A Data-aware Prefetcher for Accelerating Graph Analytics

M.Tech Project Stage 2 Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Anubhav Jais
(Roll No. 203079012)

Under the guidance of
Prof. Virendra Singh

Department of Electrical Engineering
Indian Institute of Technology Bombay
January 6

Acknowledgement

I express my gratitude to my guide Prof. Virendra Singh for providing me the opportunity to work on this topic.
I would also like to thank Varun Venkitaraman and all the seniors and my batchmates in the lab for the continuous guidance and support.

Anubhav Jais
Electrical Engineering
IIT Bombay

Abstract

Graph analytics is the field of processing, storing, and analyzing graphs. Graph processing is used in numerous applications which operate on large data sets. Its application includes social media, web searches, and VLSI design optimization. Graphs can better express the relationship among entities as compared to other data-structure. CPU-based graph processing is inefficient because graph workloads have very irregular memory access patterns, which causes the cache hierarchy to perform poorly.

Large graphs are compressed for reducing storage overhead and follow irregular memory access patterns because data is retrieved through multiple indirections. Cache Prefetching is a main memory latency hiding technique implemented to boost the performance of the processor. Prefetchers implement Cache Prefetching to tackle the famous “memory wall” problem, where the memory access from slow memories like RAM, hinders the performance of the system.

This work proposes a new technique for graph prefetchers to get higher performance than the existing prefetchers. Some of the prefetchers studied are 'Prodigy', 'Droplet', and others. Studying all the prefetcher designs, this work proposes a Data-aware prefetcher that can prefetch neighbourhood data into the cache. The idea proposes to find the degree of the vertices and select the vertices with a degree greater than a pre-defined threshold, and make prefetching decisions accordingly to boost performance.

Contents

List of Figures

Chapter 1

Introduction

Graph processing is used in multiple domains like social networks, e-commerce recommendations, bio-informatics, network analysis, path planning, machine learning, and others. It has lots of potentials since many real-world problems can be mapped to a graph. Due to large data size and arbitrary connectivity, graph processing faces multiple issues like random memory accesses resulting in long-latency main-memory access. Graph processing still performs poorly in the current computing system, despite many attempts to improve its efficiency.

Graph-based algorithms like breadth-first search and page rank are widely used on various platforms like social media and machine learning which generates big data in the form of graphs. These applications are essential tools of graph analytics to process large graphs. The performance offered by these applications is hindered by irregular memory accesses. Compressed sparse row(CSR) format is one such way to store graphs which requires multiple in-directions to get to the required data, for example, A[B[i]] is an indirection where to get the value from array A, B[i] must be first accessed. Such data dependency leads to poor system performance due to excessive DRAM stalls caused due to continuous memory access. To overcome such stalls, data and instruction caching are used to store data for faster access. Techniques like cache replacement and cache prefetching are employed in the caches to improve the effectiveness of the caches.

Simple cache prefetching solutions try to remember the access pattern by calculating the address change often called stride [8], [9], [10], [11] they perform poorly for irregular memory patterns. Some hardware and software prefetchers use machine learning to predict the upcoming address requirements[6], [7]. These prefetchers rely on complex machine learning algorithms and are currently not practical.

To increase the performance of any computer architecture, one of the main objectives is to have an apt-cache replacement policy. However, due to the irregular workloads of the graph, even the best replacement policies fail to give the desired outcome. The traditional policies all are good for streaming applications, but for graphs, they result in a lot of memory traffic leading to stalls. Belady's MIN replacement policy is the ideal case for how the cache should work, but the problem with this policy is, it requires oracular knowledge of the replacement decisions which is not possible in the real world.

In recent times cache prefetching solutions for graph processing have been designed[1], [2], [3], [4], [5]. They rely on the address information of various lists present in formats like CSR. Prefetchers for graphs are often hybrid and use software for finding out the address bounds of lists of the graph and hardware support to make prefetching decisions. Droplet and Prodigy are some of the prefetcher techniques in recent times. Works like Prodigy[1], an L1-D prefetcher which uses software support to form a Data Indirection graph(DIG). DIG is used to store the address bounds and capacity of all CSR lists. Once the DIG is formed, the prefetcher can easily index into the lists like vertex lists to prefetch.

Hence it is clear that to improve the performance of the graph workloads either design a better replacement policy that can take care of the graph's irregular workload or use a graph prefetcher to remove the memory latency that originates because of graph workloads.

The main focus of this work is centered around the graph prefetchers.

1.1 Background

1.1.1 Graph data-set representation

A graph *G(V, E)* has a set of vertices represented by V and a set of its edges represented by *E*. In a directed graph an edge *e = (u, v)*, indicates an edge from u to v ,where u and v are the source and destination vertices respectively. In an undirected graph it just denotes an edge between u and v. Adjacency matrices and adjacency lists are two common methods for representing graphs in memory. Finding a relationship between vertices, u and v, is easy with an adjacency matrix. The most significant disadvantage of using an adjacency matrix being that it requires $O(V^2)$ bits of storage space. The storage problem is magnified in the case of sparse data, which has few links between any two vertices. An adjacency list is a more compact version of the adjacency matrix. However, because graph traversal causes irregular memory access patterns, it incurs pointer chasing overheads when traversing the list over address space.

Real world graphs which are sparse are stored in storage efficient manner using the Compresed Sparse Row (CSR) format. Figure 1.1 shows the

CSR representation of a simple graph with out-edges to a vertex. CSR uses couple of arrays:

- **Offset array:** offset[i] stores the neighbour array offset and links to the relevant property array index for a vertex id i.

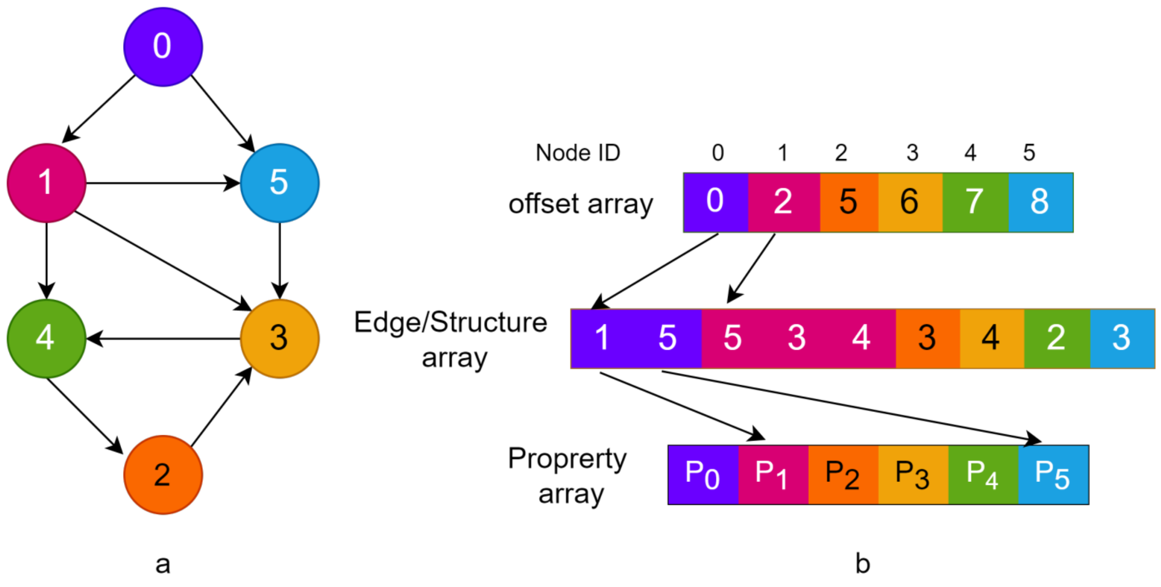


Figure 1.1: Compressed Sparse Row (CSR) graph using in-edges to a vertex

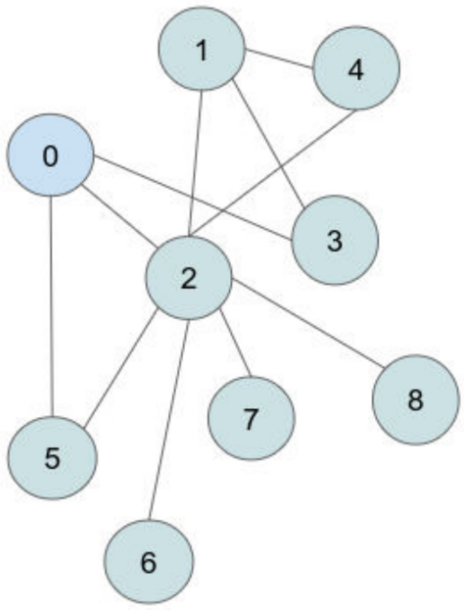
- **Structure/Edge/Neighbour array:** The edge array keeps each vertex' neighbours in a continuous manner. neighbour[j] (where j $\in$ [offset[i], offset[i + 1])) stores the out (in) neighbour vertex ids of vertex i.
- **Property array:** Property[i] stores computed or partial results pertaining to a vertex id i e.g., The page rank application uses two property arrays, one of which maintains the rank of all vertices for the current iteration and the other of which stores the rank of all vertices for the previous iteration.

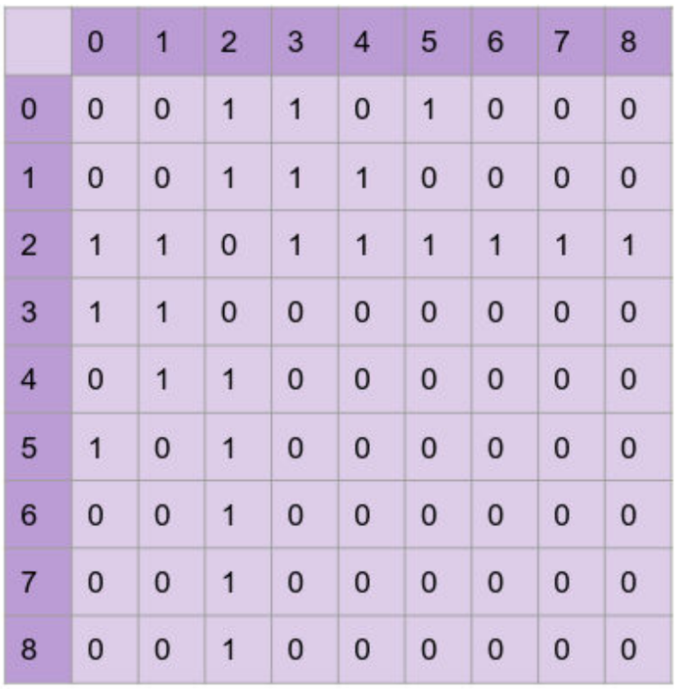
Graph data is categorize in three types: Intermediate data, edge data, and property data.

1. **Edge data:** The edge data is accessed to obtain the neighbour vertices. These data are present inside the edge array.
2. **Property data:** This stores the data of a particular vertex. These data are present inside the property array.
3. **Intermediate data:** Any data other than that of edge or property data like local variable and queue.

1.1.2 Adjacency Matrix

Both of these representations are common in storing graphs. Fig.1.2 shows a graph with 8 nodes and 10 edges. Adjacency matrix for such will be a matrix of size NxN where N represents the number of vertices in the graph. The element at row x and column y represents the weight of an edge between the nodes x and y. For a unweighted graph $w_{xy} = 0$ if edge is absent between two nodes and $w_{xy} = 1$ if an edge exists. Also $w_{xy} = w_{yx}$ for an undirected graph. Indexing into the matrix provides $O(1)$ search time for accessing the edge and node information. The storage requirements of the adjacency matrix are very high with respect to other graph representations which $O(N)^2$.





	0	1	2	3	4	5	6	7	8
0	0	0	1	1	0	1	0	0	0
1	0	0	1	1	1	0	0	0	0
2	1	1	0	1	1	1	1	1	1
3	1	1	0	0	0	0	0	0	0
4	0	1	1	0	0	0	0	0	0
5	1	0	1	0	0	0	0	0	0
6	0	0	1	0	0	0	0	0	0
7	0	0	1	0	0	0	0	0	0
8	0	0	1	0	0	0	0	0	0

Figure 1.2: Adjacency Matrix of a sparse graph

1.1.3 Adjacency Lists

Adjacency Lists are listed representations of a graph. For each vertex list of only its neighbours are stored in a list. So, an adjacency list is a collection of lists of neighbours of all the vertices. The searching time in adjacency lists is $O(V)$ and $O(E)$ is the space complexity, where E and V are the number of edges and vertices in the graph respectively. Fig 1.3 shows an adjacency list, here indexing into node 3 provides the list of its neighbours which are 0&1. It can be seen that the redundant zeroes representing the absence of an edge in the adjacency matrix are removed and the representation has become more compact.

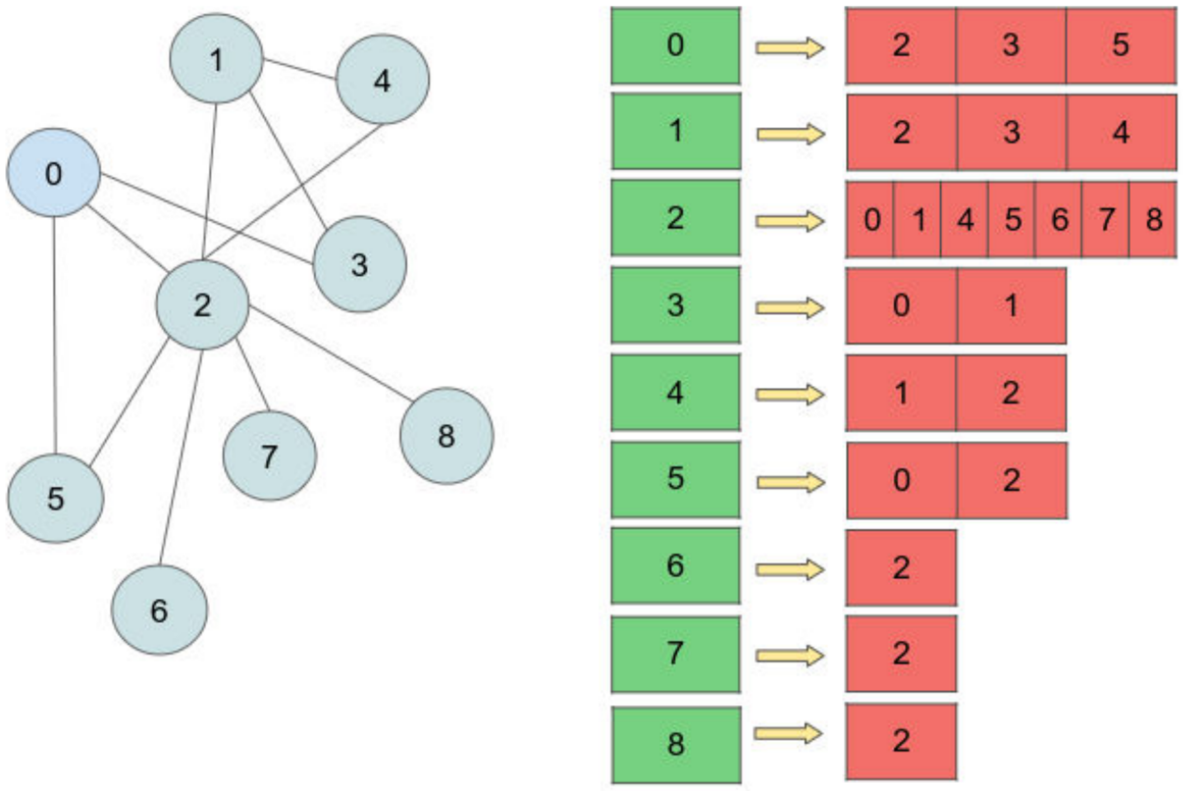


Figure 1.3: Adjacency List

Graph processing characteristics and bottleneck

- **Irregular memory access:** Due to the dispersed nature of graph connection, graph computing often requires a huge count of irregular memory accesses. As a result of such access to the main memory having high-latency, on-chip caches are mostly ineffective, resulting in poor usage of computing resources.
- **Atomic overhead:** Large-scale graph data is often analysed in parallel. To reduce data contention, such concurrent graph data processing uses atomic operations. In general atomic execution requires multiple operations and results in a significant increase in performance overhead
- **Irregular computation:** The computing workloads for various vertices can vary significantly because of the power-law distribution. This will result in a significant workload imbalance as well as increased communication overhead.

1.1.4 Graph Algorithms

Graphs in the industry are of huge sizes and various applications or algorithms are run on them to process and analyse the data. Some of those applications are:

- **Breadth first search(BFS):** BFS is a graph traversal algorithm which traverses the graph's breadth or nodes of the same level in one iteration thus being called breadth first search. The source node is pushed to a queue and then popped for processing. After that one by one its neighbors are pushed inside the queue, a node is popped and the corresponding neighbours are pushed inside again. This process continues until all nodes are covered. bfs keeps track of accessed nodes by maintaining a visited list.
- **Page Rank(PR):** Page rank is a web page ordering and recommendation algorithm named after Google's co-founder Larry Page. PR assigns pagerank scores to all the vertices.
- **Connected components(CC):** Connected component algorithm is used to find out the weakly or strongly connected components in the graph. Each connected sub graph and its nodes are given a label and disconnected nodes are also provided with a unique label.
- **Betweenness Centrality(BC):** BC algorithm is applied on a graph to find out how central or connected a vertex is with respect to other vertices.

Graph applications have been classified into three categories here:

- **Complete Frontiers:** Applications which process all the nodes in all the iterations, like PageRank.
- **Partial Frontiers:** Applications which process a subset of nodes in an iteration and the size of subset keeps going down with iteration. Like PageRank-Delta.
- **Jumbled Frontiers:** Applications which process all the nodes but in a jumbled manner and frontier size can vary. Like BFS.

1.1.5 Cache Prefetching

Cache prefetching is a technique to improve system performance by hiding the main memory latency access. Prefetchers try to bring the data and instructions to the cache for faster memory access than the memory access to the DRAM. Prefetching involves various ways to predict the addresses to be accessed by the core in the future. The prefetcher stores those predicted data into the cache, therefore, making the cache prone to pollution in case of incorrect predictions. The overall aim of a good prefetcher should be to learn the memory access patterns, through which it can predict the behaviour of applications that usually are highly dynamic.

Prefetcher performance is analyzed by the following metrics:

- **Coverage:** Coverage tells the effectiveness of the prefetcher for a given system.

$$\frac{Cache\ misses\ Eliminated\ by\ Prefetching}{Total\ Cache\ Misses} \tag{1.1}$$

- **Accuracy:** Accuracy is the measure of the correctness of the prefetching algorithm.

$$\frac{Cache\ misses\ Eliminated\ by\ Prefetching}{Total\ Prefetched\ Addresses} \tag{1.2}$$

- **Timeliness:** Timeliness is a measure of how early required block is prefetched into the cache before it is referenced by processor.

1.1.6 Power Law

Power Law in graphs relates to the fact that a small percentage of nodes exhibit a higher degree, thereby connecting with more nodes than others and defining the majority of the graph's structure. It can be formally described using Barabási–Albert's (BA) model. It uses preferential attachment mechanism, an algorithm to generate random scale-free networks. Internet, the World Wide Web, and social networks are am example pf scale-free networks. They contain few nodes with a higher degree as compared to the other nodes in the network.

Chapter 2

Literature Review

Graphs have highly irregular memory access patterns and hence present the problem of memory bottleneck due to these irregular accesses. One possible solution to reduce the memory latency and thereby increase the processor performance is to prefetch the data beforehand and store it in the cache. This technique is called Prefetching.

Figure 2.1 shows the performance of the measure of an in-memory

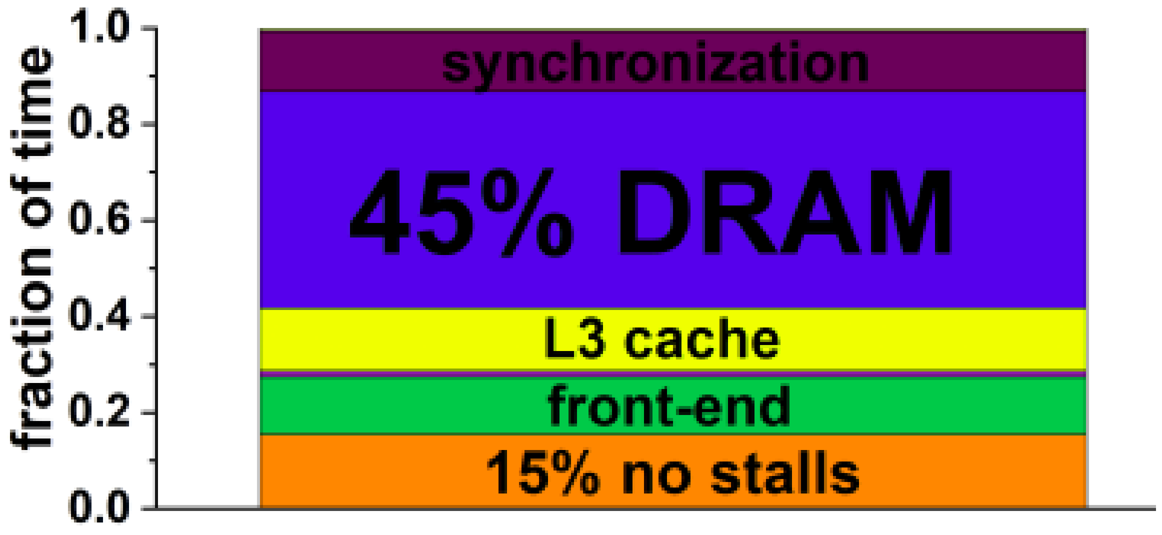


Figure 2.1: Cycle stack of PageRank on orkut dataset

system for graph analytics. It is bounded by the inefficiencies in the memory subsystem, which makes the cores stall as they wait for data to be fetched from the DRAM. This experiment was done on PageRank benchmark with orkut dataset and it was observed that 45% of the cycles are DRAM-bound stall cycles whereas the core is fully utilized without stalling in only 15% of the cycles.

Prefetch degree is how many cache lines are prefetched at a time. Stream buffers, Stride prefetching, and GHB-based are some of the widely used prefetchers.

There are a lot of basic prefetchers available, like Next-line prefetcher which prefetches A+1 line for every cache line A that is fetched. Stream prefetcher prefetches multiple A+1 lines at a time. Stride prefetchers are similar to stream except that they prefetch at a certain stride(offset X) distance (A+X). There are also some table-based prefetchers like Global history buffer(GHB) prefetchers. They use a History Table which is indexed by some key, e.g., PC or load address. They have an index table which is accessed by directly indexing into it. GHB is a FIFO with pointers between entries.
However, these prefetchers fail to identify the irregular memory accesses patterns associated with graph workloads. Hence we need graph prefetchers that exploit the storage patterns of graphs to prefetch more efficiently.

2.1 Graph Prefetcher

Cache Prefetching is a technique to improve performance of the computing systems. But prefetching must be used wisely as it can bring pollution to the cache defeating the purpose of its use. To learn different mechanisms of prefetching for graph analytics a literature survey was carried out. Most of the prefetchers use the CSR format to represent graphs. The CSR format allows linear storage of all the data which gets divided into lists or arrays of constant size for the static graphs. Prefetchers can be software, hardware or hybrid.

Most of the recent works incorporate a hybrid approach like Prodigy[1], Graph Prefetching using Date structure knowledge[2], Droplet[3] and IMP: Indirect Memory Prefetcher [4].

Indirect memory prefetcher(IMP)[4] is used to access single-valued indirection patterns such as:

ADDR(A[B[i]]) = Coeff × B[i] + BaseAddr

The coefficient is the size of each element in A and BaseAddr is A[0] address. Hence the only unpredictable factor is B[i]. If B[i] is known, we can find out the Addr[A[B[I]]]. IMP prefetches and reads the value of B[i + x] into cache. It uses predicted values of Coeff and BaseAddr to calculate A[B[i + x]].
The idea is to identify at least two (B[I], ADDR(A[B[i]])) pairs, such that the Indirect Pattern Detector(IPD) can use them to find prefetch address. For this, it finds index-address pairs from the set of accesses that the cache sees until it finds a coe. and BaseAddr that satisfies the above-mentioned equation, for at least two of them. Then by using these two equations, we can find the values of coe. and BaseAddr, which can be used later to in the equation to find addr[A[B[i]]].

The problems with this work is that it only supports single-valued indirection and also it prefetches only part of the graph data structure.

Droplet[3] exploits the memory heirarchy and its usage patterns to predict and store the prefetches.

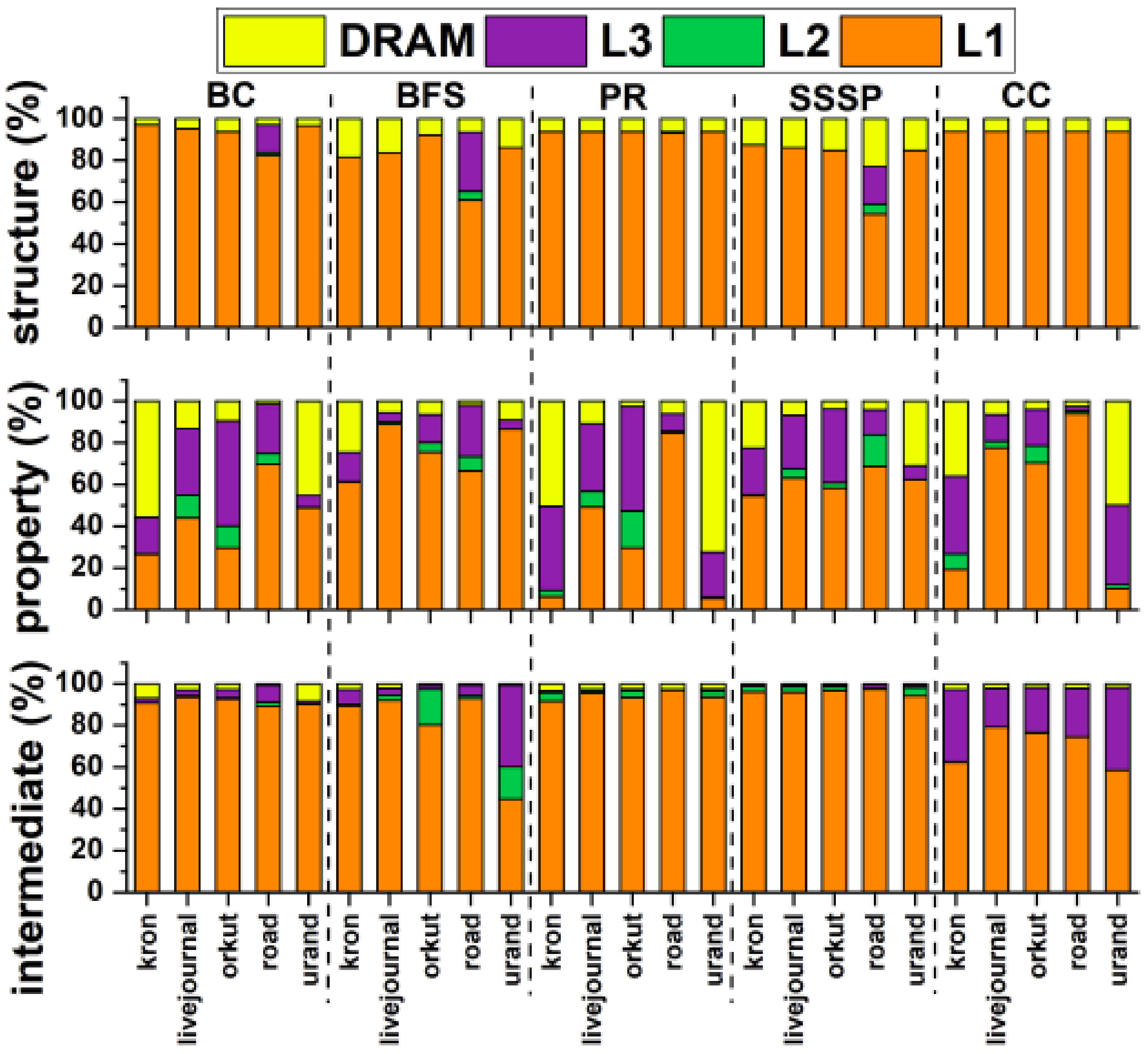


Figure 2.2: Breakdown of memory hierarchy usage by application data type

Figure 2.2 shows three data structure in graph CSR format: Structure, Property and Intermediate data. It can be seen that most of the property data loads missed in the L1 cache cannot be serviced by the L2 cache but can be serviced by the LLC and the DRAM. It is evident that the accesses to intermediate and Structure data are mostly on-chip cache hits in the L1 cache and the LLC. The reuse distances of the three data types explain why the private L2 cache fails to service the memory requests.
This shows that private L2 caches show less utilization. DROPLET consists of two data-aware components:
- The L2 streamer for prefetching structure data.
- A Memory controller based property prefetcher (MPP)

MPP uses the streaming structure data to prefetch property data into L2. From the graph it was shown that structure data requests almost always go to DRAM, hence the stream prefetcher fills the L2 with structure data, and also a copy of structure data is given to MPP which triggers the prefetch of property data into L2.
The problem with DROPLET is inaccurate prefetch timeliness i.e. either the prefetch was too early, in which case the prefetched data got evicted before getting used, or the prefetch was serviced too late.

Prodigy[1] is an intelligent prefetcher that uses a co-designed hardware-software solution to reduce memory latency. It's a software-hardware co-designed prefetcher. It's an L1-D cache prefetcher which triggers on an access to work queue and prefetches data from vertex list, edge list and property array.

Prodigy targets graph applications, Machine learning, linear algebra applications, and others. These applications consist of two types of data-dependent access patterns. Prodigy uses static program semantics from software and dynamic information from hardware, thereby combining both hardware-software solutions.

The main contribution is a Data Indirection Graph (DIG). It is used to represent the program information of data structure. It also stores how the memory is accessing the data structures. It is a low-cost solution that can prefetch data depending on the data structure information and how it is traversed. It maintains timeliness by varying the prefetch distance. It can work with CSC and CSR formats.

Prodigy's performance is compared with non-prefetch techniques and with other graph prefetchers. It uses only 0.8kB of storage. It outperforms modern and non-prefetch prefetcher solutions by 1.5-2.3x and 2.6x respectively. On average, it also saves energy by 1.6x.

The knowledge of Data Structure is used to design a prefetcher. It uses CSR format to represent graphs. By snooping on the load requests to the cache from the processor and by prefetcher, it can generate new prefetch requests. Since L1 is the closest to the processor, hence prefetching into it will provide the best performance. The best performance is obtained by prefetching the visited and edge data. The problem arises from the fact that work queue and vertex list data are needed to access those data structures. Hence it is designed with the bound addresses of all four arrays.

It can give timely loads by snooping the L1 data cache and its own prefetcher data and then making decisions accordingly. It achieves a performance gain of 3.3x and 2.3x on average.

Chapter 3

Scope of Improvement and Proposed Idea

3.1 Issues with Prodigy

Prodigy is a state-of-the-art prefetcher that surpasses the performance of all previous works, but the issue with prodigy is untimely and inaccurate prefetches. Fig. 3.1 shows that almost 40% of prefetched data gets evicted before being demanded. Working on those incorrect or late prefetches provides the opportunity to improve the performance of the prodigy.

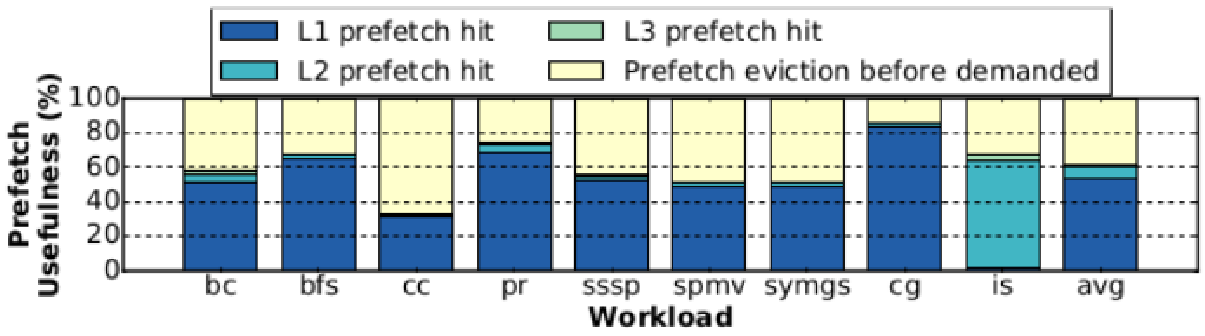


Figure 3.1: Hit location on Prefetched data for Prodigy

Along with incorrect and early evictions, the low use of L2 and the last-level cache are also observed. It is seen that prefetched data rarely gets a hit on both levels. We aim to solve both issues, to increase hit in lower level caches and to reduce the useless prefetching from happening.

As discussed in the above section, Prodigy faces the problem of untimely and inaccurate prefetches. By untimely, it is meant that a cache block was prefetched and it resided in the cache. But during that duration, access to it came only after its eviction of it. This issue could be solved if prefetching was made less aggressive. Inaccurate prefetches were caused due to misprediction by Prodigy and need to be resolved.

Prodigy is a graph prefetcher that always starts prefetching from the first list in the DIG, work queue in case of BFS. The first list is represented

by a trigger edge which is a self-loop over it. The problem with such an approach is that prefetching directly into the vertex list or edge list is not possible. In the case when access to a large degree vertex comes it will have a huge number of neighbors and property data along with them. Even in such cases, Prodigy starts prefetching from a new vertex in the work queue.
In short Prodigy has mainly two issues:
-Useless prefetches.
-Low utilisation of L2 and LLC caches.

3.2 Proposed Idea

Demand access of a vertex with a large degree causes such high evictions. The huge number of edge and property data of neighbors will be accessed causing the eviction of prefetched data. Along with the hypothesis, it is understood that the term large is an ambiguous term, the experiments below will try to find out an approximation for the large degree. The above hypothesis emphasizes the presence of large degree nodes of the graph and access to their data causes performance loss in Prodigy. Following are the ideas proposed that can be modified in the Prodigy prefetcher to get even better performance:

- Stop prefetching next node data when demand accesses are for high-degree nodes.
- Prefetch edge or property data for high-degree nodes(current) rather than prefetching every data for the next node to be processed.
- Perform experiments to bypass the high-degree nodes by finding a threshold value.
- Implement the analyses above technique across the cache hierarchy to find which cache level will give the best performance for prefetched data.
- Check the reuse number of each node and prefetch into the register directly if greater than a threshold.

Chapter 4

Future work

The final aim of this work is to reduce the memory latency in graph applications through graph prefetchers, thereby increasing the performance of the system.

Future work includes the following:

- To modify the prodigy code and make it compatible with SniperSim.
- Run experiments and analyse the current work in order to find where the issue persists in cache hierarchy.
- Modify the code to calculate hit percentage for varying degree of nodes.
- Identify by running simulations, in which cache level to prefetch and which data structures to prefetch.
- Analyse the effect of prefetching distance on performance.

Chapter 5

References

1. N. Talati et al., ”Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design” HPCA 2021.

2. S. Ainsworth et al., ”Graph Prefetching Using Data Structure Knowledge” ICS 2016.

3. A. Basak et al., ”Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads” HPCA 2019.

4. X. Yu et al., ”IMP: Indirect memory prefetcher” MICRO 2015.

5. C. Zhang et al., ”RnR: A Software-Assisted Record-and-Replay Hardware Prefetcher” MICRO 2020.

6. E. Bhatia et al., Enhancing Signature Path Prefetching with Perceptron Prefetch Filtering, HPCA 2016

7. A. Asgari et al., A Case for Multi-Page Multi-Layer Perceptron Prefetcher 2021

8. J. Kim et al., Lookahead Prefetching with Signature Path 2016

9. P. Michaud et al., Best-Offset Hardware Prefetching 2016

10. M. Bakhshalipou et al., Bingo Spatial Data Prefetcher, HPCA 2019

11. J.Kim et al., Path confidence based lookahead prefetching, MICRO 2016

12. Balaji, Vignesh, et al. ”P-opt: Practical optimal cache replacement for graph analytics.” 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2021.

13. S. Beamer, K. Asanović, and D. Patterson, “The gap benchmark suite,” 2017.