Aggressive Prefetching for Graph Application at Last Level Cache

Patel Mit Kumar
Computer Science and Engineering
Indian Institute of Technology
Ropar, INDIA
2020csm1016@iitrpr.ac.in

***Abstract*—Graph processing is the most essential processing and analysis technique for diverse area such as wide variety of big data applications etc.. Graph analytics outperforms traditional data structures and relational databases due to its ability to clearly express relationships between things. However, performance for single-machine with in-memory graph analytics is severely limited at the LLC because to inefficiencies in the memory subsystem. We analyse the memory hierarchy for graph processing workloads and propose a prefetcher in this study, which consists of two contributions. first, we analyze the memory(cache) hierarchy for graph processing workloads, and second, we propose a prefetcher architecture that outperforms other state-of-the-art prefetcher for graph benchmarks.**

First, we perform a critical in-depth analysis of graph workload in Cache especially in LLC (Last Level Cache) for single-core and Multicore OoO (Out of order) CPU. We have analyzed 1) LLC (Last Level Cache) Miss rate for Load, RFO and WriteBack Access 2) Bandwidth Utilization by LLC and MainMemory for without Prefetcher and with Prefetcher Like BOP(Best Offset Prefetcher), SMS, VLDP (variable Length data Prefetcher) 3) Performance improvement with Increase in LLC size 4) Degree of Prefetcher of VLDP, BOP, etc.. 5) LLC cache Block Characterization based on their Access cycle. As a result, we conclude that an aggressive Prefetcher has a good Potential to Perform better than a state-of-the-art Prefetcher Like BOP, VLPD, and SMS for Graph Benchmark.

Second, based on our observations, we propose Aggressive Prefetcher, based on global history for graph applications. Aggressive Prefetcher will predict block within some constant range using nearest recent Accesses. In addition, our Aggressive Prefetcher has a dynamic degree of prefetching so its degree may vary with Graph Benchmark Our Prefetcher achieves 2%-130% performance improvement over a no-prefetch baseline, 2%-25% performance improvement over a conventional SMS prefetcher[2006], 1.3%-56% performance improvement over a Variable Length Delta Prefetcher[2015], up to 8.2% performance improvement over a best-offset prefetcher implemented as best offset to prefetch[2016] and Bingo Prefetcher[2019] the state-of-the-art prefetcher for graphs.

I. Introduction

Graph processing is currently widely used for large scale data-related problems in various fields for example recommendation systems, web search, social networks, financial money flow, air traffic flow, fraud detection, etc The high potential of the graph is derived from a rich, meaningful, and widely applicable data that consists of a set of entities interconnected by relational links. In this project, we will closely observe the memory pattern of a large scale Graph Application and its bottleneck behavior and then we will design a last level cache prefetcher to utilize the Memory bandwidth optimally.

Earlier work on graph analysis and processing was very much dependent on on a hardware platform such as distributed system, Distributed Memory, Non-core Systems, and some Specialized Accelerators. But we have observed that in recent time many industry and academic data are fit Comfortably in Single RAM at the server Due to the availability of such huge RAM and affordability of high memory density in Single Machine recent time. As a result, single RAM memory machines are an appealing alternative to other machines. As In contrast to typical distributed systems, a small scale-up big-memory system does not require consideration of any of these factors. the difficult task of graph partitioning among nodes and the avoidance of large networks overhead in communication In the industry [7] [8], many programming frameworks have been developed for this type of platform [9] [11]. While non-core systems and accelerators need massive computing power, To increase locality, preprocessing is used to create partitioned data structures. However, the performance of any single-machine with in-memory with large scale graph analytics is bounded by the critical inefficiencies of the memory subsystem, which possibally make the all cores stall as they continuiously wait for data of application to be fetched from the DRAM for Computation. This means there is a scope for good accuracy prefetcher to prefetch Data from Memory before actually requesting to improve CPU performance.

The goal of this aggressive prefetcher paper is to address the memory inefficiency and other issue for single-machine with in-memory for large scale graph analytics. First, we perform a critical in-depth analysis of graph workload in Cache especially in LLC (Last Level Cache) for single-core and Multicore OoO (Out of order) CPU. We have analyzed 1) LLC (Last Level Cache) Miss rate for Load, RFO and WriteBack Access 2) Bandwidth Utilization by LLC and MainMemory for without Prefetcher and with Prefetcher Like

BOP(Best Offset Prefetcher), SMS, VLDP (variable Length data Prefetcher) 3) Performance improvement with Increase in LLC size 4) Degree of Prefetcher of VLDP, BOP, etc.. 5) LLC cache Block Characterization in dead block or polluted block based on their Access cycle. and possible scope to enhance the performance of CPU by designing a special prefetcher. As a result, we conclude that an aggressive Prefetcher has a good Potential to Perform better than a state-of-the-art Prefetcher Like BOP, VLPD, and SMS for Graph Benchmark.

Second, based on our observations, we propose some guidelines and we propose an aggressive prefetcher, which has reasonable accuracy but will predict blocks aggressively using a global history buffer for graph applications (benchmarks). An aggressive prefetcher will predict a block within some constant range and search for the nearest access point to the address. In addition to that, we store evicted blocks by prefetch block address along with their respective cycles to analyse the pollution due to the prefetcher. Our prefetcher achieves 2%-130% performance improvement over a no-prefetch baseline, 2%-25% performance improvement, and 9 out of 9 times performing better over a conventional SMS prefetcher [2006], 1.3%-56% performance improvement, and 9 out of 9 times performing better over a Variable Length Delta Prefetcher [2015], almost 8.2% performance improvement, and 7 out of 9 times performing better over a best-offset prefetcher [2016] and 5 out of 9 times performing better than the bingo prefetcher [2019], the state-of-the-art prefetcher for graphs.

The paper is organised as follows.In Section 2 we will discuss the Background and Related Work on Prefetcher. In Section 3, we will perform a detailed analysis of the graph benchmarks in the cache, especially in LLC, which contains experimental setup benchmarks. In Section 4, we will characterise the behaviour of graph applications on the basis of LLC miss rate, BW (bandwidth) utilization, etc., and lastly, we will conclude the argument using observations from the above section. Section 5 proposes one aggressive prefetcher and gives a detailed description of our baseline microarchitecture. In Section 6, we will present an experimental evaluation of the prefetcher with different configurations, comparing it with other prefetchers. Specifically, Hardware Prefetcher, as well as a supporting argument for the results. Finally, This study concludes in Section 7.

II. Background AND Related Work

In this section, we discussed on Classification of hardware prefetchers and related work done on hardware prefetchers for graph benchmarks.

A. Background

Prefetching is a technique for getting a block hit in the cache by bringing a piece of data to the cache before it is requested by a processor.it essentially provides the following benefits:

1) Reduce the Compulsory misses
2) Reduce memory Latency

There are many different design patterns for prefetchers. Usually, the hardware prefetcher is divided into two categories.

1) Conservative Prefetcher: Conservative prefetchers utilize a very low degree of prefetching, usually less than one. The following are the benefits and drawbacks of conservative prefetcher.

1) conservative prefetcher usually Reduces potential for cache pollution since they are predicting very very fewer blocks with high accuracy so there will be minimum pollution
2) Likely higher accuracy, since it is predicting very less so it will try to predict the best possible block from all available potential Prefetch addresses. it likely has high Accuracy as shown in Fig. 14 where VLPD is very conservative so it has high accuracy than our Aggressive Prefetcher.
3) Conservative prefetcher send relatively fewer requests to DRAM many it has high accuracy so most of those prefetch request hit at LLC so very negligible lower bandwidth
4) Likely lower coverage

2) Aggressive Prefetcher: Aggressive prefetchers utilize a very high degree of prefetching, usually greater than one. The following are the positive and negative points of conservative Prefetcher:

1) Well ahead of the load access stream
2) Aggressive Prefetcher Hides memory access latency better because we are sending the request in very frequently, so in the pipeline model, it reduces the latency for memory access as shown in fig. 11 where our prefetcher is sending many requests, still overall it could manage to hide latency
3) Aggressive prefetcher have usually Higher coverage since we are prefetching a large number of the block so it will reduce LOAD and RFO Miss so coverage will be increased according to Equation. 2
4) downside of Aggressive Prefetcher is lower accuracy since we are prefetching a lot of blocks so high possibility that many of them might not be useful, so it reduces the Accuracy according to Equation no 1.
5) it requires higher bandwidth since we are sending too many requests to the DRAM
6) it needs to evict useful Regular block from LLC in order to add Prefetch block in LLC which leads to pollution of block which is the major downside of Aggressive Prefetcher.

B. Related work

SMS is SMS (Spatial Memory Streaming) SMS captures page-level access patterns by storing spatial patterns by storing bit-vectors and uses this vector bit to prefetch the block. The prefetching degree of SMS varies with benchmarks. it has a very large history buffer, as shown in TABLE IV. [13] [6]. Our prefetcher is more accurate and uses less history table size.

VLDP is Variable Length Data Prefetcher. it keeps store deltas histories of the Conservative request at Cache line which are Miss at the cache and using these deltas it prefetches the next offset for prefetcher Address within one page. this deltas history is used for other blocks for prefetching.it has good accuracy and a conservative prefetcher. which bound VLDP performance in Graph benchmarks Significantly. see in Fig. 8. [12]

BOP(best offset prefetcher), tries to learn the best offset using recent accesses and, after the learning phase, predicts the address using the best offset within the page. The good thing about this BOP is that it has good accuracy. It is a conservative type prefetcher. The degree of prefetching is fixed. The original prefetcher has a degree of one [10] which bounds its performance in graph applications.

Bingo is a Spatial data prefetcher. that predicts based on PC+delta, PC+Offset, PC+Address, etc... Bingo spatial data prefetchers in which the short and the long events are used to select the best access pattern among all possible prefetch possibilities for prefetching. using PC+ Offset and PC+Address as the event we try re-reference a single history table to find the associated access pattern. and this helps to remove storage overhead. [3] and [2]

DROPLET tries to use inherent reuse distance to find different types of dataproperty data, structure data, intermediate data and property data triggered by structure data which is have overcome the serialization. It has good accuracy, but since it has graph prepossessing which classifies addresses, it belongs to which category structure or property that requires software support, it does not fall completely into the hardware prefetcher category. [4]

Graph Prefetching Using Data Structures The Knowledge paper is also explicitly designed for graphs and it has a 2.3 to 3.3 speed up, but this prefetcher is designed only for the BFS (breadth-first search) algorithm. [1]. Since it was not possible to perform better in different algorithms like single-source shortest path, connected component, etc., our work allows us to perform better in all graph algorithms.

III. EXPERIMENT SETUP

In this section, we present the experimental setup and the benchmarks used for our Exploration.

A. Simulating Platform

For performing experiments, we have used the Champsim simulator. An x86-based simulator is used in the trace simulation model. We select ChampSim over another simulator because it has the support of all existing state-of-the-art prefetchers, so we can compare our results with theirs. Moreover, it has a good prefetch plugin model where we can directly write out prefetching logic without editing the existing simulator. The baseline architecture parameter of ChampSim is described in Table III. We have done experiments on single-core and multi-core (4 cores). We are running 400M instructions per simulation to study the behavior of graph applications. and collect stats like bandwidth utilization, coverage, etc..

Algorithm	Description
Breadth First Search	Traverse a diracted or undiracted graph level by level using queue
PageRank (PR)	Rank each vertex on the basis of the ranks of its neighbors
Single Source Shortest Path (SSSP)	Find the minimum cost path from a source vertex to all other vertices which are connected to source vertex
Triangle Counting (TC)	Triangle Count is a graph community detection algorithm that counts the number of triangles that travel through each node in the graph.

TABLE I
ALGORITHMS

Dataset	vertices	edges	Description
road	1.97M	2.76M	mesh network
amazon	410K	2.43M	social network
webgoogle	916K	4.32M	social network

TABLE II
DATASETS

B. Benchmark

We employ the standard GAP benchmark set [5], which comprises of optimised and multi-threaded C++ implementations of some of the most popular and commonly used representative algorithms in graph analytics, such as BFS, linked component, and TC (Triangle counting), among others. We chose GAP over a standard software framework for graph profiling in order to rule out any framework-related performance overheads and uncover the real hardware bottlenecks for graph benchmarking. We employ four GAP algorithms, which are listed in Table I. and Table II summarises the datasets that were utilised to create benchmarks. It contains both unweighted and weighted graph data. [5]

IV. CHARACTERIZATION

To understand the memory-bound behavior we analyze the cache hierarchy and Memory Bandwidth.

A. observation 1 - LLC Missrate Access

We observed that graph benchmarks have a huge LLC Miss Rate of almost 29% average1 we also observed that a data

Parameter	Configuration
ROB size	352
Processor width	6
Fetch width	4
SCHEDULER SIZE	128
L1-D	SET=128 , Way-Ass=8 , Latencies=1
L1-I	SET=128 , Way-Ass= 12 , Latencies=1
LLC(last level cache)	SET=2048*Number of core , Way-Ass=16, Latencies=10 /* common for all core */
Main Memory	SIZE = 8GB , Read Qeue size is 64
Branch Predictor	Bimodel

TABLE III
CPU SIMULATOR CONFIGURATION

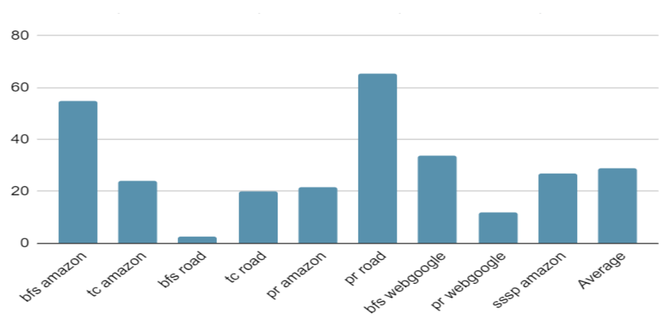

Fig. 1. LLC Miss Rate for Load and RFO

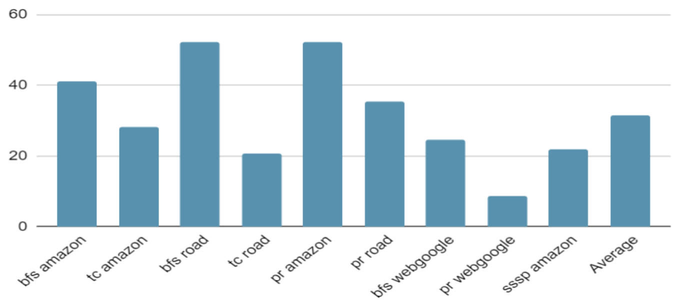

Fig. 2. Average Cycle difference between two Memory request at LLC

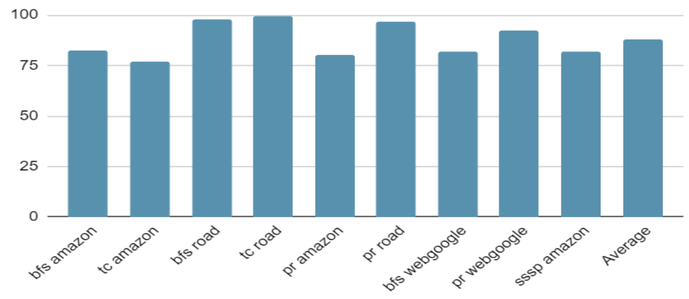

Fig. 3. percentage Dead block Evicted by LLC controller

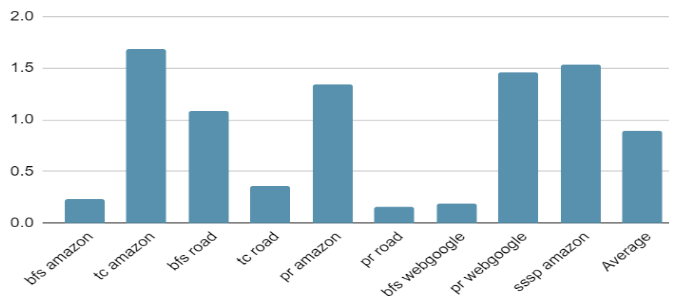

Fig. 4. Percentage of evicted bock which are re-requested by LLC withing next 1M cycles of their respective eviction

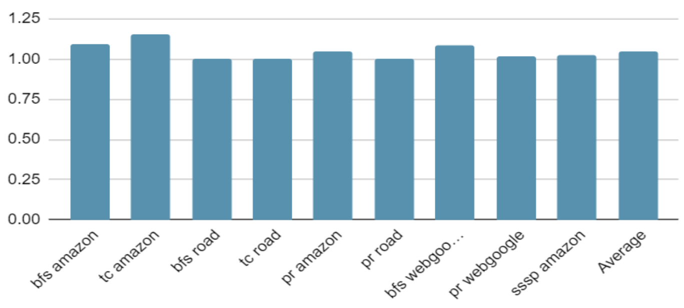

Fig. 5. SpeedUP Achieved by 4X cache size

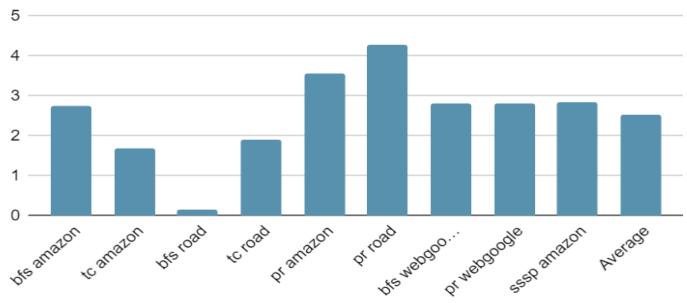

Fig. 6. Percentage of Bendwidth Utilized by LLC to MM

request either read or write will be served after an average of 32 cycles. Fig.2

B. *observation 2 - Evicted Block analysis*

Almost more than 87.74% blocks which are evicted in graph benchmarks are dead block 3 and rest 12.26% are being used after some cycles only on an Average 0.893% of the total evicted block are re-demanded by L2 within the next 1M cycle after the eviction of that block. it is shown in Fig4. using this information we can say that the block used once is very likely (more than 99%) that the evicted block will not repeat in the next 1M cycles. that's why it has a very high miss rate. because it keeps demanding a new block. so once it is used we can evict it. and bring a new block. there is no point to keep it in LLC once it's demanded full-fill to L2.

C. *observation 3 - Effect of Cache size*

To check the locality of the graph trace , we increased the LLC cache size to 4X (four times) and then measured the speedup over 1X LLC cache size. The improvement was negligible. Simulation hardly manages to improve around 5% of performance. So it supports the above observation 2 that blocks are used once only and their accesses are not repeated within an LLC for the majority of accesses, so increasing LLC size is not increasing IPC. Fig. 5

D. *observation 4 - Bandwidth Analysis*

By observing the bandwidth utilization between LLC and Main Memory, we found that utilization is very low on average, around 2.52% Fig.6 which is very less as compair to other SPEC benchmarks.

E. Conclusion and Opportunities

By observing the above observations and the experiment accuracy fig9 it seems that in graph benchmark it is hard to get good accuracy in hardware-based prefetchers. It is, even harder when you have limited information like an address and type of address. usually PC and other information are not observable at LLC in real-time Machine. even state-of-the-are prefetcher like BOP(best offset Ptefetcher) and VLDP(variable length data prefetcher) has less Accuracy around less then 70%. By observing No 1 and No 4, LLC has a high miss rate, so many requests are going to DRAM, but bandwidth utilization is low, so if we take risk opportunity of prefetching requests earlier than their actual access, then we might be able to hit those blocks at LLC, so eventually, it improves performance. Since it has less bandwidth utilization, even if we send many prefetch requests aggressively, it can still handle it. Observation No 2 and No3: increasing the size of an LLC does not improve performance, and it We have almost 87% of the blocks evicted being dead blocks, so we can aggressively evict blocks and try to prefetch block, which might have even less chance of access. Why Aggressive Prefetch will work

1) **Likely lower accuracy** LLC has almost 87% block are evicted are dead block means they will not access in future and only 0.89% of total evicted blocks are accessed in the next 1M cycle. which is very less so if we consider this 0.89% block will cause pollution than it is very less so we can take the risk of evicting those block from the cache and try to prefetch more blocks which might have reasonably fair chance of access. it is better to prefetch block who have even more then 20% chance of getting hit than keeping those evicted block which has only 0.89% chance of being used in next 1M cycle stats are shown in fig 3 and fig 4
2) **higher bandwidth:** prefetching Aggressively it generate more Memory Request to the DRAM so it will utilized more Bandwidth.
3) **Higher coverage, better timeliness :**Getting good accuracy in a graph benchmark is very hard to achieve using only addresses and core info. Instead of prefetching a high-accuracy block, try prefetching a large number of blocks. which has less accuracy, but when combined, it will give us a better chance to hit the block in LLC (Last Level Cache).
4) **Hides memory access latency better :**On average, there is around a 35 cycle gap between two consecutive Ideally, an LLC hit can handle one request and require four or five cycles using the pipeline. So it already has a gap of 35 cycles. So we can utilize this gap by sending prefetching in between these two gaps. So, sending requests aggressively will not cause any performance harm.

   Request Aggressively, the statics of the prefetcher simulation are shown in Fig.2 and after Aggressively Prefetching in Fig. 11

V. AGGRESSIVE PREFETCHER ARCHITECTURE

In this section, we first provide an overview of our proposed prefetcher. Next, we discuss the detailed architecture design of Aggressive Prefetcher.

A. Overview and Design Choice

In this section, we will provide details of our prefetcher, its relevance and implementation.

since we want to make our Prefetching Algorithm Aggressive so the degree of prefetcher should be high it should be atleast greater than 1. degree of Prefetcher means how many prefetch requests are generated based on one address. As you can see in Fig 7 we are generating a Prefetcher Request based on the input address at LLC and then trying to search in LLC. If it is already in LLC, then we do nothing, but if it does not hit at LLC, then we send this Prefetch request to the lower level of the memory hierarchy, which is the main memory for LLC.

Our algorithm is prefetching the address by observing the recent history of addresses. We have taken the size of the recent history buffer to be 1024, but we can take it more. The more it has a history, the more it will be aggressive and accurate. But due to hardware overhead, we chose to make it 1024. The same is true for Max Stride Size. We must select the MAX Stride Size of 31, but it can be increased for greater aggressiveness.

Our core logic of the algorithm is very simple. We are making the chain for three addresses with the same delta, where delta is the difference between two addresses. Using the current address and the address from the recent history buffer, We first fetch all addresses whose delta with the current address (current request address) is less than the Max Stride Size, and then we prefetch addresses based on the below equation. If we did not find any recent addresses within the range, then we will prefetch the next block address since we do not want to miss the opportunity to prefetch in aggressive prefetching.

eg.

Recent Table = [15 , 54, 75 , 96, 940, 1005] and

MaxOffset (O)=25
currently request is X = 66

then fatched address will from range [66 - 25 , 66 + 25]
Range is [44 , 91]

so we will fatch address from recent table which lies in this range [44 , 91]
which are { 54 , 75 }
for 54 it will prefetch X + delta
where delta = (66-54) and X = 66
Prefetch address = 78 /* sequence 54 – 66 – 78 */

for 75 it will prefetch X + delta
where delta = (66-75) and X = 66

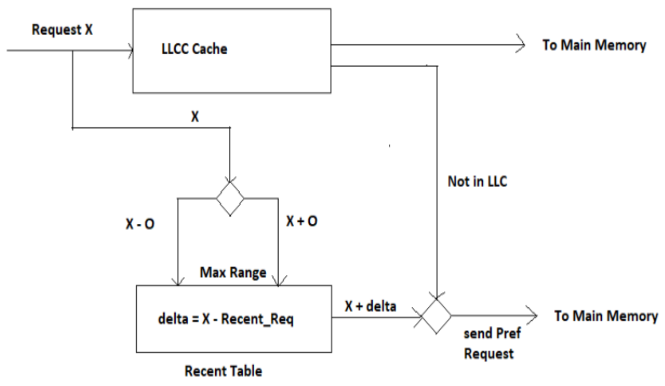

Fig. 7. Block Diagram of Aggressive Prefetcher

Prefetcher	Description
Vldp	PREFETCH DEGREE = 4 DHB SIZE = 16 OPT SIZE = 64 DPT SIZE = 64
SMS	REGION SIZE = 2 * 1024 PHT SIZE = 16 * 1024

best prefetch offset	RR$_TABLE_SIZE = 1024$
	DEGREE = 1

Bingo	REGION SIZE = 2 * 1024 PHT(History table) SIZE = 16 * 1024

TABLE IV
PREFETCHER PARAMETER

Prefetch address = 57 /* sequence 75 – 66 — 57 */

As we can see, it will generate multiple prefetcher requests from one address, so its degree will always greater than 1. A detailed algorithm can be seen in the algorithm 1.

B. Hardware overhead

In this section, we will discuss the hardware overhead required to setup an aggressive prefetcher at LLC (Last Level Cache). In our prefetcher, we don't use a lot of components and information about blocks, or any extra information like PC (Program Counter), etc. Instead, we just use the address and the type of request, such as read or write.

In our prefetcher, we have one recent history buffer. which can store up to a maximum of 1024 recent block addresses at any given point of time. The other is the Prefetch queue. The Recent History Buffer has a fully associative memory overhead and is a LRU (Least Recently Used) replacement-based buffer. By assuming that the address is 40 bits and an extra 8 bits for other meta data, We require 6 bytes per entry, so the buffer size is 1024*6 bytes, which turns out to be 6 KB of memory overhead. Except for this, we are not using any extra hardware support. 7

VI. EVALUATION

A. Prefetch Experiment Setup

We have set up four different state-of-the-art prefetchers like vldp (variable length data prefetcher), SMS, BOP (best offset

Algorithm 1 Prefetcher

```
Recent [1024] /* will store recent 1024 Address Block */
MaxRecentAccess ← 1024
Log2BlockSize ← 6 /* Block Size is 64 Byte */
MaxStride ← 31

procedure PREFETCHER OPERATE(ADDRESS)
    countPrefReuest ← 0
    RequestBlock ← ADDRESS ≫ Log2BlockSize
    Recent.ADD(RequestBlock)

    i ← 0
    loop:
    if abs( RequestBlock - Recent[i] ) < MaxStride then
        addr ← RequestBlock + (RequestBlock - Recent[i])
        addr ← addr ≪ Log2BlockSize
        Prefetch Address (addr)
        countPrefReuest ← countPrefReuest + 1
    i ← i + 1.
    while i < MaxRecentAccess.

    if countPrefReuest == 0 then
        addr ← RequestBlock + 1
        addr ← addr ≪ Log2BlockSize
        Prefetch Address (addr)
    return
```

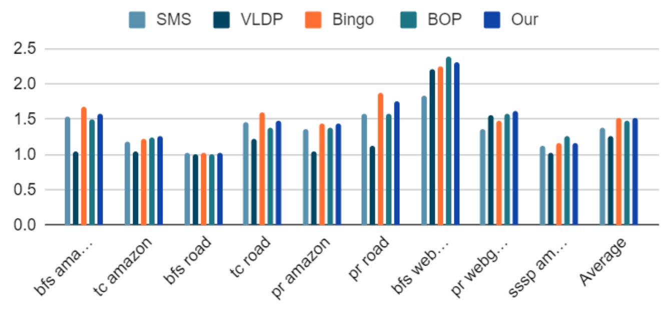

Fig. 8. Speedup with Base line (no prefetcher)

Ptefetcher), and Bingo Spatial Data Prefetcher. The parameters used for this prefetcher are in the table IV.

For our prefetcher, we have selected the Recent History Table Size of 1024 addresses and the Max Stride Size of 31.

B. Performance Results

In this section we will do in depth discuss on results or Prefetcher on Graph Beanchmarks.

1) Speed Up: As demonstrated in Fig.8, vldp (variable length data prefetcher) performs the poorest of the five prefetchers, but bingo and our prefetcher perform almost equally, although bingo requires 16 times more Large History Table. Best offset prefetcher (BOP) and SME orderly follow.

In every benchmark, our prefetcher came in first or second place. This is not the same as bingo or the best offset

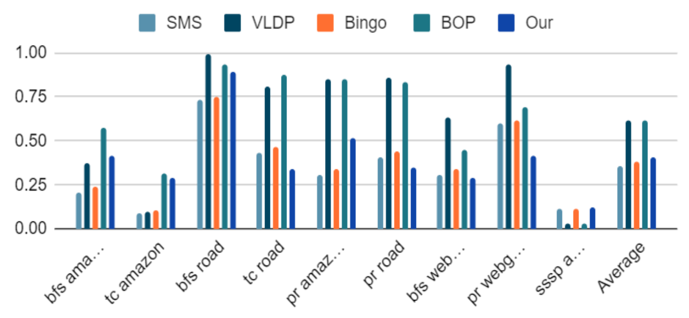

Fig. 9. Accuracy of Prefetchers

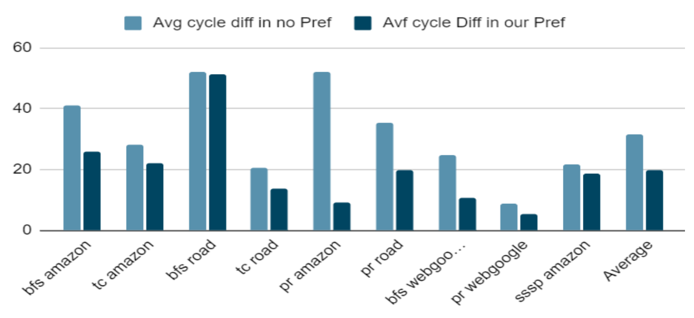

Fig. 11. Average cycle difference between two Memory request at LLC

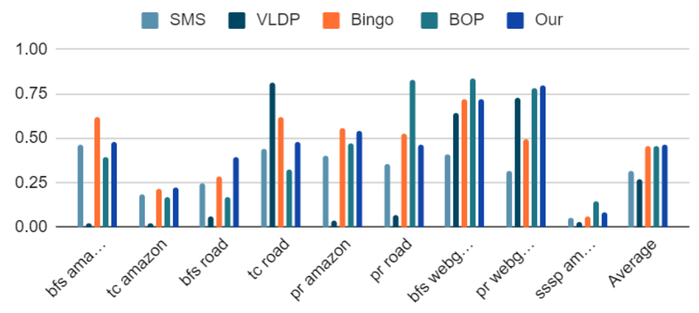

Fig. 10. Coverage of Prefetchers

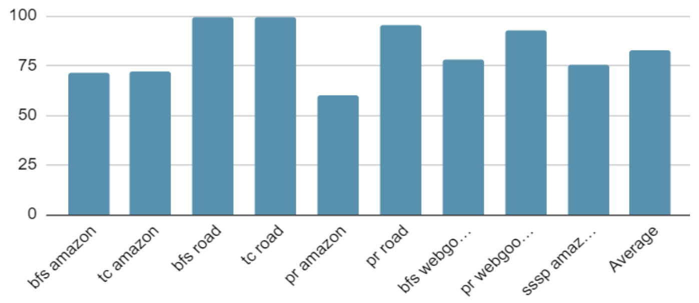

Fig. 12. Prefentage of evicted block never Access by LLC again.

prefetcher. In all benchmarks, our prefetcher outperformed VLDP and SMS prefetchers.

from Fig.14 we observed LLC miss rate for LOAD and RFO is Inversely Proportional to Speedup. whee re miss is is low then speed up is high , one other observation is if Non-Prefetcher has higher LLC Miss rate then our algorithm get high speed up in case of PR road benchmarks. LLC miss rate is 63% so we get speed up 1.75 similar to BFS web-google where Miss rate is 35% so our prefetcher get speed up 2.3 and on opposite case in bfs Road non Prefetcher has 3% miss rate so our prefetcher got speed up 1.019 only.

2) Accuracy: As shown in Fig9 we are not getting good accuracy, and it was expected that our algorithm is designed aggressively, so our goal was to prefetch more blocks. with reasonable accuracy. And still, we have better accuracy than SMS and Bingo Prefetcher.

$$Accuracy = \frac{(useful)}{(useful + useless)} \quad (1)$$

3) Coverage: As shown in Fig 10 we are getting the highest coverage, followed by bingo, bop, SMS, and VLPD orderly. VLDP has very good accuracy, but it has the least coverage among all prefetchers, which is the reason behind the lowest IPC of VLDP in all benchmarks.

$$Coverage = \frac{(useful)}{(useful + LoadMiss + RFOMiss)} \quad (2)$$

C. Explaining Aggressive Performance

In this section, we will discuss why our prefetcher is working better than other prefetchers, like 7 times in BOP (best prefetch offset), 9 times in VLDP (variable length data prefetcher), 9 times in SMS, and 5 times in bingo, out of the 9 benchmarks shown in fig 8.

For calculating pollution, we are considering that ”Any block which is evicted by prefetch block will re-request by LLC within 1M cycles after its eviction, then we consider that block as a polluted block.” Our prefetcher has almost 0.7% block of the total evicted block is a polluted block. As illustrated in Fig.13

As shown in Fig 9 we are not getting good accuracy. but we are prefetching a lot of blocks and pollution in our prefetcher in very less and the majority of the evicted block were dead block is shown in Fig 13 and 12 so overall our prefetcher is managed to get better results.

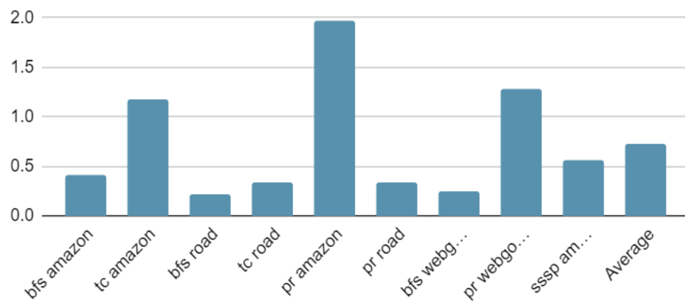

Fig. 13. Prefentage of block re-access by LLC within 1M cycles

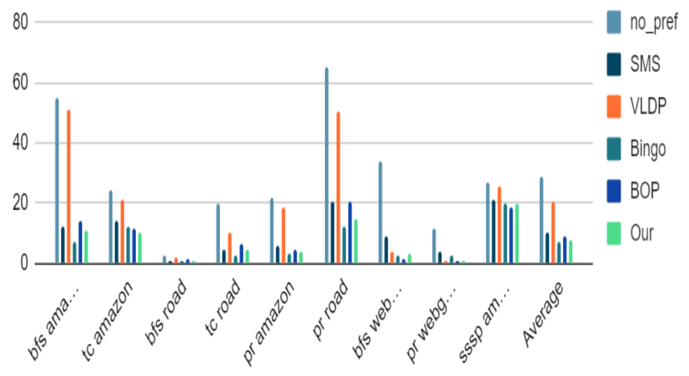

Fig. 14. LLC Miss rate for LOAD and RFO in LLC

Using the below equation, we can justify our performance gain. We get better results with a higher useful block.

$$usefullblock = Accuracy * TotalPredicted(toDRAM) \tag{3}$$

Since we have less accuracy but we are predicting a lot of prefetchers due to the aggressive nature of the algorithm, the number of blocks hit in LLC (helpful block) is larger than other prefetchers. As in opposition to our prefetcher, VLDP has very good Accuracy Fig9 but it prefetches very fewer blocks, so it actually generates fewer useful blocks and eventually decreases its performance. We can explain this by throwing coverage Fig 10 also if we can see equation no. 2. that indicates that for higher coverage, we need to minimize load and RFO misses, so if we are hitting more blocks to LLC, then we can say it has performed better. Our prefetcher has a very low LLC miss rate, shown in Fig. 14

VII. Conclusion

In this paper, we have analyzed the behaviour of the cache hierarchy for the multi-core system for graph benchmarks. Based on this analysis and observation, we can conclude that the aggressive prefetcher has a very good scope in graph benchmarks. Based on our observation of graph benchmarks, we propose an aggressive prefetcher. which is based on finding some of the best offsets using recent history to prefetch blocks. Experimental results show that our Aggressive Prefetcher outperforms previous work such as VLDP (variable length data prefetcher), SMS, BOP (best offset prefetcher), and Bingo. For SMS and VLDP, it outperforms in all benchmarks. In BOP, it performs better in 7 out of 9 benchmarks, and in Bingo, it outperforms 5 out of 9 benchmarks.

It is observed that accuracy is relatively less important than coverage for graph benchmarks. As it was proved in the VLDP prefetcher, highest accuracy, worst speedup. An aggressive prefetcher performs well when LLC has a huge dead block or a high LLC miss rate.

References

[1] Sam Ainsworth and Timothy M Jones. Graph prefetching using data structure knowledge. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–11, 2016.

[2] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Domino temporal data prefetcher. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 131–142. IEEE, 2018.

[3] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 399–411. IEEE, 2019.

[4] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. Analysis and optimization of the memory hierarchy for graph processing workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–386. IEEE, 2019.

[5] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[6] Michael Ferdman, Stephen Somogyi, and Babak Falsafi. Spatial memory streaming with rotated patterns. *1st JILP Data Prefetching Championship*, 29, 2009.

[7] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 505–514, 2013.

[8] Shizhi Jiang, Yiwei Ci, Qiusong Yang, and Mingshu Li. Matryoshka: A coalesced delta sequence prefetcher. In *50th International Conference on Parallel Processing*, pages 1–11, 2021.

[9] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1–20, 2016.

[10] Pierre Michaud. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 469–480. IEEE, 2016.

[11] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. Graphjet: Real-time content recommendations at twitter. *Proceedings of the VLDB Endowment*, 9(13):1281–1292, 2016.

[12] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 141–152. IEEE, 2015.

[13] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. *ACM SIGARCH Computer Architecture News*, 34(2):252–263, 2006.