A Data-aware Prefetcher for Accelerating Graph Analytics

M.Tech Project Stage 1 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
October 3

Acknowledgement

I express my gratitude to my guide Prof. Virendra Singh for providing me the opportunity to work on this topic.

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. Due to the randomness in the connectivity of entities, it becomes difficult to capture the locality of reference even by the cache hierarchy in the modern computing systems. 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 tries to find a new prefetcher that can give us a better performance than the existing state-of-the-art prefetcher that have been developed so far. 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 L2 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 a social network, e-commerce recommendation, and bio-informatic, network analysis, path-planning, machine learning and others. It has lots of potential 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 due to irregular memory access patterns placed in by representations used to store graphs. Compressed sparse row(CSR) format is one such way to store graphs which requires multiple in-directions to get to required data, for ex. 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 change in address often called stride [8], [9], [10], [11] they perform poorly for irregular memory patterns. Some hardware and software prefetchers use machine learn- ing to predict the upcoming address requirements[6], [7]. These prefetchers rely on complex machine learning algorithms and are currently not practical.

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 which uses software for finding out the address bounds of lists of the graph and hardware support to make prefetching decisions. Droplet, Prodigy are some of the prefetcher techniques in recent times. Works like Prodigy[1], a state-of-the-art L1-D prefetcher that 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 list to prefetch.

1.1 Background

1.1.1 Cache Prefetching

Cache prefetching is a technique to improve the performance of the system by hiding the latency of main memory 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.

The performance of a prefetcher is analyzed by calculating these Metrics:

- **Coverage:** Coverage tells the effectiveness of the prefetcher for 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 measure of how early required block is prefetched into the cache before it is referenced by processor.

1.1.2 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 source vertex u to destination vertex v whereas in undirected

graph it denotes an edge between vertices 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.

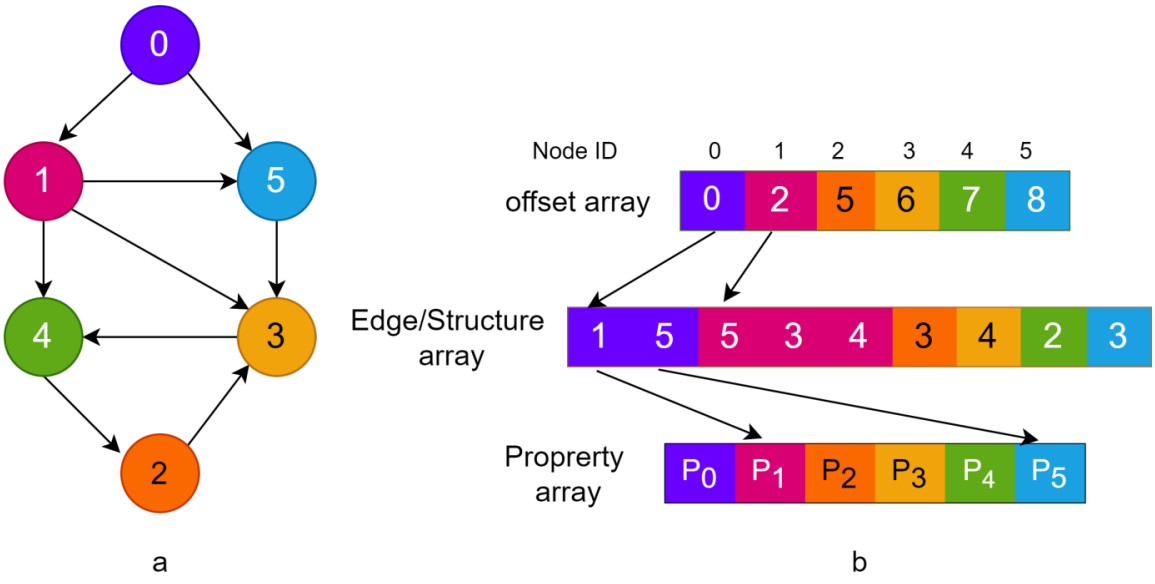


Figure 1.1: Compressed Sparse Row (CSR) graph using in-edges to a vertex

- **Structure/Edge/Neighbour array:** The edge array keeps the neighbours of each vertex in a contiguous 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.

1.1.3 Adjacency Matrix

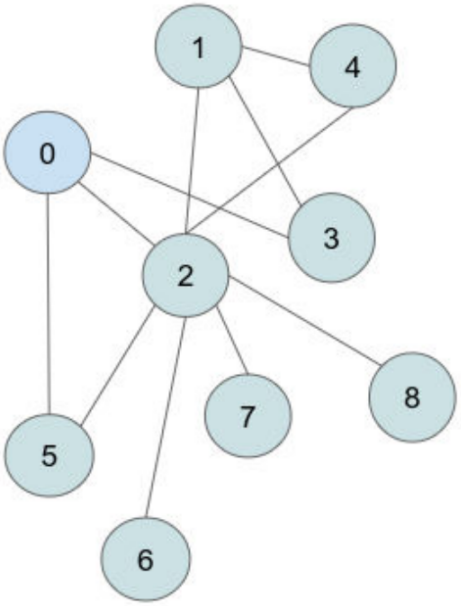
Both of these representations are common in storing graphs. Fig.1.2 shows a graph with 8 nodes and 10 edges. Adjacency matrix for such will be a matrix of size NxN where the number of vertices in the graph are N. The element at row x and column y represents the weight of an edge between the nodes x and 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 $O(N)^2$.

1.1.4 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 space complexity is $O(E)$, where E is the number of edges and V is the number of vertices in the graph. 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.

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



	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

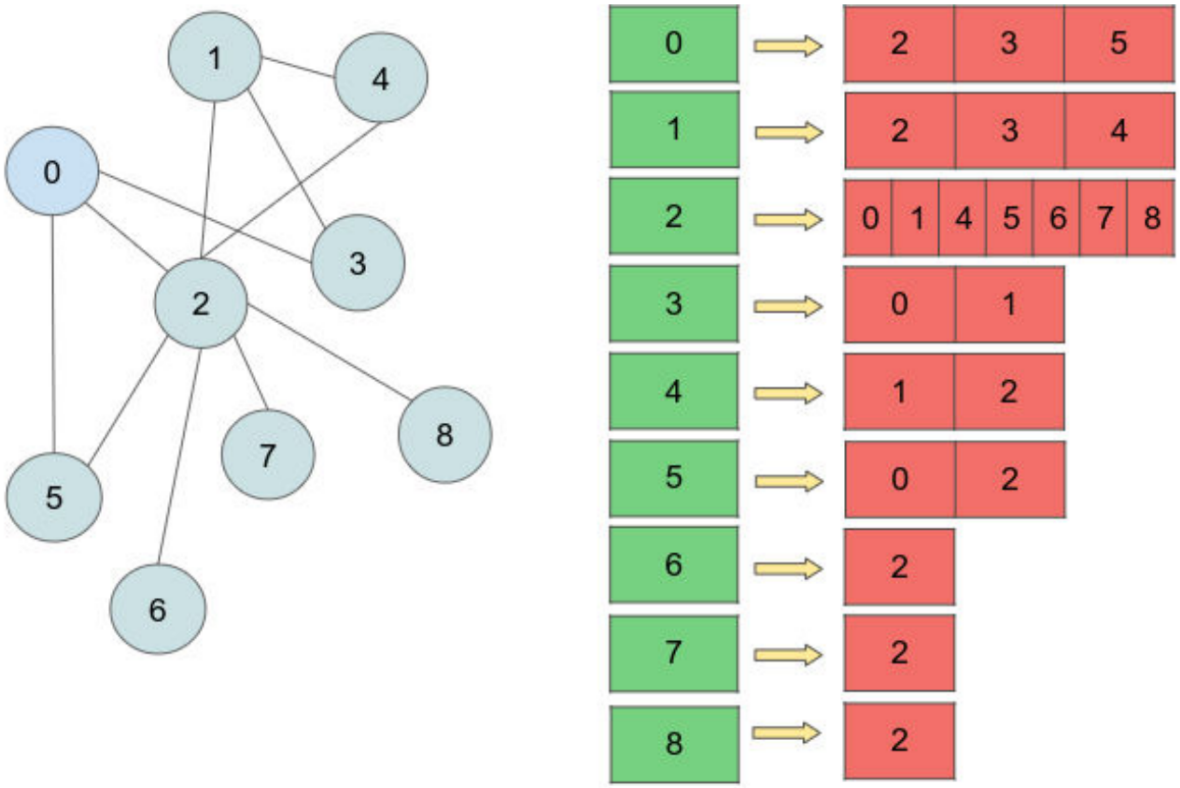


Figure 1.3: Adjacency List

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.

1.1.5 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):** It is a graph traversal algorithm. It 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. All its neighbors are pushed inside the queue, one by one a node is popped and its neighbours are again pushed inside. 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 page rank 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.

Chapter 2

Literature Review

2.1 P-OPT: Practical Optimal Cache Replacement for Graph Analytics

The main insight of this work [12] is that for graph applications, the transpose of a graph succinctly represents the next references of all vertices in a graph execution; enabling an efficient emulation of Belady's MIN replacement policy. In this work, P-OPT is proposed which is an architecture solution that uses a specialized compressed representation of a transpose's next reference information to enable a practical implementation of Belady's MIN replacement policy.

The paper shows that state-of-the-art cache replacement policies are ineffective for graph processing. It shows that guiding replacement based on graph structure in T-OPT allows emulating Belady's MIN policy without oracular knowledge of all future accesses. Fig 2.1 shows how a graph's transpose can be used to emulate optimal replacement technique. It shows two scenarios A and B with a 2-way set-associative set cache. It shows how replacement decisions will be made on a cache miss.

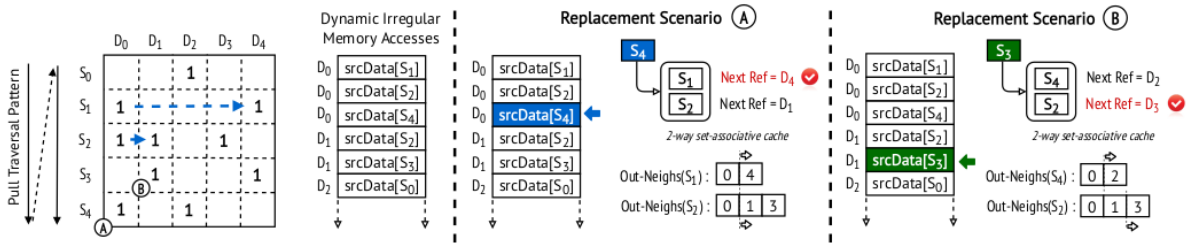


Figure 2.1: Using a graph's transpose to emulate OPT

However this approach has some issues in T-OPT; naively accessing the transpose imposes an untenable run time overhead and cache footprint. Hence to reduce the overheads associated with this, we use quantised Rereference information, and making a replacement decision by quantizing the graph's transpose. We model this using Rereference matrix. A Reref-

erence Matrix is a quantized encoding of a graph's transpose with dimensions *numCacheLines* * *numEpochs*. *numCacheLines* is the number of lines spanned by the irregularly accessed graph data, *numEpochs* is determined by the number of bits required to store quantized next references. Fig 2.2 shows the structure of Rereference matrix.

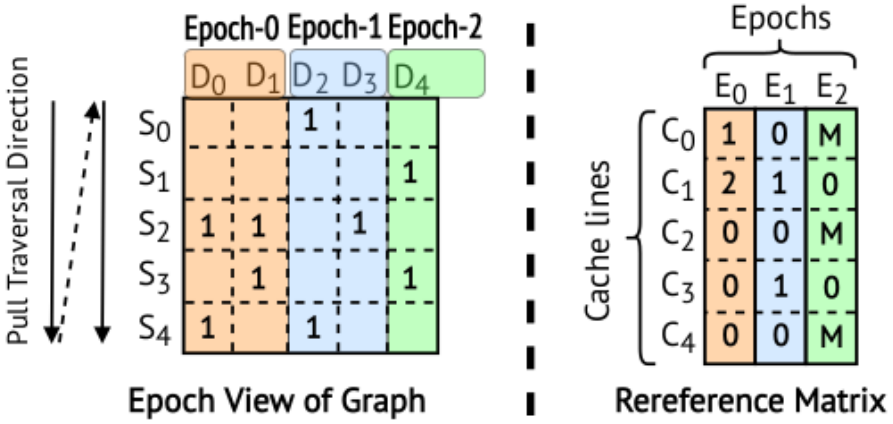


Figure 2.2: Reducing T-OPT overheads using the Rereference Matrix

It can be easily inferred that this quantisation will lead to some information loss. We call this quantisation loss. This paper proposes a solution to mitigate this loss. Fig 2.3 shows the modified design of rereference matix. Algorithm shown in the fig 2.4 shows how next references are calculated using reference matrix, along with quantisation loss mitigation.

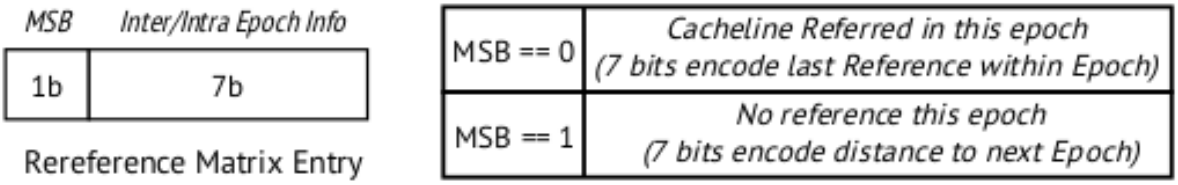


Figure 2.3: Modified Rereference Matrix design to avoid quan- tization loss

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.

Algorithm 2 Finding the next reference via Rereference Matrix

```
1: procedure FINDNEXTREF(clineID, currDstID)
2:     epochID ← currDstID/epochSize
3:     currEntry ← RerefMatrix[clineID][epochID]
4:     nextEntry ← RerefMatrix[clineID][epochID + 1]
5:     if currEntry[7] == 1 then
6:         return currEntry[6 : 0]
7:     else
8:         lastSubEpoch ← currEntry[6 : 0]
9:         epochStart ← epochID * epochSize
10:        epochOffset ← currDstID − epochStart
11:        currSubEpoch ← epochOffset/subEpochSize
12:        if currSubEpoch ≤ lastSubEpoch then
13:            return 0
14:        else
15:            if nextEntry[7] == 1 then
16:                return 1 + nextEntry[6 : 0]
17:            else
18:                return 1
```

Figure 2.4: Finding the next reference via Rereference Matrix

It was observed that LLC miss reductions drop with reducing frontier size in PageRank-Delta (partial frontiers) like applications and are even poor than LRU and DRRIP for non-sequential node processing applications like BFS (jumbled frontiers). Out of the above three Partial and Jumbled Frontiers were known to have a scope of improvement. By using a simple bitmap, for partial frontiers, LLC miss reductions can be improved.

2.2 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 Compressed sparse row(CSR) format to represent the 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], Droplet[3], Graph Prefetching using data structure knowledge[2], and IMP: Indirect Memory Prefetcher [4]. Software or operating system support enables the prefetcher to know the bounds of addresses of different lists of CSR format. For ex. BFS uses 4 data structures called work queue, offset list, edge list(neighbour array) and the property array. A prefetcher on know-

ing bounds or first and last address along with element size can directly access the elements of these lists by indexing into them. Droplet[2] is a data-aware L2 cache and LLC prefetcher which separately prefetches edge data and property data. Whenever access to a node comes its edge data neighbours are prefetched in a streaming behaviour. As the prefetched edge data reaches the memory controller of LLC the prefetched edge data is used to find out the addresses of their property data. Droplet assumes that whenever a node is accessed there is an urgent demand of its neighbours and their property data and therefore brings edge and property data of a node together in one prefetching sequence. IMP or indirect memory prefetcher is an extension of graph prefetcher which first classifies different memory access patterns one of which is what CSR base graphs show. That is to access node A of a list, (A[B[i]]) like data must be accessed where B is a data element of another list. To determine the value of A[B[i]] first B has to be accessed and then only A[B[i].

Graph Prefetching using Data Structure knowledge is an L1-D prefetching technique to prefetch edge and property data. It also acts as an inspiration for new designs like Prodigy. It requires compiler support to know the address bounds of different lists of the graph. On getting the address bounds it becomes easy for the prefetcher to access any element of the list on knowing their index. On seeing access to the work queue prefetching sequence starts which works in two modes.

- **Vertex-offset Mode:** This is a normal working mode of the prefetcher. On seeing access to the work queue a delta or a distance D is applied to the index of the currently accessed node in the work queue. From that element, a prefetching sequence starts where after work queue, vertex list offset and all neighbours of it from edge list are prefetched.

- **Large-vertex mode:** Whenever access to a large vertex comes prefetching is done into the neighbour nodes of the large vertex instead of again prefetching a sequence from the work queue.

The issue with the graph prefetcher design is that they only work with bfs related applications where there exists a work queue. But the salient feature of it is the breaking up of prefetching into different modes depending on the degree of the vertex being accessed.

2.3 Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design

Prodigy [1] is an L1-D prefetcher designed to work on graph-based data. Prodigy prefetches into all the data lists of CSR format. It requires compiler support to form a Data-Indirection Graph(DIG) which is used to store address bounds and the relation between different data lists. Data-Indirection Graph is a common data structure for the graphs of any type of representation. Prodigy requires the dataset/graph to be divided into multiple lists (like CSR format) to access the bounds of the lists and forms a Data-Indirection Graph. Each node of DIG stores the node_id for identification of a list, base_addr, capacity and data_size are used to find out address bounds(first and last address) of the list represented by the node. The edges of DIG represent the type of indirection between the lists, edges also tell the order of prefetching. An edge from vertex list to edge lists conveys that edge list elements are prefetched after the vertex list is prefetched. Each graph has at least one trigger node which has a self-loop over it called trigger edge. Whenever demand access comes to an address, prefetching start only from its trigger edge and continues till the leaf node of DIG.

Initial Address = base_addr
Final Address = base_addr+capacity data_size

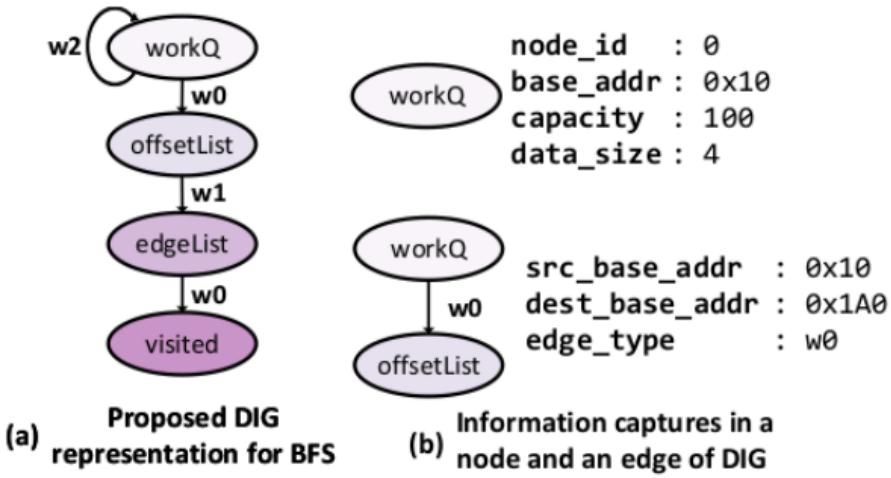


Figure 2.5: DIG representation and information capture in it's node and edge

2.3.1 Software support to form DIG

Prodigy requires software support to create Data In-direction Graph. Some functions have been provided to the user for registering various nodes and edges of DIG, and modification in the compiler are made to understand the functions.

```
params->RegisterNode(&(*scores.begin()), g.num_nodes(), 0);
params->RegisterNode(g.out_index_, g.num_nodes()+1, 1);
auto t_p1_0 = params->RegisterTrigEdge((NodeId)0, (NodeId)0, UpToOffset, SquashIfLarger);
auto t_p1_1 = params->RegisterTrigEdge((NodeId)1, (NodeId)1, UpToOffset, SquashIfLarger);
params->RegisterNode(g.in_index_, g.num_nodes()+1, 2);
params->RegisterNode(g.in_neighbors_, g.num_edges(), 3);
params->RegisterNode(&(*outgoing_contrib.begin()), g.num_nodes(), 4);
params->RegisterNode(&(*scores.begin()), g.num_nodes(), 5);
params->RegisterTravEdge((NodeId)2, (NodeId)3, PointerBounds_int32_t);
params->RegisterTravEdge((NodeId)3, (NodeId)4, BaseOffset_int32_t);
auto t_p2_0 = params->RegisterTrigEdge((NodeId)2, (NodeId)2, UpToOffset, SquashIfLarger);
auto t_p2_1 = params->RegisterTrigEdge((NodeId)2, (NodeId)5, UpToOffset, SquashIfLarger);
```

Figure 2.6: Various functions used to register DIG for pr application

- **RegisterNode:** This functions various nodes or the CSR lists of the DIG.
- **RegisterTravEdge:** This functions registers a traversing edge, an edge between different nodes to represent indirection.
- **RegisterTrigEdge:** This function applies a trigger edge over the given node.

It can be seen that in pr application a structure data list in_neighbours is registered with NodeId 3 and there exists a traversal edge between in_neighbours and a property data list outgoing_contribution.

2.3.2 Prefetching Mechanism

After storing the address bounds Prodigy can identify the source of the data whether it was from the vertex list, edge list or property list. When access to a work queue comes, Prodigy triggers a prefetching sequence. A prefetch look-ahead distance D over the trigger node is applied and prefetching is started, it then moves to the vertex list and prefetches all its neighbours and their data from the property list respectively. The prefetching candidates are stored in the prefetch request list, a buffer for keeping the prefetch requests.

The prefetch look-ahead distance D depends on the number of lists generated by the application which reduces with an increase in the number

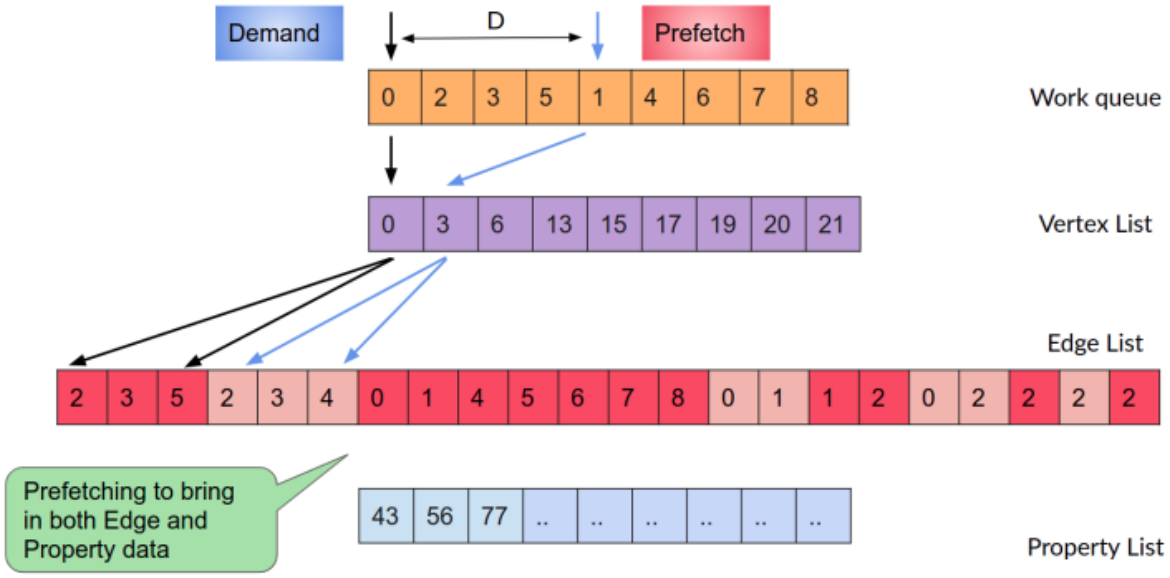


Figure 2.7: Prefetching Mechanism of Prodigy

of lists. Along with DIG, a PFHR (Prefetch history register) is maintained by Prodigy to keep track of prefetch requests to make prefetcher non-blocking. Non-blocking behaviour allows the prefetcher to keep on generating prefetch requests while the previous requests are being processed. As the prefetched data enters the cache it is read by the prodigy and further new requests are generated for its neighbours.

Chapter 3

Scope of Improvement and Proposed Idea

3.1 Issues with Prodigy

Prodigy is a state-of-the-art prefetcher which 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.

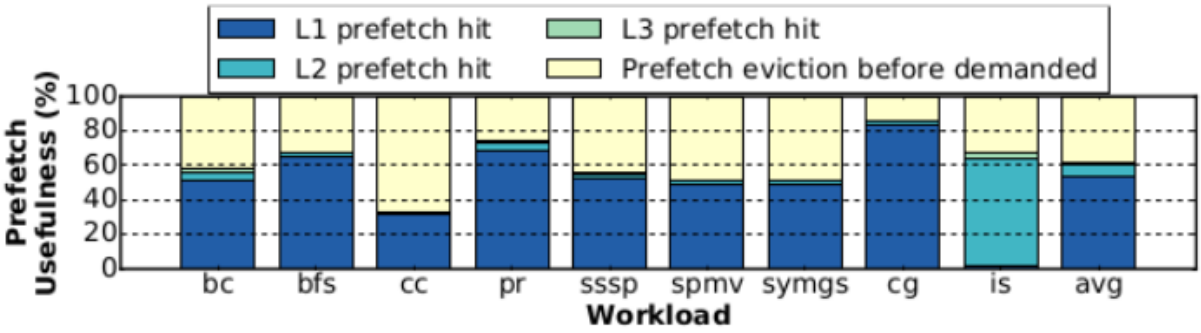


Figure 3.1: Location of hit on Prefetched data for Prodigy

Along with incorrect and early evictions, the low use of L2 and last-level cache is also observed. It is seen that prefetched data rarely gets a hit in both the levels. We aim to solve both the the 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, the access to it came only after the 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 which 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 the access to a large degree vertex comes it will have a huge number of neighbours and property data along with them. Even in such cases, Prodigy starts prefetching from a new vertex in the work queue.

3.2 Hypothesis

Demand access of a vertex with a large degree causes such high evictions. The huge number of edge and property data of neighbours 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 emphasises on presence of large degree nodes of the graph and access of 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:

- On accessing a large degree node Stop prefetching, stop eviction of prefetched data. This will Reduce cache pollution as instead of useless prefetch data useful demand accessed data will be present.
- To vary the range of degree of nodes to find the optimum threshold point, above which to stop prefetching.
- Implement the analyse above technique across the cache hierarchy to find which cache level will give best performance for prefetched data.

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.

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.