CSE-240C Simulation Assignment 1

Gandhar Deshpande

ABSTRACT

This report is written in fulfillment of the Simulation Assignment 1 for the Course CSE240C Advanced Microarchitecture, Winter 2022 at the University of California, San Diego. It looks at the submissions for the Instruction Prefetching Championship. It gives a overview of three of the submissions at a high level. It analyzes two of them in detail and reproduces their results. It then looks at two major design factors in the designs and tries to run an design space exploration to understand the impact of these factors on performance. Finally it looks at hardware budgets, and how the designs scale up and down with the budget. It also analyzes the timeliness of the two designs.

1 Introduction

Before we go into the various designs of prefetchers, I would first like to discuss the idea of Instruction Prefetching in general. A generic processor architecture is divided into two major parts, the "Frontend" and the "Execution-core". The execution core, as the name suggests is responsible for doing the actual execution, which is typically computation or memory accesses, but the processor is far more than just the computation. There is also a significant amount of metadata that goes with every computation, which contains information of what is the computation, where do you get the operands, where do you write the result and so on.

This is the frontend of the processor, which fetches, decodes the instruction so that it can be executed. There are major parts in this frontend, like a instruction fetch unit, which will get instruction from the cache, instruction decode unit which will decode the instruction into machine understandable signals. There are also other critical parts which support these functions, like instruction prefetchers and branch predictors, which function to make sure that the instruction cache is full of valid and useful instructions and how the control flow would change.

While in the early days, the processor would be limited by the slow computation, as time has passed, computation has become much faster, while memory still remains a bottleneck. In this context, when there is a branch misprediction, or an instruction cache miss, this is going to be a big penalty due to large latency in fetching from memory. With this, the frontend of the processor gains a lot of importance, since it contains the components which are going to reduce this bottleneck to the greatest possible extent. In this paper we will particularly focus on different designs of instruction prefetchers and their impact on performance as well as other factors like scalability and future applications.

The report is arranged as follows:

- Section 2 discusses three of the winners in the competition at a high level.
- Section 3 discusses the two of the prefetchers in detail going over various design factors.
- Section 4 shows the results obtained by the prefetchers.
- Section 5 analyzes key factors in the design which would affect the performance and shows a design space exploration over those factors.
- Section 6 analyzes the trend shown in this design space exploration
- Section 7 looks at the scalability of the design and produces the results at a lesser hardware budget
- Section 8 finds the trends in scalability.
- Section 9 looks at timeliness of the two predictors and the role of timeliness in prefetchers in general

2 Three winners

The three winners that I chose to analyze here are **FNLMMA**, **BARCA** and **EPI**. These three designs try different approaches to solving the problem of prefetching.

FNLMMA stands for Footprint Next Line Prefetcher + Multiple Miss Ahead Prefetcher. This design looks at existing literature and the drawbacks of each design and tries to counter those drawbacks by combining 2 prefetchers. The author first proposes the use of a Shadow cache which will be smaller than the Icache, and misses in this shadow cache will trigger a prefetch. Footprint Next Line prefetcher(FNL) prefetches up to K next lines if they are likely to be used in next N instructions. This reduces the amount of prefetches done and does not pollute the cache with unnecessary blocks which will go unused. FNL cannot prefetch blocks which are not contiguous. To work around this, Multiple Miss Ahead Prefetcher is used. The author first talks about a Next Miss Predictor which is used to predict the next miss in the ICache on getting one miss. The MMA is an extension of this which allows the prefetcher to predict not just the next block which will be missing, but the *nth* block which will be missing. This allows timely prefetches and can be used to cover access latency of the second level of cache.

BARCA tries to solve this problem with a different approach. The solution is based on searching a control flow

graph. Each node in the graph is a region which contains cache blocks. On certain demand accesses, this graph is searched depth first and the results of this graph are used as candidates for prefetching. The graph is agnostic to branch instructions and treats branching as adding another node in the graph. There is also a shadow cache which mirrors the L1I cache and is used for filtering prefetches as well as deciding the usefulness of the prefetches. They apply a lot of optimizations and tune these parameters to get an optimal solution which performs very well.

EPI is Entanglement Prefetcher for Instructions, which uses entanglement as a factor to decide prefetching candidates. The idea is to track a *source-entangled* cache line that should trigger a prefetch of a *destination-entangled* cache line such that the destination cache line is available in the cache in a timely manner. To do this, the algorithm tracks the latency between the cache miss occurring and the requested cache block actually entering the cache. It then finds the instruction which is at an equal latency before the cache miss occurred and entangles that instruction as a source and the missed instruction as a destination. This ensures that when the source instruction occurs, a prefetch is issued for the corresponding destination, which will make sure that the destination instruction is available in the cache when the control tries to execute it. There is further compression used to allow greater amount of data to be used that can be chosen as a design tradeoff.

3 Design of winners

In this section, we look at the details of two of the winners discussed in the previous section - FNLMMA and BARCA.

3.1 FNLMMA

FNLMMA is designed as a combination of two different each of which resolve a different problem. Together they are combined to give a better result. The major components used in this design are an I-shadow cache which is a smaller version of the instruction cache and is used to trigger prefetches.

For the FNL prefetcher, we have two 64K entry tables - Touched and WorthPF. These are 1 bit and 2 bit entries respectively. They are used in the following manner - When there is a I-shadow miss on block B, Touched[B] is set and if Touched[B-1] is set, WorthPF[B-1] also gets set to 3. This gets progressively decremented after a period of L cache misses, and the Touched table entry is reset. This acts as a partial resetting. The prediction algorithm basically says that if on Ishadow cache miss on Block B, WorthPF[B] is not null, prefetch block B+1, and check WorthPF[B+1]. If WorthPF[B+1] is not null, prefetch B+2 and so on. The maximum number of blocks to prefetch used in the design is 5 with a reset period of 8192.

The MMA algorithm uses a 8k entry miss ahead prediction table. Each entry in this table is 71 bits, 12 bits for partial tag, 58 bits for block address and 1 bit for managing confidence and replacement. Apart from this both prefetchers have filter, 16 entry filter with 58 bit entries for MMA to make sure that prefetch candidate chosen has not been prefetched recently, and a 128 entry filter with 17 bit entries for FNL which will work in a similar fashion. In the submitted design, the author uses upto 5 ahead prefetches from FNL and upto 9 miss ahead prediction in the MMA.

3.2 BARCA

The Barca prefetcher uses a control flow graph to link source and target regions to store information of cache blocks to prefetch on demand accesses. The regions contain blocks of cache which are prefetched. The graph adds a new edge pointing to a node when a new region is discovered. the edges are weighted with a 18 bit counter to count the number of times the edge has been traversed. The graph itself is modelled as a 256 * 64 set associative memory of edges. The tags used are the addresses of the source region. The replacement policy used in this graph is a least frequently used policy.

A search is initiated on a demand fetch, an instruction cache fill or a return instruction. A queue keeps track of recently accessed regions. If a new region which is not a part of this queue is accessed, a depth first search is initiated. The algorithm keeps a running product of the probabilities indicated by the edge weights to make sure that it crosses a certain threshold, else the algorithm considers this to be a wasteful search query. A shadow cache is kept which mirrors tags in the I-cache. This shadow cache is used for filtering useful and useless candidates. For every region searched, all blocks in the region that are not found in the shadow cache are considered candidates.

This algorithm uses a variety of structures to store metadata related to prefetching. They divide regions into a number of blocks, which are varied based on the size of the control flow graph.The ishadow cache stores the tags from i-cache, the tag for each block, a valid bit, a replacement bit for LRU and the address of source block from which it was prefetched. It has its own prefetch queue, from which a fixed number of candidates, in this case 4, are dequeued to the simulators queue at every cycle.

Along with this, the algorithm also uses some optimization techniques. The region addresses are compressed and used to index a 128 entry area table with random replacement policy. As mentioned before return calls are treated as a special case of branches with a particular target, which improves the accuracy of the prefetcher. The authors found that the prefetch queue within the prefetcher often is empty. To reuse this bandwidth, they also maintain a "would-be-nice" queue with lower probability candidates which are dequeued if the prefetch queue is empty, ensuring that the simulator prefetcher is getting the required number of entries every cycle. There is a recently searched list kept to filter out searches for blocks which have been searched in recent memory. Finally, they also tweak the probability counts to make it reflect the usefulness of an edge, not just the true probability. For this, they increment the edge count by 3 when a edge is traversed and is useful, decrement by 2 when it is useless, and increment by 5 when it is useful but late, to reflect that this edge is not only useful, but should be the top candidate so as to achieve timely prefetching.

4 Reproducing the winner results

In this section, we will plot the performance of the two prefetchers, Barca and FNLMMA and try to reproduce the

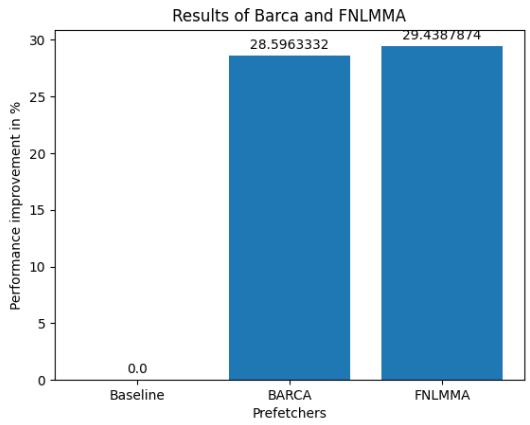

Figure 1: Performance of Winner Prefetchers

results obtained in the papers proposed. I am running the code with no modifications, exactly as it is found on the website. All IPC results are calculated with a geometric mean of the IPC in various traces. The results show the performance improvement normalized over the baseline. For reproducing results, all 50 traces are used in the baseline as well as in Barca and FNLMMA.

When compared to the reported statistics from the respective papers, we find that Barca reports 28.3% speedup, which is close to what I have reproduced. The FNLMMA paper reports a speedup of 28.7% which is a more marked difference from the reproduced result. However, a contributing factor to this may be an update to the code provided on the website, which does not strictly follow the parameters as stated in the paper. The code uses a MMA distance of 10, while mentioned as 9 in the paper. There are also other small differences like the a filter size of 24 in MMA. These differences may be the reason for this improvement in the performance.

5 Design Space Exploration

For the two designs chosen Barca and FNLMMA, I choose 2 parameters in each and vary them to see how they affect the performance of the prefetcher. For running this design space exploration, I chose a selection of 30 traces from the 50 available traces, due to time constraints and large amount of time taken for simulation. These traces were chosen randomly from the 50 traces and were selected to keep a fair representation of the various types of traces available like client, server and SPEC. I chose 5 client traces, 18 server traces and all the SPEC traces for evaluation of the design space exploration.

For FNLMMA, the parameters chosen for exploration were- ahead-distance for MMA and FNL distance(MAXFNL). For MAXFNL, I chose 2 values 3 and 5, while for ahead distance i chose 5 values - 7, 9, 12, 15, 25. I varied MAXFNL for various values of ahead distance and the results are plotted below. I also varied the reset period for the base design for 3 values - 4096, 12288 and 16384.

For Barca, I chose 2 parameters as the depth of the graph search and length of the recency queue. I varied the depth of

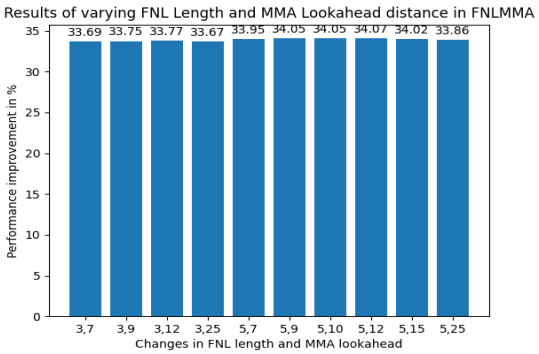

Figure 2: FNL+MMA variations

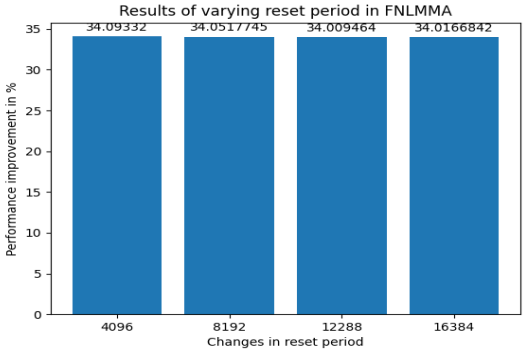

Figure 3: Reset Variations in FNL+MMA

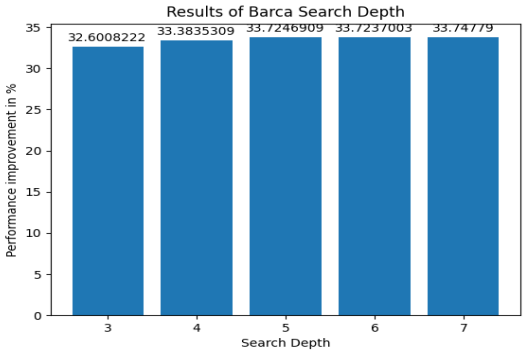

Figure 4: Search Depth variation in Barca

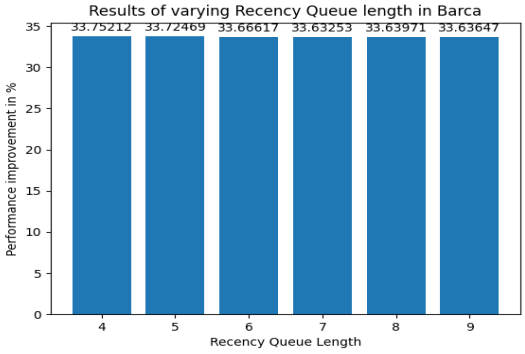

Figure 5: Recency Queue length variations in Barca

the depth-first search between values 3 to 7 and the recently searched queue length between 4 and 9.

6 Evaluation and Analysis

Since all the design space exploration was conducted with a set of 30 traces(traces available on github link in section 11), they are compared to performance of the baseline architecture against those 30 traces only. The values look to be larger than achieved in the final results. The reason for this may be the randomized selection of traces, which may have left out some of the worse performing traces, leading to better values.

Figure 2 shows us the effects of varying the length of the FNL prefetcher and the lookahead distance of the MMA prefetcher. We can see that as we increase the lookahead distance, the performance shows slight improvement, however it nearly tapers off around a distance between 9 and 12 and goes slightly below that as we increase more. In this paper, the author takes the value 10 for the lookahead distance. This may be due to the simulations being run on all traces gives a clearer winner for the value of 10. We can also see that for a FNL distance of 3, even with a larger lookahead, the prefetcher still performs worse than a smaller lookahead with FNL filter of 5. The author does state that beyond 6, the FNL filter does not improve, so it is likely that we would see similar results of the results dropping at or above 6.

Figure 3 shows the variations of the reset period and its effect on the prefetcher. As we can see, the effect of varying the reset period is not very large. We can see that 4096 seems to be performing slightly better than 8192, which is selected by the author. A possible reason for this choice could be that 4096 may be too short an interval for reset in some of the larger server loads where the instructions may take longer to repeat themselves.

Figure 3 shows the effect of changing the search depth of the depth-first search. We can see that a depth of 5-6 is ideal. However, there is a condition attached to this variation, which is the minimum probability that a running product must cross to be considered as a candidate. The authors do not mention any method of computing this probability, so to extend the model to a depth of 6 and 7, values were added based on the trend in the existing values. This could be tuned better perhaps to get a better performance. However I feel that the authors chose a value of 5, as at every increasing depth, the computation would compound and might result in late fetches and turn out to be counterproductive.

Figure 4 shows the effect of changing the length of the recently searched queue, which keeps track of the recently fetched blocks to filter out redundant prefetch requests. From the results we can see that having a larger recently searched queue is not very beneficial. One possible reason for this is that there are almost 24 instructions available at any given point, which would mean that having a filter for 6-8 addresses simply might not be enough to actually contribute much to the performance of the prefetcher.

7 Scalability

To understand the scalability of these two prefetchers, we must first look at what are the structures which contribute the most to the size of the prefetchers.

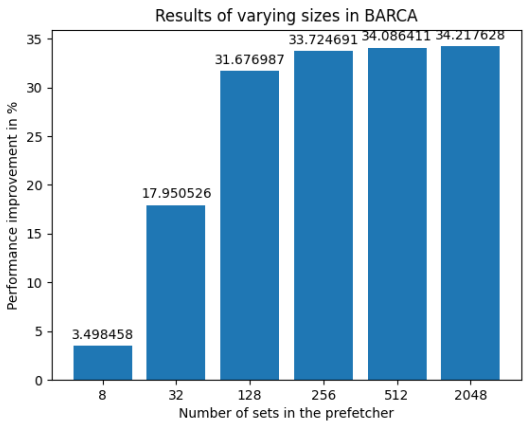

Figure 6: Varying the size of the graph

For Barca, the size of the Control Flow Graph is the major contributing factor towards the overall size of the predictor. There are of course other contributors in form of the various queues used in computation and storage of candidates, but those are trivial when compared to the CFG. The CFG is a set associative structure with 256*64 edges in the original paper. The size is varied by changing the number of sets in the graph while keeping the associativity constant across the various sizes. The paper does look at scalability and has given parameters which will need to be varied along with the size of the graph to provide best results. For example the Blocks per region need to be changed while varying the size to provide optimal results. In this case I am borrowing such values from the paper to get a best idea of the scalability. I change the size by varying the number of sets in the graph to the 8, 32, 128, 512, 2048 and observe the results.

For FNLMMA, the main size structures which contribute to the overall size are the Touched table, the WorthPF table and the miss-ahead prediction table. All of these are defined with a single variable in the code in logarithmic format, so the size is varied by simply changing the value of this variable, which changes the size in factors of 2. In this paper I evaluated the results with 1/8, 1/4, 1/2, 2, 4, and 8 times the size of the given predictor.

From the Figures 6 and 7, we can see that for a higher budget, both perform at an approximately similar level of performance which seems to plateau around the values chosen by both the papers. The returns seem to diminish around that point and the size increase is exponential in nature. The sizes compared for Barca are 10.63KB, 20.38KB, 59.38KB, 215.38KB and 839.38KB. The sizes of the structures in FNLMMA are 12.71KB, 24.58KB, 48.33KB, 95.83KB, 190.83KB, 380.83KB and 760.83KB. These are corresponding budget values to the factors in the graphs, lower to higher on the X-axis.

As we can see, the FNLMMA performs very well on a lower budget of 10KB, much more so compared to the Barca prefetcher around the same budget. However the Barca prefetcher exponentially improves its performance as the size increases almost matching the performance at near 60KB values and being comparable for all the budgets thereafter.

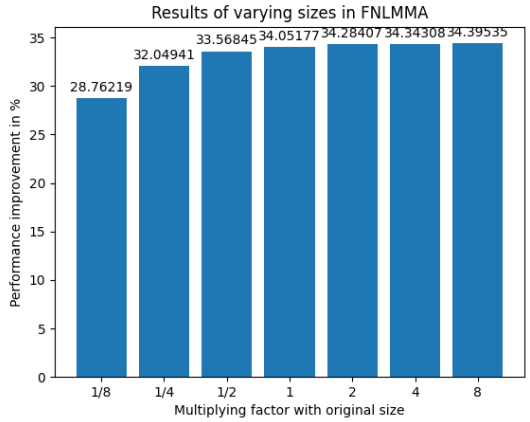

Figure 7: Varying the size of the tables

At higher budgets, both the prefetchers perform approximately in the same range. The FNLMMA prefetcher seems to have a slightly improved performance, however the performance is not scaling but is almost constant as the size increases.

8 Scalability Evaluation and Analysis

We now try the examine the reasons for the performance of these two prefetchers at scale. We find that the FNLMMA performs much better compared to the BARCA prefetcher at a lower budget. One of the reasons for this could be the fact that the BARCA algorithm uses a control flow graph, which means that the as the size goes down, so does the amount of data that can be stored in each of the nodes, which results in a poorer performance as compared to the FNLMMA, which uses tables to prefetch the data.

Another reason might be that the algorithm of the control flow graph requires enough data so as to be able to compute the depth first search candidates and if the data stored is not enough, it might lead to wrong prefetches resulting in the edge counters getting reduced simply due to lesser data to work with.

One other reason which might induce this result here is that the parameters chosen in the algorithm were tuned particularly for the model submitted for a budget of 110KB. This means that the algorithm may require tuning the parameters to suit the new size.

THE FNL+MMA in contrast performs well due to the fact that the FNL uses two tables which are reset every 8192 instructions. Since the size of the tables is reduced, it may be possible that different instructions end up modifying the same counter, making sure that it does not get reset, which may be a contributor towards an improved performance at a lower size.

9 Timeliness

Timeliness is an important factor in prefetcher design. The reason for timeliness playing such an important role is that prefetching too early may cause the processor to think of the block as unused and be discarded, and late fetch may result in a miss.

Both the algorithms have techniques to ensure that the result is fetched in a timely manner. The FNLMMA prefetcher uses 2 prefetchers in combination to ensure that prefetches are done in a timely manner. The FNL prefetcher is predicting a footprint of the next line which needs to be prefetched, which would ensure that not only is the next block available on a prefetch, but also consecutive blocks if they appear to be useful. This reduces the number of prefetches to be issued for the next block. The MMA looks at multiple ahead misses, to see a pattern of what the future misses would be in case of a miss based on history. This too will ensure that these blocks are fetched in time to reduce the number of prefetches requested. To ensure that the data is not too much in the future, the miss-ahead prediction table predicts only up to 9 blocks ahead of the miss. The author also shows that predicting up to 30 does not provide much difference in the performance. This value should be chosen to cover the latency of the lower level cache, so that timeliness gets assured.

The Barca prefetcher uses a limited depth first search of the graph to ensure that prefetches are done in a timely manner. To ensure that the algorithm takes into account the timeliness, they modify the probability counter mechanism to incorporate timeliness. The counter works on the principle that every useful prefetch increments the edge value by 3 and every useless prefetch decrements the edge value by 2. However, if the prefetch was useful, but was late, the increment is done by 5 not by 3 to integrate into the algorithm, the usefulness of this edge so that the next time this edge gets higher priority. The data is prefetched and kept in a prefetch queue which is used to populate the prefetch queue of the simulator. Along with this, there is also a "would-be-nice" queue, which also stores candidates which do not have enough space in the prefetch queue. All of this ensures that there are multiple prefetch candidates available at any point in time. Since the depth-first search is searching for 5 levels, this will ensure that

10 Implementation details

The code used and implemented in this project can be found in the public github repository : `https://github.com/gdpande97/CSE240C_Sim1` The repository also contains all the result files generated from the various simulations, as well as scripts written to run the simulations in a batch mode and scripts to extract relevant values from the results.

The championship simulator is not included in this, and instructions on how to integrate these code pieces into the simulator are mentioned in the ReadME.md file.

The championship simulator can be found at `https://github.com/ChampSim/ChampSim`

11 Acknowledgments

The whole study is done using the infrastructure provided by the 1st Instruction Prefetching Championship. I use their simulator for simulating all the prefetchers, with the logic for all mechanisms apart from the prefetcher provided by the Championship. I also use the code made available by the various winners in the competition to the competition to simulate and explore the design space of the two designs studied in this paper.

12 References

[1] Daniel A. Jimenez, Paul V. Gratz, Gino Chacon, Nathan Gober BARCA: Branch Agnostic Region Searching Algorithm.

[2] Andre Seznec. The FNL+MMA Instruction Cache Prefetcher.