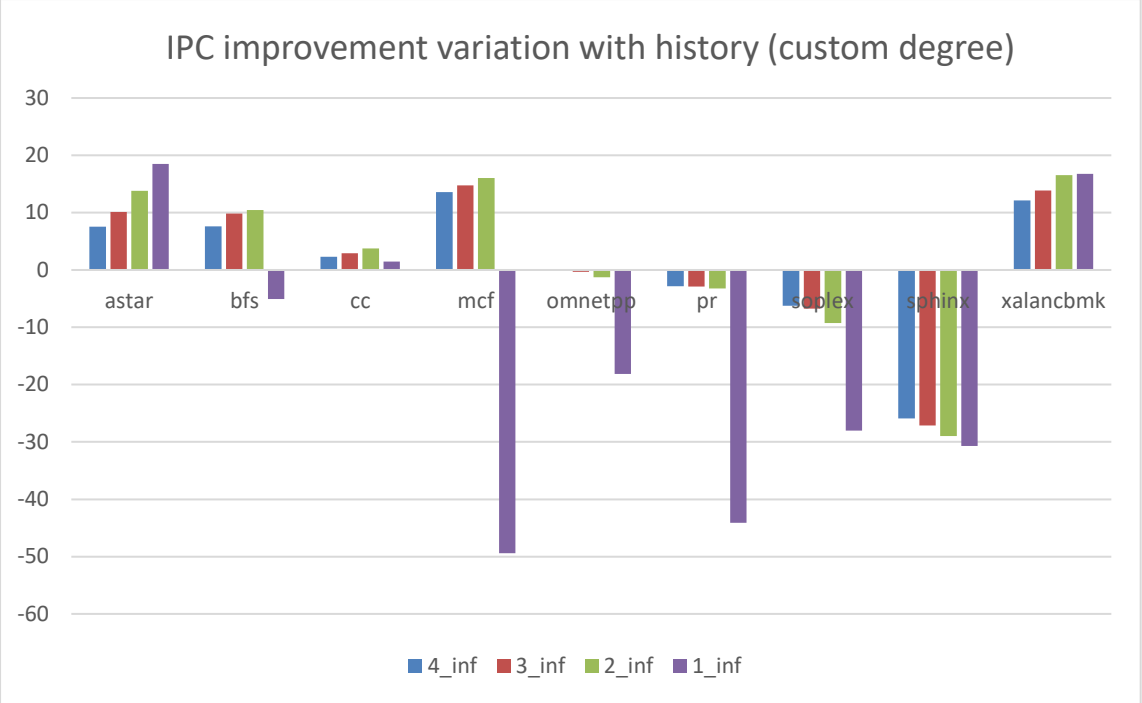


Figure 3. %IPC improvement variation with history. This experiment was done with a custom degree. (Notation- 4_inf means a history length of 4 for trigger, and inf means a custom degree).

Possible explanation- Since we used custom degree, we do not know at each instant how many prefetches were issued. When we took the average degree of prefetch (**Figure 4**) issued by the prefetcher; We observed that for lower history length, the width of the table (prefetch degree) blew up, which led to a lot of erroneous prefetches. **This explains why some benchmarks (mcf, omnetpp, pr) have a large reduction in %IPC improvement compared to baseline.** So it is necessary to control the prefetch degree to a smaller value especially at lower history lengths so as to reduce the negative effect of the aggressive prefetcher. This will be done in the next section when we are moving towards a practical prefetcher.

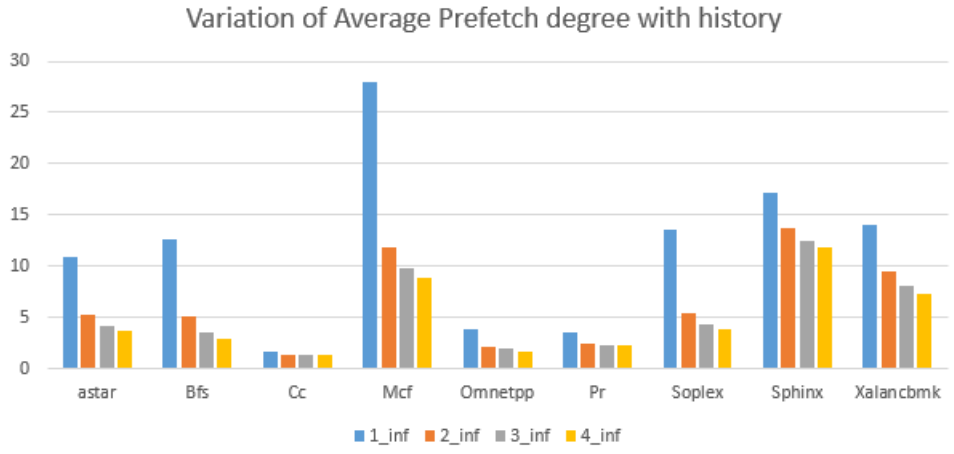


Figure 4. Actual degree of prefetch(average) that is taking place for each of the benchmarks when we use a custom degree.

By increasing the degree, we were not able to remove the decreasing trend in %IPC improvement with increasing history.

But if we compare the results with degree 1 prefetch (**Figure 5**). For benchmarks with positive %IPC improvement, increasing degree lead to an absolute increase in %IPC improvement. For benchmarks which got affected negatively with degree 1, using a custom degree reduced the negative effect of this loss (**a net positive improvement for %IPC improvement for most benchmarks**).

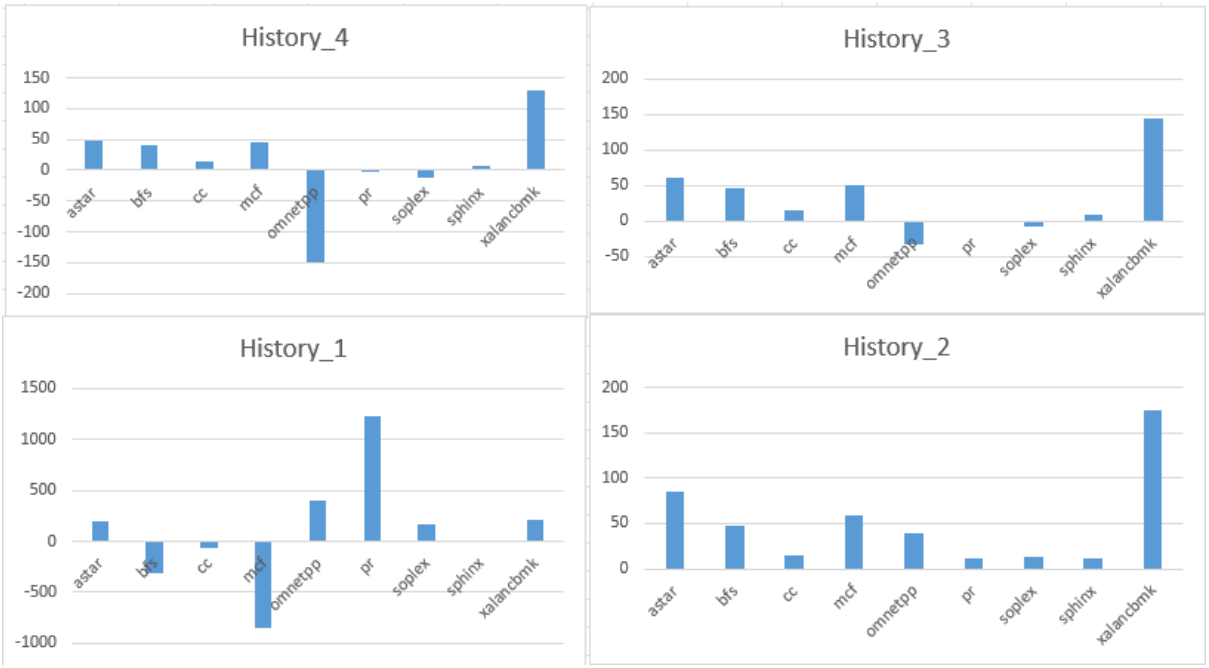


Figure 5. Change in %IPC improvement from degree=1 to custom degree for various history lengths.

Effect of using a hybrid prefetcher-

To account for the coverage loss (because of increased training time for longer histories), we try doing a simple next block/next line prefetching if the markov table is not able to make a prediction at that instant. The initial results on astar_313B benchmark looked promising, it showed a >1% improvement in IPC reduction.

Result- As seen from **figure 6 and figure 7,** we are now able to reap the benefits of increased history with benchmarks like bfs and cc. Benchmarks like mcf, omnetpp are dominated by high degree at history length of 1, so it is not able to see the benefits of the next line prefetcher.

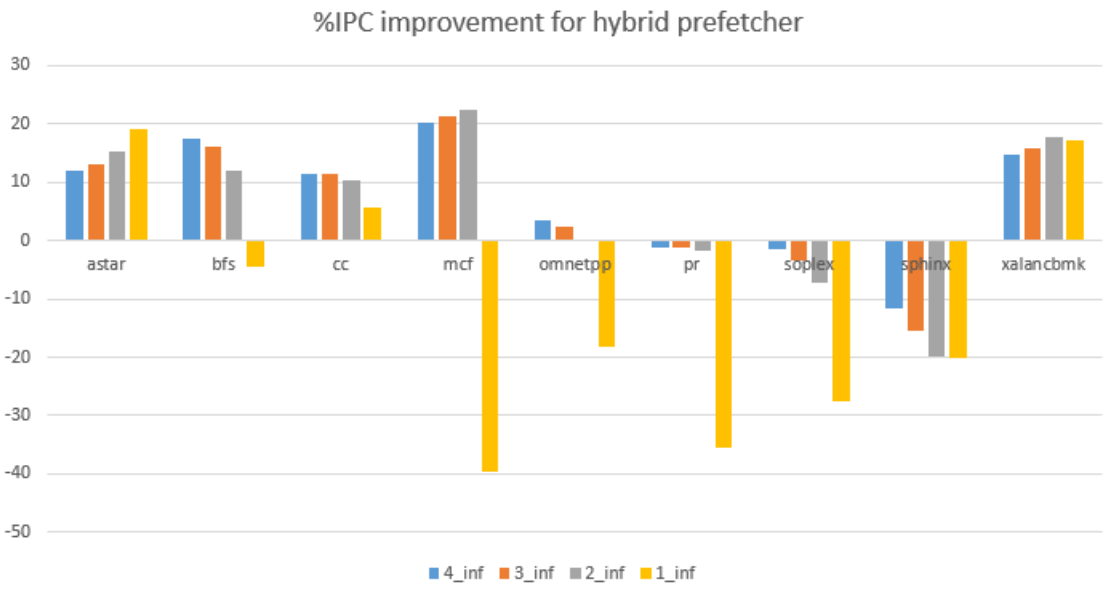


Figure 6. %IPC improvement for varying histories of a hybrid prefetcher. The prefetcher uses next line prefetching if it is not able to find the address in the table. (Notation- 4_inf means a history of 4 addresses with a custom degree).

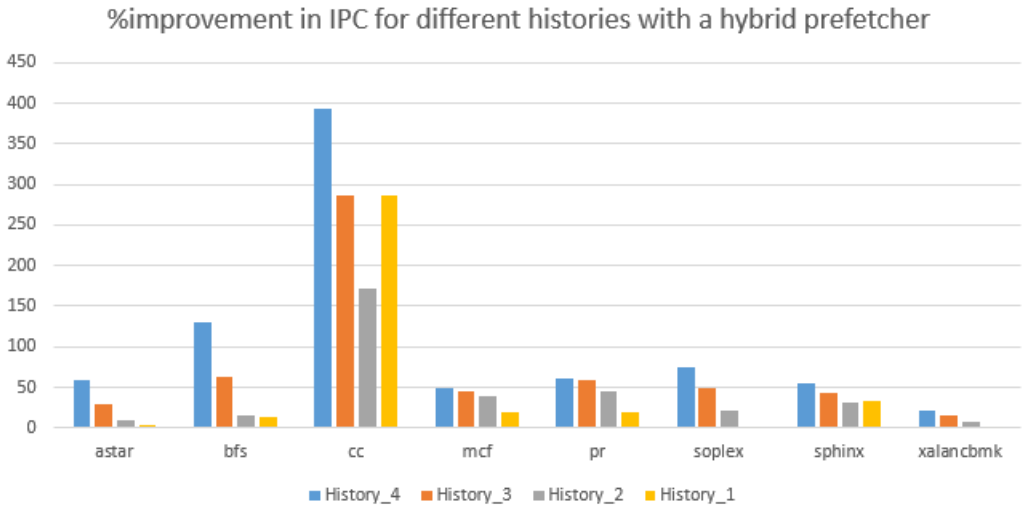


Figure 7. All benchmarks show an increase in %IPC improvement when we go from a markov prefetcher to a hybrid prefetcher. (Notation- History_4 refers to history length of 4).

Figure 7 shows how %IPC improvement changes when we go from a simple markov prefetcher to a hybrid prefetcher using a next line technique. All benchmarks across history lengths show an improvement in the %IPC improvement as observed.

What are the tradeoffs of using a longer history?

The size complexity will be equal to (number of rows in markov table)*(address length)+ (number of rows*number of columns*address length). The 1st factor is for the index storage

and the 2nd factor is for the prefetch address storage. The no. of columns should be equal to the maximum degree of prefetches being performed.

The time complexity of the prefetcher can be approximated as follows-

a) Content addressable search of the tag (O(size of table))
b) Finding the addresses to prefetch in case of higher degree (O(degree of prefetcher))

The prefetch latency will be proportional to size of markov table*degree of prefetcher.

Practical prefetcher (How can we reduce the negative aspects of using a longer history?)

Now that we have seen ideal results with no constraints, we now need to build a practical prefetcher. The markov table size (number of rows) will be decided based on the **sizes** (**Figure 8**) **and average degree of prefetch (Figure 4, table 4**) of actual benchmarks in the unconstrained experiments performed until now. These will be used provided they are reasonable for an on-chip implementation.

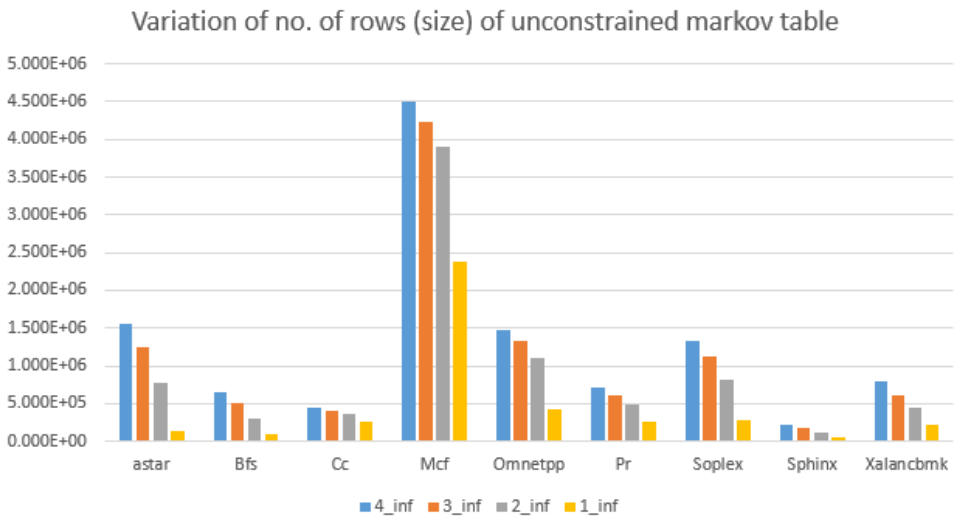


Figure 8. The average number of rows of the markov table (size) needed for the unconstrained prefetcher

History length	1_inf	2_inf	3_inf	4_inf
Average degree	11.67	6.3	5.3	4.85

Table 4. Average degree of prefetcher (actual degree used on average) derived from figure 4 for the custom degree prefetcher. (Notation- 4_inf means a history of 4 addresses with a custom degree).

For degree we choose the ideal degree to be **6** since we know that having a very high degree especially for lower histories in harmful for IPC. Six was chosen so that it could accommodate required degree for history lengths of 2,3 and 4. We don't consider 11.67 (history length=1) as we know it is detrimental (**Figure 3**)

For the total size, we won't be able to meet the required sizes of any prefetcher. They are all too big for on-chip storage. We consider **128KB** to be the size of our prefetcher based on typical L2 sizes (512KB). So no. of rows in our table- 128KB/(degree*64 bit address)= 128*1024/(64*6) = 341 rows. Further if we have a size constraint, we must have a replacement policy, so we implement an LRU replacement policy for the practical prefetcher.

So finally we chose a **markov table size (rows)= 341**, and **degree= 6** and an **LRU** replacement policy. We also had a next line prefetcher based on our hybrid prefetcher results.

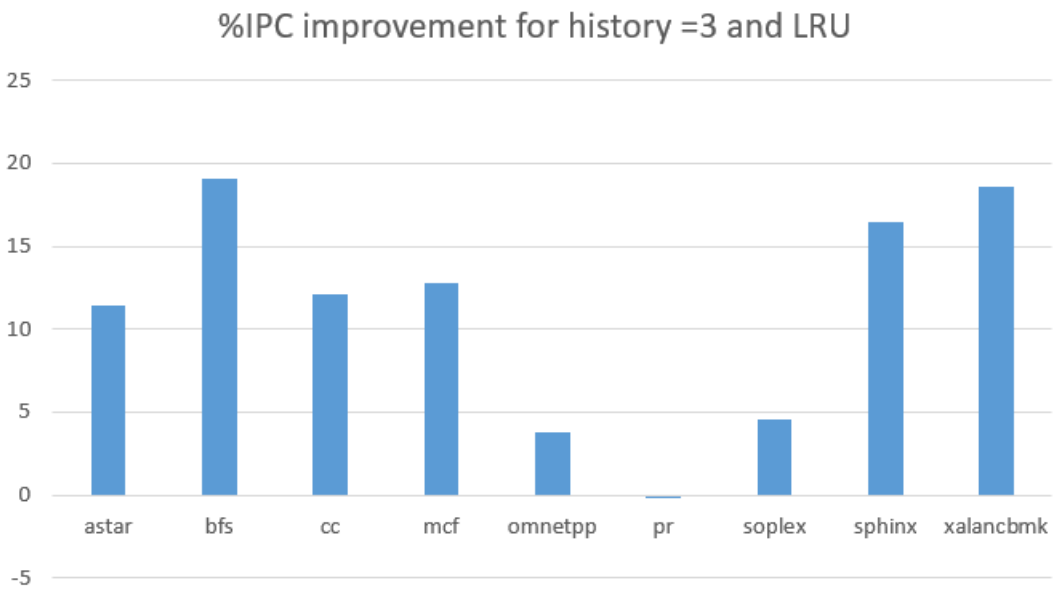


Figure 9. %IPC improvement for LRU based prefetcher with prefetch degree=6 and history=3.

As observed in **figure 9**, we are able to get good %IPC improvements, and able to eliminate the negative effects of higher degree for certain benchmarks.

Conclusion

We explored the effect of longer history on address correlation based markov prefetcher. We were able to improve on our results by using a higher (but constrained) prefetch degree and by including a next line prefetcher when markov table fails to predict a valid prefetch. Finally we were able to get a practical prefetcher which employs all these benefits to get good results across all the benchmarks

An analysis of the efficacy of value prediction

Shreyas Ravishankar SR48925

Introduction

Modern processors have managed to solve most bottlenecks regarding performance. This includes **branch prediction**- which ensures that branches don't cause delay, **prefetching**- to take care of compulsory misses, **caches** to enable faster on-chip memory access, dynamic dataflow to execute out of order while still taking care of data dependencies. The only issue now remains is we are limited by the no. of RAW dependencies in the program. It was known to be a fundamental limit until value prediction was introduced and developed in [1] [2],[3].

Value prediction basically means predicting the entire value of the register. It is based on value locality, which means values tend to repeat over the course of the program just like the nature of branch decisions.

Previous work

There has been a lot of work in this area in the academic community. This section will briefly describes the various efforts that have been done in the area of value prediction.

[3] introduced the idea of value locality and its applicability to load values. Instruction address loads seem to have better value locality than data loads. They also make several observations of why inherently value locality exists, due to the presence of data redundancy, virtual function calls, error checking to name a few. They exploit the value locality using a load value prediction (LVP) mechanism which consists of a PC localized counter, load value prediction table to store prediction history and a verification unit to verify the predictions.

[1] extends the above idea to not only load values but all kinds of integer and floating point registers. They show impressive performance improvements with value prediction.

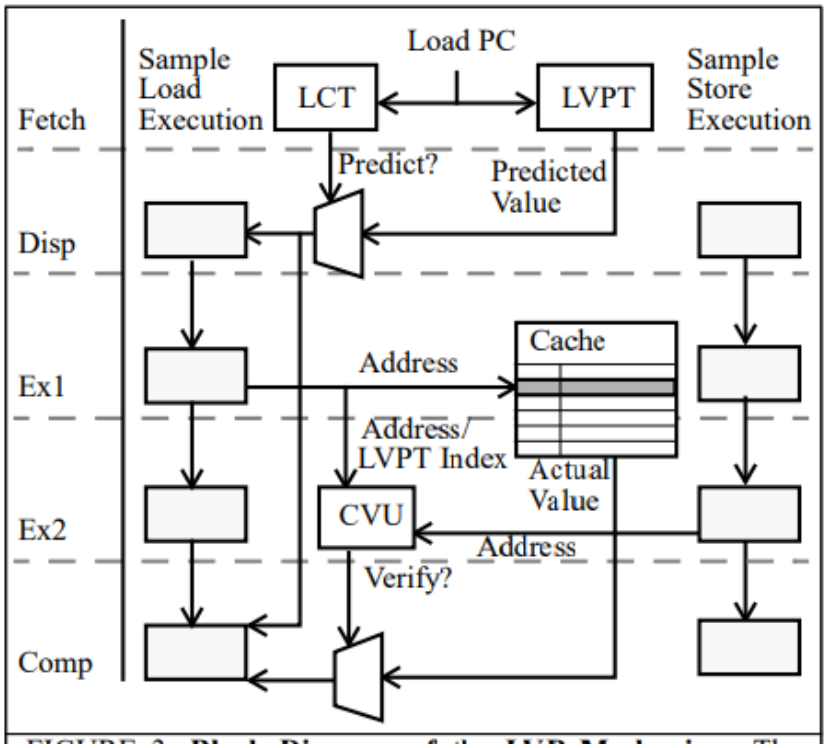


Figure 1. The load value predictor as given in [3]. The LCT is like a counter for maintaining confidence. The LVPT is used for making predictions, and the LCT is a counter for maintaining confidence for those predictors.

[2] Was also a similar line of work to the above, to prove the efficacy of value prediction. It showed various similarities between text compression and branch prediction to value prediction. It introduced the concept of **computation** and **context based prediction**. The former is based on performing calculations on previous values like calculating stride or using the last value as the prediction. The latter uses history to identify occurrences of a context-value pair. These are typically more accurate than computation based predictors. [8] expands on various practical aspects when implementing two level context based value predictors. They talk about factors such as context functions, combine functions, value history table and value predictor size etc. It focuses on the implementation compared to earlier theoretical work by the same authors [2].

[9] introduced VTAGE, a value predictor based on the branch predictor ITTAGE. It is based on global branch history and path history. It betters FCM based predictors and can be used with stride based predictors. They are able to make prediction on tight loops where back to back prediction is required. They also introduce the concept of forward

probabilistic counters to mimic the presence of large counters using smaller counter widths.
[10] introduced the EVES predictor which improves both the VTAGE and the stride predictor to give a more powerful predictor. Here they make use of the last committed value of an instruction, and the no. of inflight occurrences of that instruction to compute the effective stride. They also propose a speculative update mechanism.

Finally [11] introduces HCVP, a better way of combining branch history and value history to make value predictions. They also posit that predicting values for load address producing instructions and high fanout instructions are highly beneficial for performance.

Aim of this work

1) Prove that value locality exists in current workloads
2) The simulator used does not have the basic predictors such as **FCM**, **stride, last value** predictors. So firstly we try to get those predictors to work with various configurations.
3) Fine tune the metric for speedup calculations.
4) Analyze the different kind of predictable instructions in workloads.
5) Propose a hybrid predictor based on the above instruction type analysis.

Does locality exist?

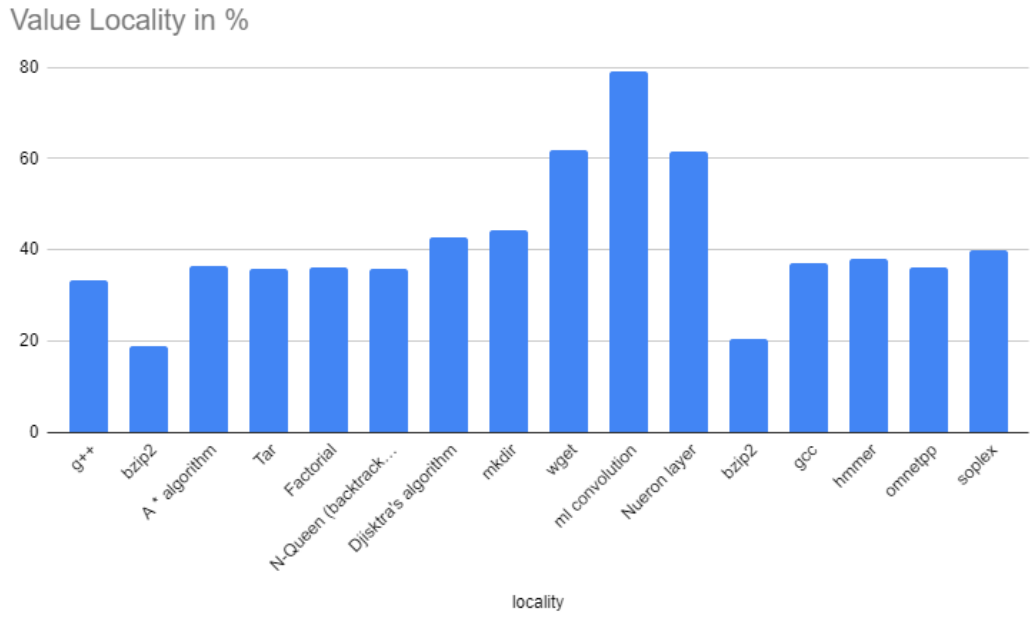


Figure 2. Value locality with a value history of 1.

The figure shows for different workloads, how many values are such that they are equal to their previous occurrence. As we can see we have sufficient value locality to exploit/predict in most workloads.

Detailed view of predictors implemented-

The Finite Context Method Predictor (FCM)

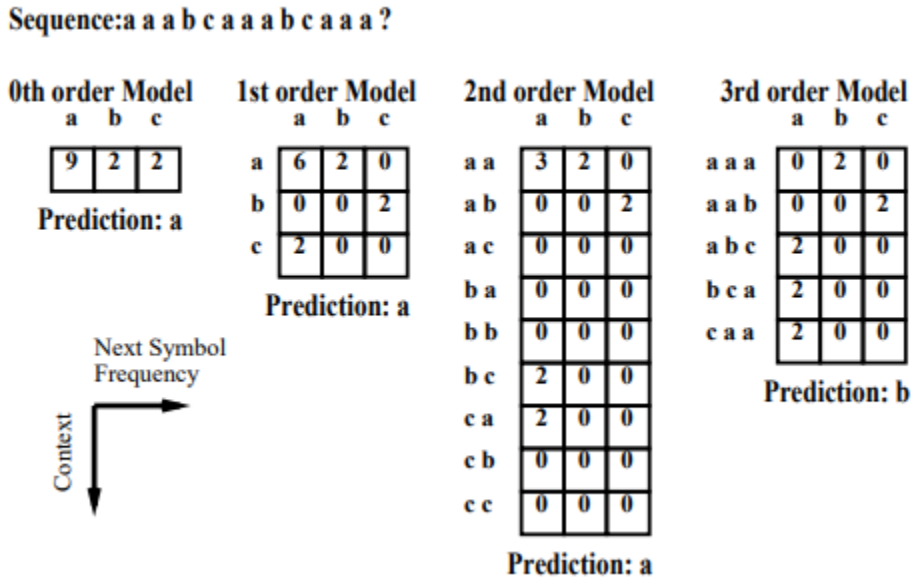


Figure 3. The nth order FCM predictor.

As described in [2] and figure 3. the finite context method is a multi level prediction scheme that uses value history context to make value predictions. Ex- A 1st order model uses a single value history context. Each row has a set of possible values with counts seen in that context. The value whose counter has the highest value is chosen to be the prediction. Our implementation uses a PC localized predictor (figure 4). The counter threshold decides how long it takes for the predictor to gain confidence

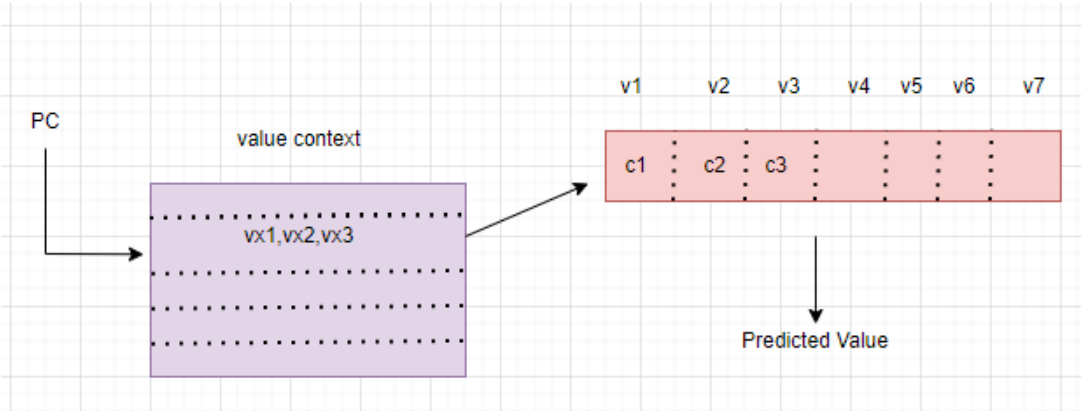


Figure 4. A PC localized FCM predictor.

The vx1,vx2,vx3 correspond to a 3 value context. Each candidate value (v1,v2,v3) is associated with a corresponding counter (c1,c2,c3) etc.

As will be described later we evaluate a 1st order model and a 4th order model for FCM. We also experiment with different counter thresholds.

The Stride Predictor

The stride predictor is similar to the FCM, given a context, strides are stored instead of values. The predicted value is the previous-value + stride with the maximum count. This is shown in figure 5.

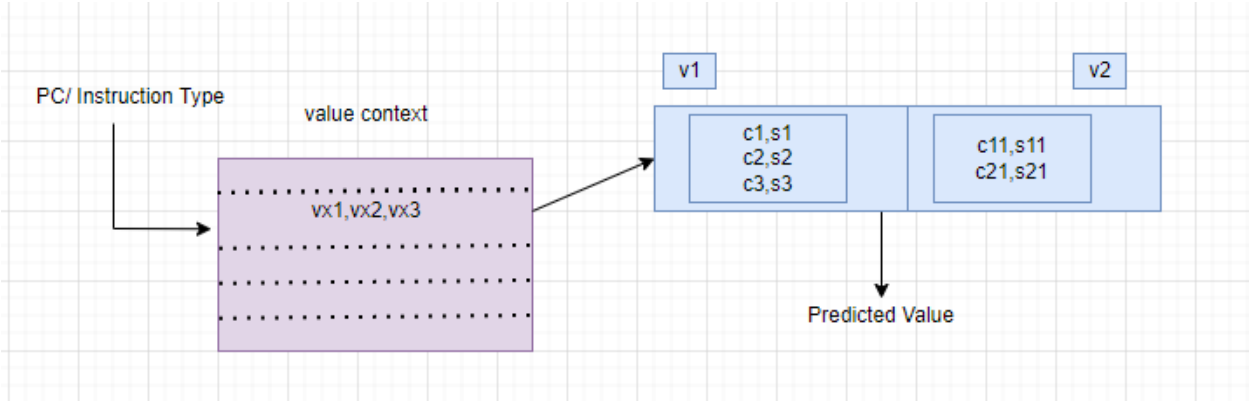


Figure 5. A PC localized stride predictor

As one can observe, each value is associated with a set of strides (s1,s2,s3) and corresponding counts (c1,c2,c3). This is because different strides can occur for the same value. The final prediction is given by stride(c max) +vx. (i.e the stride with maximum count value is chosen and added with the corresponding value. For the value context, a single order(single value) is used.

As described later, we can either localize this stride based on instruction type or by PC.

Last Value Predictor

This predicts the last value for the value prediction. We implement two variations: PC localized and type localized.

In short, the different configurations of predictors that were implemented were-

- FCM
    - PC localized single history/order- (**FCM_PC**)
    - Counter threshold (8)- (**FCM_PC_thres_8**)
    - Longer history trigger (4) - (**FCM_history_4**)

- Stride based predictors
    - Instruction Type localized- (**Stride_type**)
    - PC localized- (**Stride_PC**)
- Last value predictor
    - Instruction Type localized- (**Last_value_type**)
    - PC localized - (**Last_value_PC**)
- Value prediction unit (Reproducing the baseline) - (**VPT+CT**)

All our predictors are considered to be in the unlimited budget category. The names given in brackets will be the names used to represent the graphical results.

Tools

Intel Pin tool - A dynamic binary instrumentation tool used to analyze the code at runtime"Callbacks" are registered on instructions or basic blocks, which basically can extract and process information from the instructions.It can act as input (**can generate traces**) to many tools like **Champsim**, **Intel Vtune profiler** etc. Ex- We can extract instruction counts, memory accesses, branches. We can build processor models and other predictors using Pin.

Benchmarks

The benchmarks used were5 SPEC benchmarks - bzip, gcc, hmmer,omnetpp, soplex. In addition the predictor was evaluated on 11 non-SPEC benchmarks- g++, bzip, A*, Tar, Factorial, N-Queen, Djikshtra's algorithm, wget, mkdir, convolution and neuron layers. The last two workloads were taken from the shark ml library.

Results

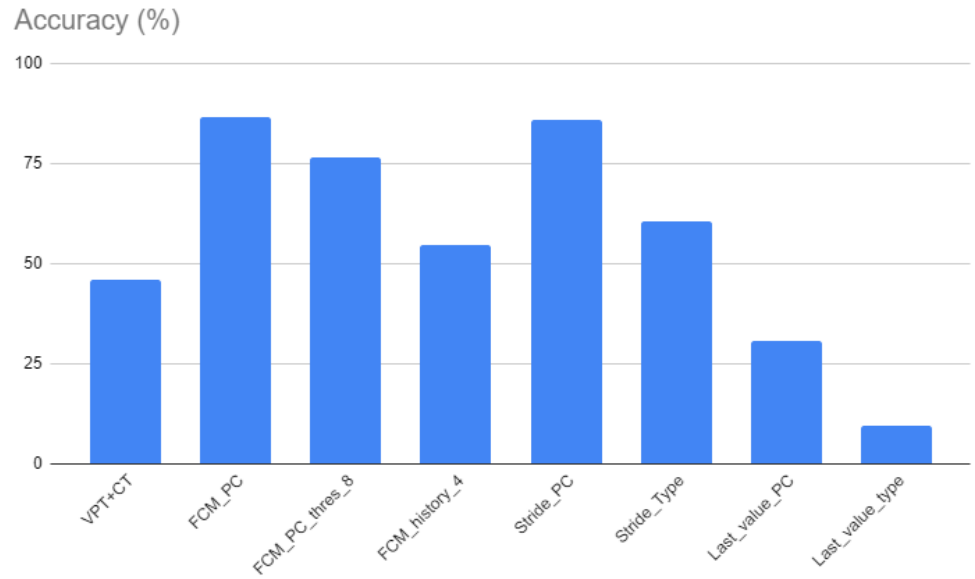


Figure 6. Average accuracy of predictors across benchmarks for different predictors.

As we can see in the figure the FCM_PC (86.78%) and Stride_PC (86.07%) have the highest prediction accuracies. Going from a counter value of 1->8 should have ideally increased accuracy. Similarly longer history from 1->4 should also have increased accuracy, But both of these show negative trends. This might be because both are losing out on important predictions because they take time to learn. Secondly, the longer contexts don't seem to be repetitive in the workloads chosen. There might also be aliasing effects introduced due to the hash function chosen to combine the various histories.

The type localized predictor (Stride_Type) and (Last_value_type) don't give good accuracies. Besides, even if they gave good accuracies, they would be practically infeasible because the table would consist of very long rows (because the no. of instruction categories are few compared to PC), which would lead to longer search times in a practical setting.

The previous set of results show that the simple PC localized FCM predictor (FCM_PC) and PC localized stride predictor (Stride_PC), so now we show the performance of the stride predictor across all workloads.

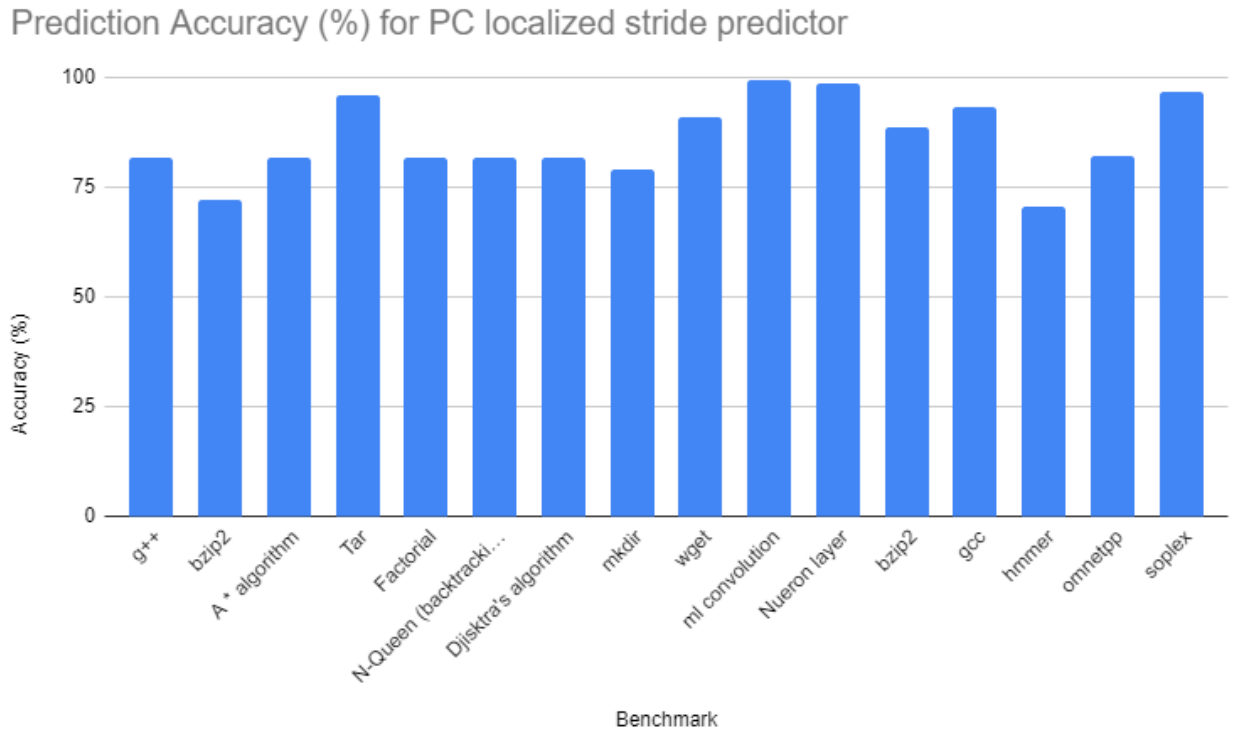


Figure 7. PC localized stride predictor accuracies for various workloads.

As we can see, the predictor is able to predict with more than 75% on most workloads except bzip2 and hmmer. It even touches 99% in the two ml workloads. This shows that these two workloads are highly amenable to value prediction.

Speedup

The simulator does not have an accurate way to measure speedup, so come up with a metric-

$\Sigma * (CPI\ of\ each\ type * \#\ of\ instructions\ of\ each\ type)/$

($\Sigma * (CPI\ of\ each\ type * \#\ of\ instructions\ of\ each\ type)$ +(Miss penalty *# of incorrect predictions)-(Stall cycles *#of correct predictions))

The latency of various types of instructions was taken from [6] and [7] for the Intel pentium processors. The stall cycles and the miss penalty were assumed to be 50% of the instruction latency. This is likely an underestimate for both the quantities.

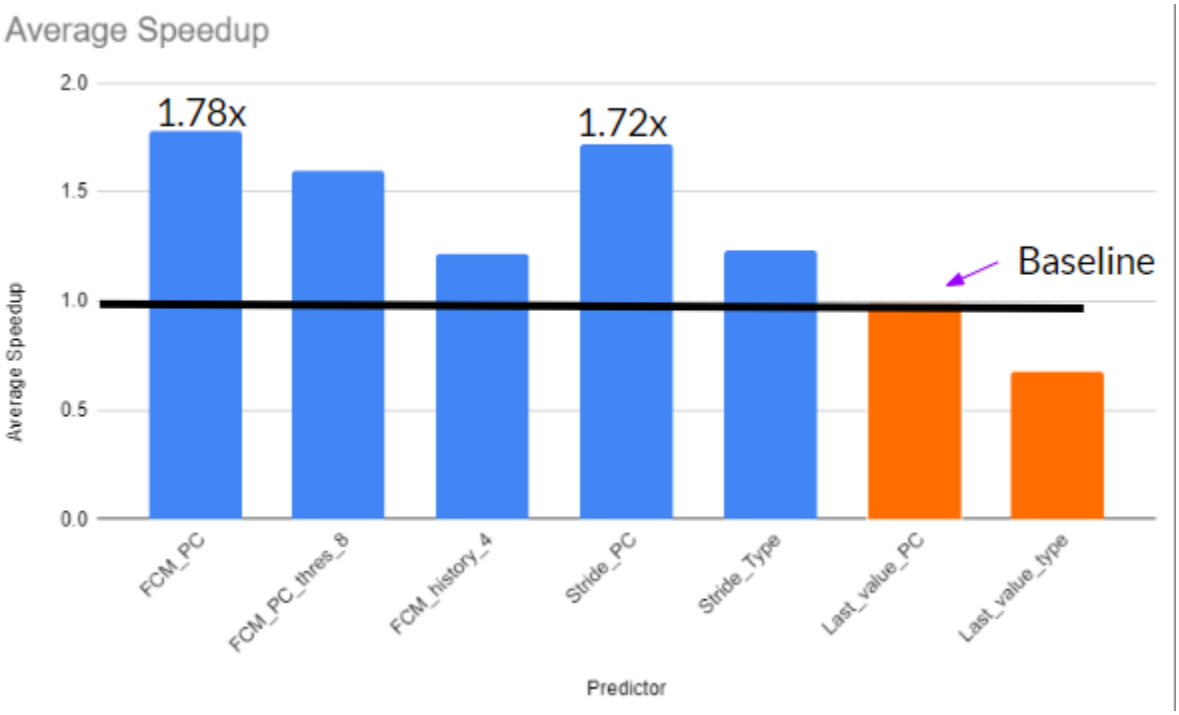


Figure 8. Average speedup obtained over all workloads for the different predictors.

As the figure shows, we get 1.72x and 1.78 speedups from the PC localized stride and FCM predictors respectively. These 2 predictors showed high accuracy and it is not surprising that they show commensurate improvement in speedup. The last value predictors (shown in orange), don't show any benefit, and actually perform worse compared to without a predictor.

Instruction type analysis

Now we want to analyze the different kinds of predictable instructions and see which ones are more predictable. We also want to come up with a hybrid predictor that does well on the important classes of instructions.
The instructions were classified into multiple types on which predictions were made-

Instruction category	Meaning
I_PURE_LOAD	Load int value into reg: **Mov reg,[memory_loc]**
I_LOAD_ARITH	Load int value and do arithmetic instruction. **add reg,[memory_loc]**
I_ARITH_1OP	Int Arithmetic with 1 operand; **add reg,immediate**
I_ARITH_2OP	Int Arithmetic with 2 operands; **add reg,reg**
I_REG_MOV	Move int reg to int reg; **mov reg,reg**
F_PURE_LOAD	Load fp value into reg: **mov reg,[memory_loc]**
F_LOAD_ARITH	Load fp value and do arithmetic **add reg,[memory_loc]**
F_PURE_ARITH	Fp arithmetic padded instruction (SIMD)
F_REG_MOVE	Fp reg **movd reg,reg**

Now each workload was analyzed to understand what percentage of the workloads are made up on the aforementioned classes of instructions-

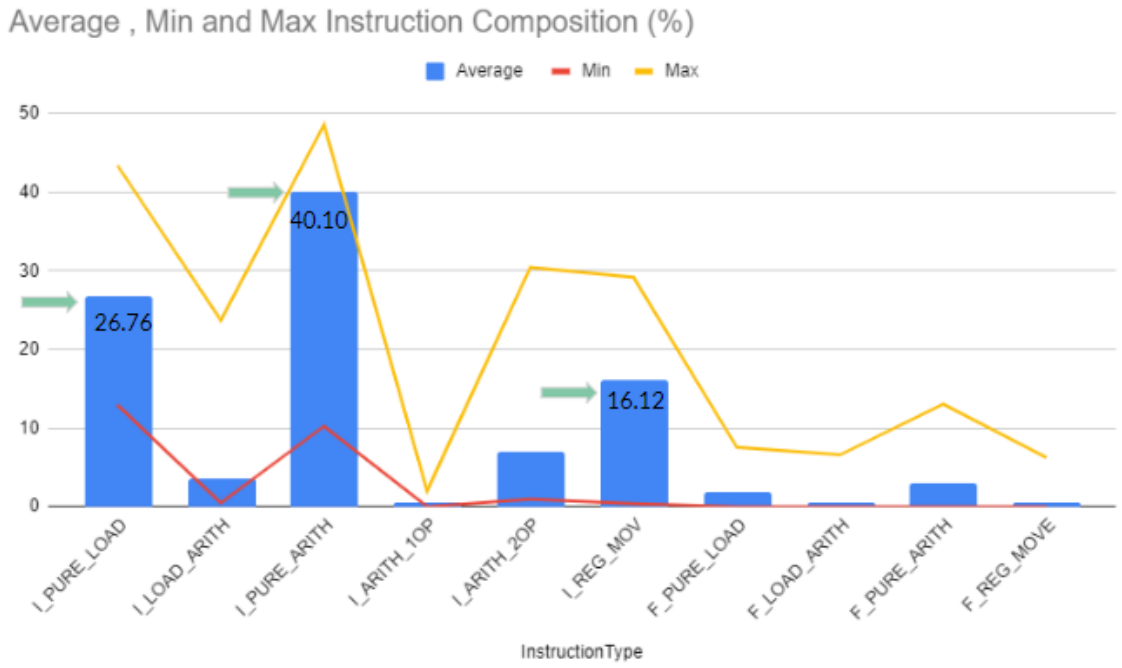


Figure 9. Compositions of instruction type across all workloads

As we can see in figure 9, the workloads evaluated are dominated by **integer loads/<I_PURE_LOADS> (26.76%), integer arithmetic/I_PURE_ARITH (40.01%) and integer register moves/I_REG_MOV (16.12%)**. We might also want to consider integer load arithmetic/I_LOAD_ARITH and 2 operand arithmetic instructions/I_ARITH_2OP as they seem to show a high degree of variance, so they might become dominant in some workloads.
It makes sense for us to choose predictors that do well on these kinds of instructions. This is because such dominant constituents of the workload will have the most impact on IPC if predicted correctly.

Now that we have the dominant instruction types, we see how the accuracy varies for these type of instructions across workloads?

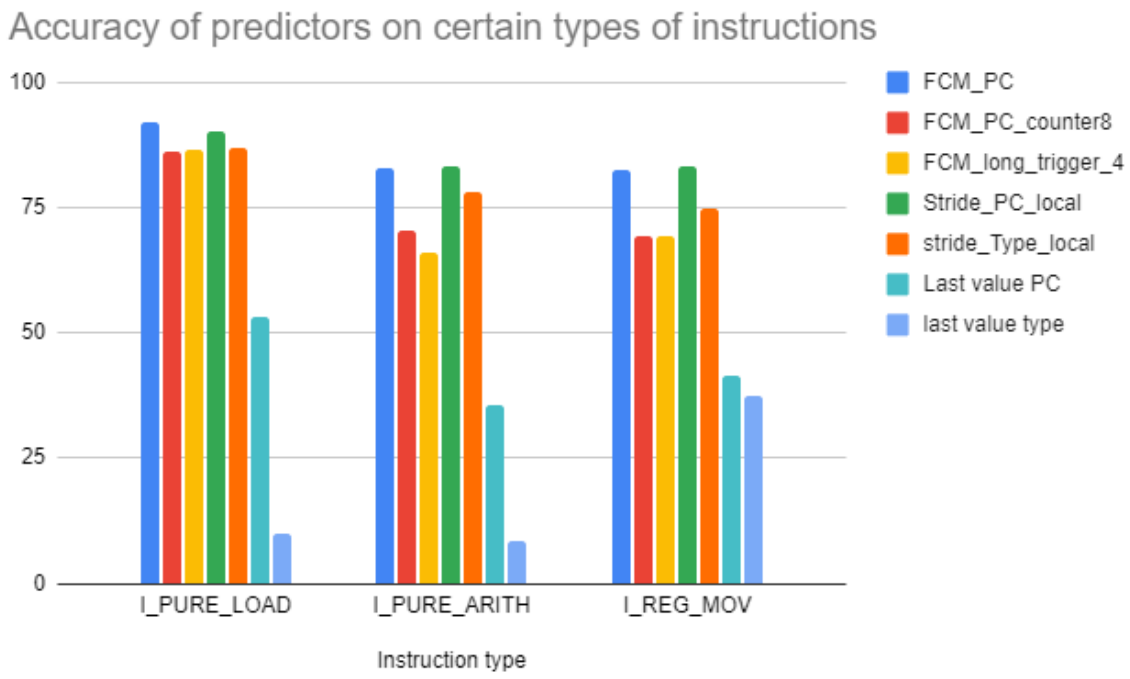


Figure 10. Performance of the various predictors on the three dominant instruction types.

To no surprise the PC localized FCM and PC localized stride do the best. Type localized stride also does quite well, but may not be practical from an implementation point of view. Now for the best performing predictor, we see how its accuracy varies across ins. types-

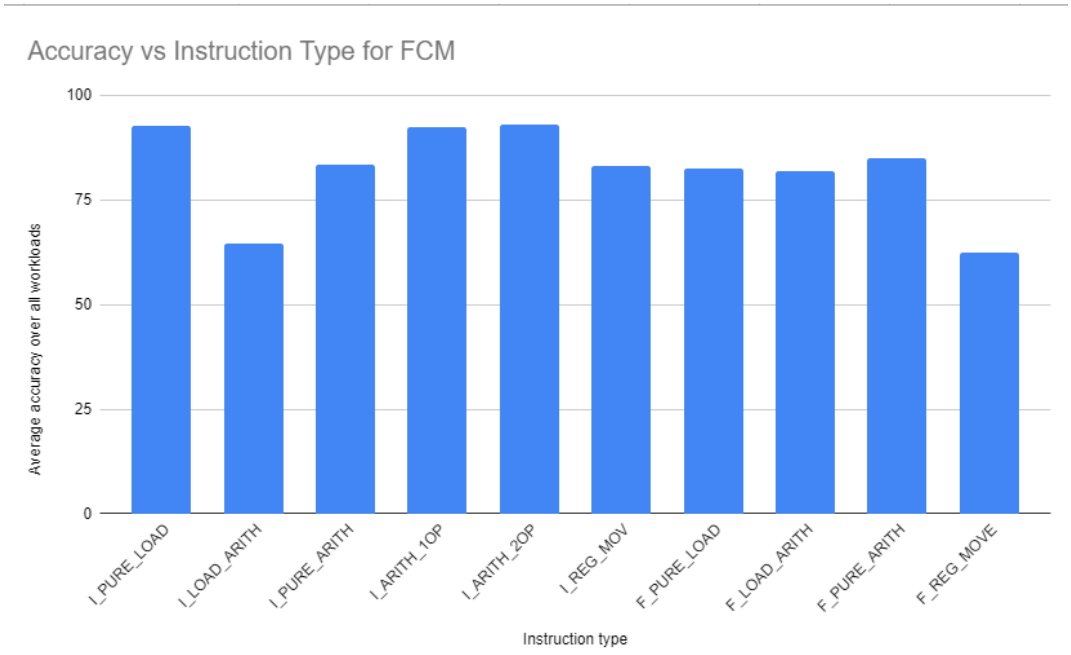


Figure 11. Average Accuracy vs instruction type for a PC localized FCM with a single value trigger.

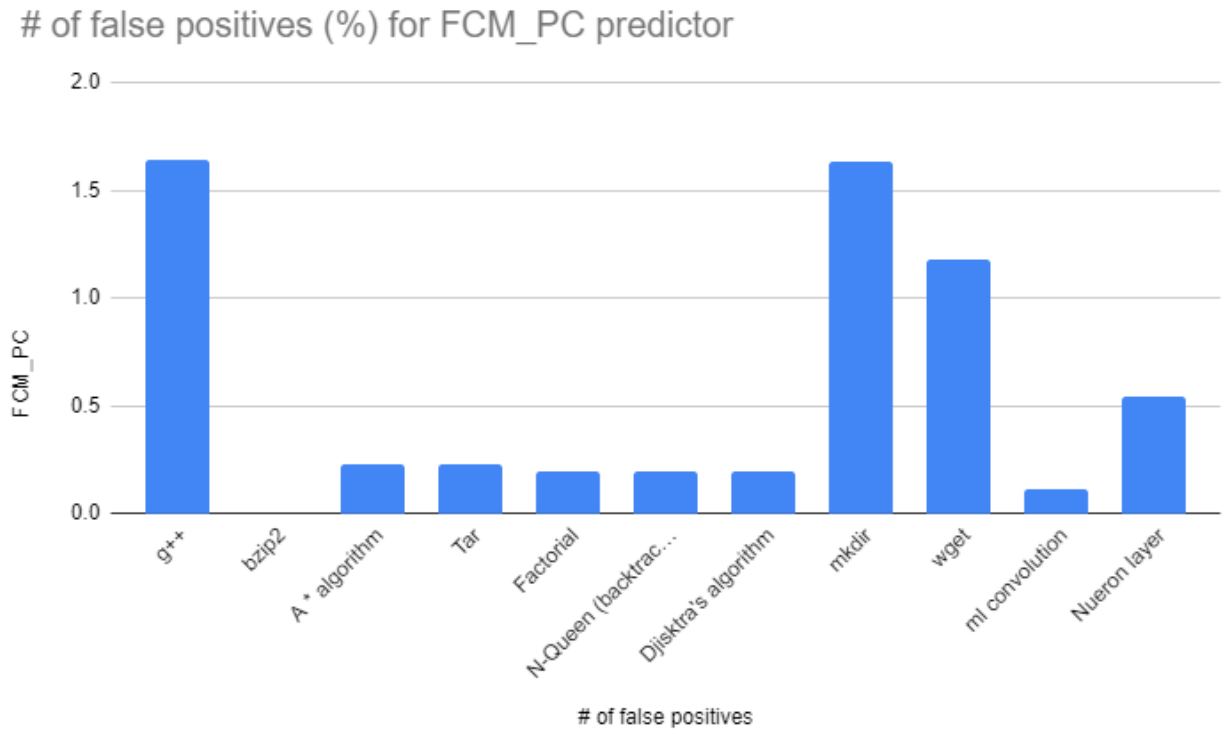
No. If we observe Figure 11, it does not do well on I_LOAD_ARITH, F_REG_MOVE. We don't care much about F_REG_MOVE as it doesn't constitute much of any workload. For I_LOAD_ARITH, if we use stride predictors it might lead to better overall performance.

More formally, we analyze the type of important instructions and come up with the best predictor for each kind of instruction.

Best predictor	Instruction type	FCM_PC	FCM_PC_thres_	FCM_history_4	Stride_PC	Stride_type	Last_value_PC	Last_value_type
fcm_pc	I_PURE_LOAD	**92.1**	86.25687524	86.72361701	90.09182044	87.00217885	53.33235357	9.758754077
stride_pc	I_PURE_ARITH	82.98	70.31921465	66.17478964	**83.24419205**	77.99468219	35.45231617	8.613750142
stride_pc	I_REG_MOV	82.39	69.38670199	69.27741148	**83.40874621**	74.90379058	41.32994314	37.55704291
stride_pc	I_LOAD_ARITH	62.66	48.08478048	53.79009467	**69.73855536**	59.42472822	42.77305147	8.045554828
fcm_pc	I_ARITH_2OP	**92.98**	85.77797101	87.45240888	92.49385154	87.04796657	77.67324543	2.904764982

The table shows the various accuracies across workload for these instruction types. Clearly from the above table for the dominant instructions, we choose Stride_PC for 3 kinds of instructions, and for 2 of them we choose FCM_PC as our best predictor.

Why would we get gains using this approach? Predicting the other unimportant instructions are actually hurting us, they don't contribute much to IPC. This is also shown in the table given below.



The table shows the % of false positives that were predicted that can be reduced by not predicting on these instructions. Even though this might not seem like much, ex- 1.187% of wget meansmispredicting on 20501 instructions out of 1726958 instructions. This means we reduce mispredictions on that many instructions. A similar analysis can be conducted for other workloads.

Thus when we construct a hybrid predictor using static analysis as shown above we get-
1)Average Speedup remained nearly same from 1.78x to 1.8x.
2)Prediction accuracy increased from 86.78% to 87% for the hybrid predictor.

The gains from the hybrid predictor are underwhelming. This might be because each predictor unlike previously is not learning from all the PC values. So either of the following can done as part of future work-
a)Use a dynamic scheme for making predictions
b)Let both the predictors learn from the data but only one make a prediction at a given point in time.
c)Come up with a better/more accurate metric for measuring speedup

This might prove the efficacy of this hybrid model better and give better speedups.

References

[1] Lipasti, M. H., & Shen, J. P. Exceeding the dataflow limit via value prediction. In Proceedings of the annual International Symposium on Microarchitecture (MICRO), 1996.
[2]Sazeides, Y., & Smith, J. E. The predictability of data values. In Proceedings of the International Symposium on Microarchitecture (MICRO), 1997
[3] Lipasti, M. H., Wilkerson, C. B. & Shen, J. P. Value Locality and Load Value Prediction. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1996
[4] Seznec, Exploring value prediction with the EVES predictor. CVP(2018)
[5] https://github.com/kdumontnu/CS246_Final_Project
[6] https://www.agner.org/optimize/instruction_tables.pdf
[7] https://www.agner.org/optimize/microarchitecture.pdf
[8] Sazeides, Y & Smith, James. Implementations of Context Based Value Predictors.
[9] Arthur Perais and Andre Seznec. Practical data value speculation for future high-end processors. pages 428–439, 02 2014
[10] Andre Seznec, Exploring value prediction with the EVES predictor. CVP (2018)
[11]C. Sakhuja, Anjana Subramanian, Pawanbalakri Joshi, Akanksha Jain, Calvin Lin. Combining Branch History and Value History For Improved Value Prediction. CVP(2018)