Improving Graph Workload Performance by Rearranging the CSR Memory Layout

Ryan Torok
Department of Computer Science
The University of Texas at Austin
Austin, Texas 78712, USA
ryan.torok@utexas.edu

December 14, 2020

Abstract

We propose *Neighbor Offset Extrapolation*, a hardware prefetching technique which aims to reduce memory latency in static graph workloads using the Compressed Sparse Row (CSR) memory layout by improving the spatial locality between nodes' IDs and neighbor lists. We extend this concept to SNOE, a software implementation which greatly improves the spatial locality of the neighbor lists with no prefetching. We also propose *Merged CSR*, an alternative memory layout which aims to obtain the same latency saving properties without the sorting requirements NOE has, while requiring little or no additional space over the existing CSR layout. We evaluate both techniques using the GAP benchmark suite with graphs on the order of 1,000,000 nodes. We find that Neighbor Offset Extrapolation is able to calculate the address of a node's neighbor list ahead of time with 100% accuracy in our graphs using less than 18K of metadata cache space. Using the Linux `perf` tool, we discover SNOE achieves a 20% speedup on average across our benchmarks versus CSR. We also find that Merged CSR obtains a 72% speedup in the triangle counting benchmark because of its traversal-heavy main loop. Other benchmarks also see gains: we see a 13% speedup in single-source shortest path and a 5% speedup in breadth-first search. Other benchmarks whose main loop iterates through nodes in memory order see performance losses with Merged CSR, but believe the introduction of an aggressive next-line prefetcher can work in tandem with Merged CSR to obtain even greater performance gains in these cases.

1 Introduction

Static graph workloads typically exhibit poor performance because of their pointer chasing behavior and poor spatial locality. Previous research has shown results for hardware prefetchers which specifically target graphs. Most of this work has focused on *pointer fetching* prefetchers that scour loaded cache lines for values which look like memory addresses, and issue prefetches for that memory. Unfortunately, pointer fetching prefetchers often issue many useless prefetches, because the software may not actually dereference every pointer in the cache line. In addition, the pointer chasing properties of graphs make even successful prefetches from pointer fetching schemes save little execution time.

Furthermore, the memory path taken in a graph workload changes based on the algorithm's inputs even within a single graph. Consider the `find()` operation in a Binary Search Tree shown in Figure 1 as an example. The path taken in the graph by `find(10)` (left) and `find(30)` (right) are completely different, and due to the poor spatial locality, extrememly challenging to predict ahead of time. These limitations place

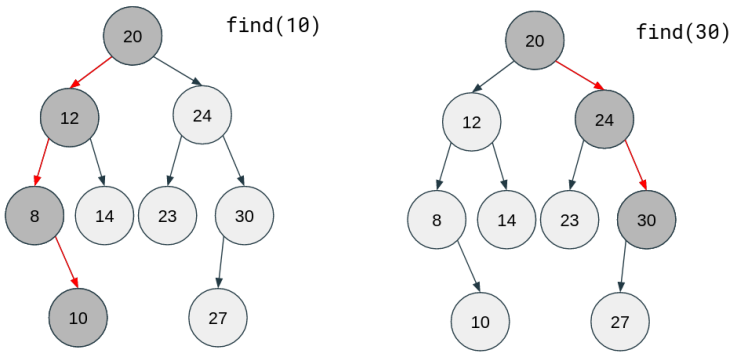

Figure 1: The traversals of two `find()` operations in a binary search tree. This demonstrates how the access pattern differs among different inputs, even if the graph is fixed.

significant restrictions on our ability to prefetch ahead of the pointer chasing in graphs. Our solutions in this paper achieve performance gains by reducing these restrictions through improving spatial locality.

We introduce two solutions for static graph workloads, which achieve performance gains by improving the spatial locality between a node's ID and the location of its neighbor list. The first of these techniques, *Neighbor Offset Extrapolation*, exploits ordering properties in a graph's structure that allow us to accurately calculate the address of each node's neighbor list, or list of nodes which are connected by an edge from our node, in a graph sorted in degree-order, without needing to load its address from memory first. NOE takes advantage of this information by issuing prefetches for the next neighbor list ahead of time. We also demonstrate SNOE, a software implementation of NOE which improves the spatial locality in finding neighbor lists in the same manner, and achieves significant performance improvements even with no prefetcher available. We also develop an alternative memory layout for graphs called *Merged CSR*, which organizes the graph's structural information in a way that allows us to eliminate the same cache misses NOE saves entirely in a software solution, without requiring any custom hardware or needing to sort the graph.

We evaluate the NOE using the GAP benchmark suite [1], on graphs on the order of 1 million nodes and 4 billion edges. By simulating the metadata caching process, we show that NOE can calculate the correct address of every node's neighbor list by storing less than 18K of metadata in the cache. This demonstrates that NOE is a practical to implement on hardware. In the future, we would like to implement NOE in a hardware simulator to obtain data on its timeliness and explore methods to reduce its over-fetch rate. Using the GAP benchmark suite [1] and the Linux `perf` tool on the same set of graphs, we find that SNOE achieves a 20% average speedup across our benchmarks. We also find that Merged CSR achieves a 73% speedup in the triangle counting benchmark, as well as a 13% speedup in single-source shortest path, and a 5% speedup in breadth-first search. Three of our benchmarks see performance losses, however, and analysis of the memory latency by function shows that Merged CSR performs particularly poorly on the benchmarks because their main loop iterates through a graph's nodes in memory order. We believe that the introduction of a basic prefetcher will alleviate this performance issue.

Both of these solutions rely on the insight that locating the list of a node's neighbors is much easier than guessing which neighbor nodes will be visted, and in what order. Because the neighbor lists in CSR's second array exhibit good spacial locality relative to the node IDs, NOE is able to calculate the address of a node's neighbor list given only its ID to issue accurate prefetches, without depending on the challenging-to-predict load to fetch the list's address that is required with no prefetching. Merged CSR achieves reliable locality in the neighbor lists by instead fusing the two CSR arrays and reassigning the nodes' IDs in a way that allows the software to know the location of a node's neighbor list without performing a separate load first.

This paper's primary contribution is the advancement of existing solutions for the longstanding problem of memory latency reduction in static graph workloads through the introduction of two new techniques to improve spatial locality in graphs. We demonstrate the feasibility of the Neighbor Offset Extrapolation, and in the future we would like to run experiments on the prefetcher to gather data on its performance and analyze the timeliness of its prefetches. We also introduce SNOE, software implementation of NOE, which we demonstrate achieves a 20% average speedup across our benchmarks in large, dense graphs. Finally, this paper also brings forth a significant strategy shift in optimizing graph applications. That is, we demonstrate the concept of using software solutions to improve spatial locality in graphs without the aid of prefetching.

Our work poses many interesting questions and insights we did not have time to explore. Firstly, how does the performance of the NOE hardware prefetcher compare to that of SNOE? Our data shows that even with the improved spatial locality, the NOE post cache still contributes to 21% of the memory latency in our benchmarks because of frequent cache evictions. We suspect the NOE prefetcher could improve performance even further, since its metadata is stored in a dedicated cache and is never evicted. Secondly, we are interested in exploring different methods for handling very high-degree nodes, whose neighbor lists are too long to efficiently prefetch all at once. We suppose that placing a limit on the number of cachelines to prefetch initially and then loading the remainder dynamically using a next-line prefetcher may be a suitable solution. Thirdly, how does Merged CSR improve with the introduction of a prefetcher? Our results suggest that Merged CSR performs particularly poorly in benchmarks whose main loop traverses the graph's nodes in memory order. With the introduction of a basic next-line or stream [9] prefetcher we expect to be able to mitigate this issue. Finally, we would also like to explore the interaction between hardware and software solutions more generally, and explore the possibility for the two to work together to achieve even greater performance gains.

2 Background

We now describe in detail how the CSR memory layout is used for static graphs, and how we can make use of locality of the neighbor lists relative to the nodes to optimize their memory performance.

Compressed Sparse Row (CSR) is a sparse matrix structure used for many applications outside graph workloads, but in this paper we will focus on its use for graphs. CSR stores a graph's structure using two arrays. The first array has length equal to the number of nodes in the graph. Each node N is given an ID, I_N, from 0 to the node count - 1. To indicate which nodes can be reached via an edge from N, CSR stores a pointer in its first array at index I_N, referencing a particular index, say J_N, in its second array. Beginning at index J_N, the second array stores a sequence of node IDs which are connected by an edge from node N, also known as the *neighbor list*. To know the length of this list, we can simply take the difference between the I_Nth and I_{N+1}st values in the first array, since the I_{N+1}st value represents the start of the neighbor list for the $N+1$st node. Thus, the second array, with all its separate neighbor lists put together, has length equal to the number of edges in the graph. Figure 2 shows a graph data structure, along with its CSR equivalent.

A common step in graph applications is an *edge traversal*; that is, reading a node's neighbor list to find out which nodes can be immediately reached from that node, and then performing some computation on the neighboring nodes. Importantly, it takes two memory accesses to find the neighbors of a node, given its ID I_N. First, we have to load the address J_N of that node's neighbor list from the first CSR array at index I_N, and then load the N's neighbor list from the second array at index J_N. These memory accesses are an example of *dependent loads*, because they cannot be parallelized by our architecture's memory controller. This is because we need to load the value J_N from the first array before we even know what address to load in the second array. Because the second load's address depends on data obtained in the first load, any memory latency incurred by the first loads will delay the second load as well. Performing several edge traversals in a row across several nodes creates long chains of dependent loads, also known as *pointer chasing*. Because dependent

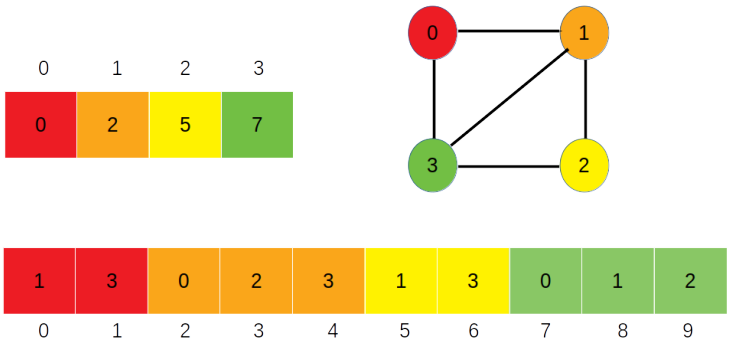

Figure 2: Example of a graph data structure (upper right), and its CSR memory layout equivalent. The numbers inside each node in the graph structure represent the node's ID in CSR.

loads must occur sequentially and cannot be parallelized by hardware, these chains are particularly harmful to performance, since a cache miss anywhere in the chain can stall the entire chain until its data returns.

However, CSR does allow some loads to be parallelized. Once we obtain the ID of a node, we can load the address of that node's neighbor list by indexing into CSR's first array using the node ID. However, graph workloads also store *internal node data*, such as the node values in a binary search tree, potential values in A* [18], or usernames in the graph of social media followers. This internal data is typically stored in arrays indexed by the node ID, so if we need to access this data in order to know which edge to traverse next, loading this data typically does not hinder performance even on a cache miss, since its load can be parallelized with the load for the neighbor list address in CSR's first array.

At several points in this paper, we will utilize an abstraction of *DRAM latency cycles*, or time units equal to the average amount of time required for a value to be loaded from main memory after a cache miss. For example, we say a chain of five dependent loads which all miss in the cache takes five DRAM latency cycles to complete, because every one of the loads must wait that length of time for its value to return before the next load can be issued. For simplicity, under this abstraction we assume all cache hits suffer no latency regardless of which level of cache the hit occured in, and that all computation operating on data which hits in the cache occurs instantaneously. For these reasons, this abstraction will not be too honest as far as analyzing prefetch timeliness in practice, but it will be extremely useful for reasoning about optimizing dependent loads. As we will see in section 4, the abstraction shows us that prefetching ahead of the pointer chasing is much more feasible in the second CSR array than in the first.

As we discussed in section 1, most previous prefetching work in graph workloads has focused on *pointer fetching*; that is, scouring memory loaded into the cache and issuing prefetches for values in the cache line which look like memory addresses. However, as we discuss in more detail in section 3, each of these methods fall short for two critical reasons:

1. The nodes in graphs tend to exhibit poor spatial locality, meaning nodes close together in memory do not tend to be close in the data structure, and vice versa.

2. Many graph workloads' traversals are dependent on data stored in the nodes, creating timeliness problems because the correct neighbor cannot be decided until the node is loaded.

The two techniques we discuss in this paper, *Neighbor Offset Extrapolation* (NOE) and *Merged CSR*,

each address both of these issues. NOE achieves this by taking advantage of the ordering properties in CSR's second array to achieve accurate predictions of the address of a node's neighbor given only its ID. On the other hand, Merged CSR sidesteps this locality issue entirely by rearranging memory so that neighbors can occupy a cache line which can be efficiently loaded knowing only the node's ID. We present a time-space tradeoff among several methods of loading internal node data which have varying applications depending on how each graph algorithm utilizes the data.

3 Related Work

Several previous publications have addressed graph memory latency, mostly focusing on hardware prefetchers. The first graph-specific prefetcher was the Irregular Memory Prefetcher [2], which took advantage of the fact that in most irregular memory access patterns based off the form $A[B[i]]$, including CSR, the innermost access will exhibit good spatial locality because the i's will occur in a loop. IMP leverages this by caching sectors of B it needed to read anyway, and use them to predict future accesses to $A[B[i]]$. In a graph, A would be the second CSR array containing the neighbors, and B would be first CSR array containing the each node's neighbor index. Accordingly, a successful prefetch on $A[B[i]]$ would be able to traverse from one node's neighbor list to the next node's neighbor list in a single DRAM latency cycle, not two, as is needed with no prefetching. The paper also proposed utilizing partial cache lines to save cache space, because in nearly all cases only one or a few values were needed from each value fetched. Unfortunately, this scheme is limited in scope, as the poor spatial locality in graphs caused the hit rate in the second CSR array to be low.

Three later papers improved on this concept, the first proposing the Programmable Prefetcher [3], which utilizes small CPUs to run simple code segments provided by the software to prefetch irregular access patterns. The second paper [4] included some excellent profiling work charactering the memory latency bottlenecks in graph workloads. The authors observed that load-load dependencies are the main performance hindrance, and the first load of these chains, which they call the *producer* load, is almost always a load on the neighbor lists in the second CSR array, which they call a *structural* load. Using their observations, they proposed DROPLET, a prefetcher which includes a *data-aware* streamer which aims to reduce overfetching versus a conventional pointer-fetching prefetcher, by marking with a flag bit which memory corresponds to graph structure data. Furthermore, the Record-and-Replay (RnR) prefetcher [5] stores sequences of irregular memory offsets, and uses software assistance to know when to replay the irregular streams, and at what base address.

4 Motivation

We motivate the techniques used in NOE and Merged CSR by analyzing why previous graph prefetching schemes have not performed as well on graphs as they have on other irregular data structures.

As mentioned in section 3, IMP [2] is limited in its coverage possiblity, because only nodes that are neighbors of previously accessed nodes have a chance to be prefetched, and the DROPLET [4] authors show the neighbor lists exhibit longer reuse distances than the nodes' internal data or the pointers to neighbors. This is especially problematic if a newer cache eviction policy is used for the neighbor cache line buffer, such as RRIP [6], SHiP [7], or Hawkeye [8], which all decide what to evict based on expected time to future access. These cache schemes all elect to keep data they predict will be accessed sooner, and accurate performace of these schemes would lead to few successful prefetches using old cached neighbor lists. In addition, because of graphs' poor spatial locality, IMP cannot reliably obtain performance improvements by caching memory-sequential neighbor lists, because it cannot make any guarantees the two nodes are near each other in terms of access time.

The scope of both the Programmable Prefetcher [3] and DROPLET [4] are limited because of the data dependencies in most graph algorithms. Although the Programmable Prefetcher allows user-supplied code to predict prefetches, it does not provide much of a time save for prefetching neighbors, because most workloads rely on internal data to decide which neighbor to take, which is not possible to predict at the time the code is submitted. Similarly, DROPLET [4], although it is able to filter out some non-structural data from its pointer prefetches, there is a fundamental lower bound on the over-fetch rate that can be achieved, because the data that determines which neighbor will be taken arrives one DRAM latency cycle later than the one the prefetches must be issued to save time.

RnR [5] also suffers from this issue. Although it performs better than IMP [2], Programmable Prefetcher [3], or DROPLET [4] on graphs, RnR's [5] performance on graphs falls significantly short of its excellent gains it achieves on matrix applications. This is because matrices tend to not suffer from spatial locality issues or data dependencies. In the typical row-major format used by all major low-level programming languages, values next to each other in a row within a matrix data structure will be in consecutive memory locations, and values next to each other in a column will be the same stride apart in memory, and that stride is known at load-time. For this reason, an operation like matrix multiplication can be optimized using a general stride prefetcher, like the Best-Offset Hardware Prefetcher [10]. Of course, RnR [5] can help immensely with more irregular matrix operations like Principal Component Analysis and Gaussian Elimination, which involve more irregular access patterns than a linear stream.

Importantly though, these irregular access patterns are highly repeatable among similarly-sizsed matrices, as the access patterns generally do not depend on the input data. Consider standard matrix multiplication as an example. Calculating the value which ends up in the upper-leftmost cell of the answer always involves streaming through the first row of the first argument and the first column of the second argument, regardless of what actual values appear there. Contrast this with the graph search algorithm A* [18], where, at any step, the next neighbor to traverse to and, therefore, the next memory to be loaded, depend on how a potential value stored with a node compares to that of the target node. Therefore, streams that RnR records will not be reliably replayable because the traversal is dynamically decided based on input parameters (in the case of A* [18], the potential value of the target node).

Even worse, these internal data dependencies seem to be a fundamental feature of graph algorithms. Consider the `find()` operation on a Binary-Search Tree formatted as a CSR graph. Just like A* [18], this operation decides to traverse to the left or right child based on a comparison between an internal node value and a hyperparameter value that is decided for each run and cannot be known at the time an RnR stream is recorded. Imagine if instead BST find did not involve this data dependency. The only other option would be that the traversal was identical for all `find()` calls on that particular tree. However, that would certainly mean we would always arrive at the same node every time, and we would not need to do the traversal in the first place! This nonsense operation is analagous to reading every element in a matrix just to find out what the lower-rightmost element is. Graph algorithms must exhibit these data dependencies to do useful work because they express their decisions in their paths through a static data structure rather than in the values written to an entirely new data structure. These highly irregular access patterns are also the primary reason past prefetching work in generalized server workloads targeting linear and strided streams [9] [10] or memory history [11] [12] [13] [14] [15] [16] [17] has seen poor performance in graph applications.

These issues highlight how we need to change our strategy with regards to determining future memory accesses. That is, instead of aggressively fetching pointers in the cache lines we load, we need to be able to make use of locality properties in graphs in order issue to reach data *ahead* of the pointer chasing, before we see a concrete pointer to the data. Although we have established that the *nodes* within a CSR graph tend to exhibit poor spacial locality, we can take advantage of how the neighbor lists are stored. A single node's neighbors are always stored in a contiguous region of memory, and the lists are always stored in the same order in the second CSR array as the nodes are in the first array. As we will see in section 4.1, these properties provide us a critical opportunity to accurately run one DRAM latency cycle ahead of the pointer chasing.

4.1 What does CSR make easy (or hard) to prefetch?

Poor spatial locality of nodes does create barriers to prefetching graphs by running ahead of the pointer chasing. The largest barrier is the difficulty of determining the the next node in the traversal. To issue timely prefetches that save a DRAM latency cycle in this manner, we must request the prefetch at the time we know the current node's ID and the address of its neighbor list, but not the IDs of the neighbors themselves. To determinte if this is possible, observe that in order to prefetch the next node, we need to know two pieces of information:

1. Which neighbor(s) will be fetched from the current node?
2. What is the ID of each of the neighbor nodes that will be fetched?

The Programmable Prefetcher [3] has demonstrated the ability to solve requirement 1 above by introducing a hardware-software interface that allows small code snippets to run on dedicated CPUs whose responsibility is to predict what neighbors to prefetch. Additionally, this technique could be expanded to process traversals with data dependencies as well, as the internal data belonging to the current node is available at the time we issue the prefetch. However, requirement 2 is much more problematic. Even if a scheme like the Programmable Prefetcher [3] were able to choose which neighbors will be accessed, it could only reasonably predict the *indices* of the neighbor list will be chosen, not the actual ID of the neighbor. Because of the poor spacial locality in graphs' nodes, predicting the ID of any of the chosen neighbors before receiving the current node's neighbor list from main memory seems very unlikely. This issue is the primary reason we do not consider running ahead of the pointer chasing by prefetching in the first CSR array to be feasible.

However, there is a different load we could attempt to optimize instead: the one going from a node neighbor pointer in CSR's first array to the actual neighbor list in the second array. To save a DRAM latency cycle in this case, we need to issue a prefetch for a node's neighbor list when we have just received the previous node's neighbor list from DRAM. This means we know the ID of the node whose neighbor list we need to fetch, but nothing else about it.

Even with this limited amount of information, we have a few advantages over prefetching in the first array:

1. For a graph traversal, even if we won't know what neighbor to visit next in a timely manner, we at least know we will need to load the neighbor list, and do not have to perform any dynamic calculation to determine what to prefetch, provided we can determine the address in a timely manner.
2. The location of the neighbor lists exhibit good spatial locality relative to the node IDs.

In other words, we almost always need the neighbor list after visiting a node, and CSR's second array always stores the neighbor lists in the same order the nodes are stored in the first array. Therefore, we should develop a technique which allows us to calculate the address of a node's neighbor list knowing only the node's ID. If we could achieve this, it would in fact save a DRAM latency cycle because we could issue a prefetch a node's neighbor list at the time we learn its ID from the previous node's neighbor list and receive it at the same time we receive the node neighbor pointer and internal data.

In this paper we introduce two techniques for eliminating this cache miss: *Neighbor Offset Extrapolation* and *Merged CSR*.

5 Neighbor Offset Extrapolation

Here, we introduce the high-level concepts for the Neighbor Offset Extrapolation (NOE) prefetcher. NOE is a hardware prefetching technique which fetches ahead of pointer chasing by taking advantage of additional

spacial locality properties we can achieve by sorting the graph's nodes in order by degree in order to calculate the location of a node's neighbor list knowing only the node's ID.

By making use of cache space to store metadata, NOE preprocesses the sorted graph and records every location the node degree changes, storing which node ID the change occurred on and what the actual degree of the nodes in that region are. We call each of these records a *post*, since it acts mildly like a signpost, letting the hardware know that the next nodes have a stated degree until another post tells it otherwise. We find in our evaluations, using graphs on the order of 1 million nodes and 4 billion edges, that a sorted graph can store all the posts using less than 18K of cache space. For a sense of scale, the L1 cache on a typical desktop CPU has a size of 16K per physical core. Given that [4] showed that the L2 cache is very poorly utilized in graph workloads, we expect this amount of metadata to cause few cache capacity issues.

Sorting the graph is necessary to eliminate a critical source of uncertainty in the locations of the neighbor lists we are prefetching. In the second CSR array, the staring index of the neighbors of a node is equal to the sum of the degrees of all the nodes coming before it in memory. Because the degrees of the nodes in an unsorted graph appear random, it is not possible to accurately calculate the offset for a given node in the unsorted graph, since the offset depends on the degrees of all the nodes coming before it. By sorting the graph in order by degree, we can get around this restriction, because the sorting the graph by degree produces sections of the graph where all nodes have the same degree. Because most practical graphs have the property that the vast majority of nodes have low degree, we will end up with long sequences of nodes that have no uncertainty in the start index of their neighbors, provided we record the region's starting node and the degree. Specifically, if we record that our sorted graph has a region of L nodes which all have degree D, beginning at starting node S that has neighbor start index O_S, we know the neighbor start index O_N of any node with ID N such that $S \leq N < N + L$ to be $O_N = O_S + D(N - S)$. Luckily, we find in experimental testing that the additional time required to sort the graph during preprocessing only increases the build time of our graphs by about 15-20%. Based on our results for NOE, this trade-off is entirely worth it for workloads running any more than a few rounds of a graph algorithm.

Knowing the actual degree of the node we are prefetching also gives an additional advantage: by knowing the degree of the nodes in each region along with the exact address at which our target node's neighbor list lies, we can exactly calculate the proper *lookahead* to with which to prefetch. We use the term *lookahead* to refer to the number of consecutive cache lines to fetch in order to ensure we load the entire neighbor list of the node we are loading [1]. This ensures we are always able to load the entire neighbor list regardless of its length, meaning we will not suffer additional cache misses resulting from not loading enough memory to store all the neighbors. We also avoid the opposite problem; that is, issuing useless prefetches for memory belonging to neighbors of nodes coming after us.

We experimentally calculated the cache size necessary to store all the necessary posts. Figure 4 shows our results. We find that on average, a cache size between 17 and 18 kilobytes was necessary to store all the necessary posts, including both the neighbor list location and the node degree. Comparing this to a few graphs with orders of magnitude fewer nodes than the ones shown, we also find that the cache size does not scale linearly with the number of nodes in the graph. This makes sense, because the number of degrees exhibited by the graph's nodes should not grow as fast as the node count itself, since we should expect the proportion of nodes with low degree to stay roughly the same as the node count increases. Observe that the size of a region does not affect the cache space necessary to describe it using posts. Whether a region where all nodes have degree 2 has 1,000 nodes or 1,000,000 nodes, we still only need one post at the beginning of the region for our calculation.

[1] Many previous papers in the prefetching domain have used the term *degree* to refer to how far ahead to prefetch, but we will refrain from using this terminology to avoid confusion with the term *degree* in the graph context, referring to a node's neighbor count.

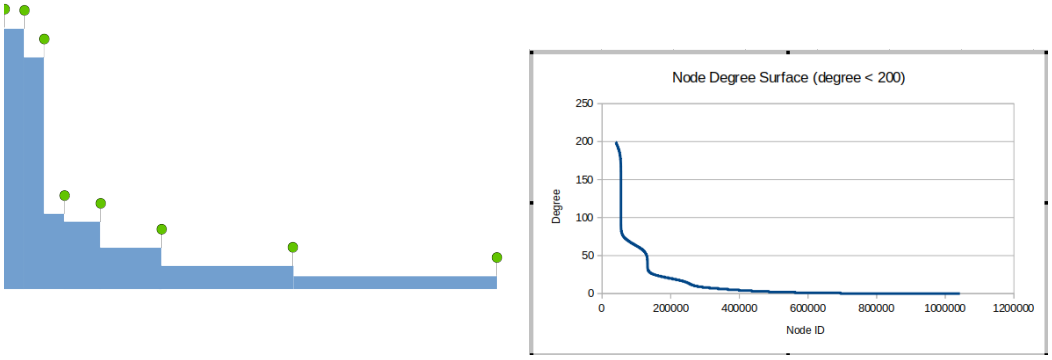

Figure 3: The left chart shows a high-level design Neighbor Offset Extrapolation, which stores metadata about where the degree changes, allowing us to calculate the location in the second CSR array of any node's neighbors, knowing only the node's ID. The right chart shows data for a real graph, indicating how most nodes have low degree. We excluded the less than 1% of nodes with degree ≥ 200 to make the chart easier to read.

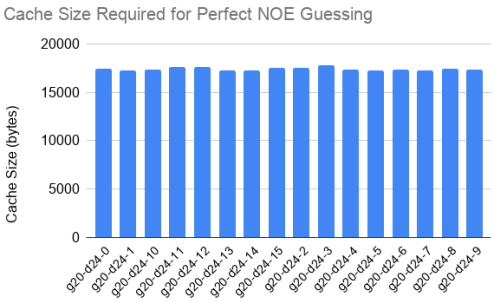

Figure 4: This chart shows the cache size in bytes required to store all NOE metadata, inserting a post at every degree transition of each sorted graph in our evaluation set.

5.1 NOE Architecture

Functionally, NOE requires a separate partition of cache space to store the posts. The post cache is stored as a sorted map, because we need to be able to find the post whose starting node ID is nearest to, but not greater than, a given node ID.

Using a dedicated instruction, the software alerts the hardware of the locations and index size of the two CSR arrays for the sorted graph, and the architecture automatically preprocesses the first array to record a post for each node ID where the degree changes, storing both the node ID and the degree.

Once the preprocesssing completes, the architecture tracks the memory access stream to search for any accesses within the second CSR array. When one occurs, NOE takes the Node ID at the location that was accessed and performs a lookup in the post cache to determine the post whose ID is nearest to, but not greater than, node ID the software accessed. Once we find this post, we perform the calcuation mentioned earlier to determine the location and size of the neighbor list for the accessed node ID. We then issue a prefetches to cover every cacheline required to fetch the entire neighbor list.

5.2 Effect of Graph Size on Metadata Size

Increasing the node count does in fact slowly increase the metadata size, however, because larger graphs present more opportunity for larger sets of degree values among high-degree nodes. When we tested two smaller graphs generated in the same manner as those in our previous experiments (with 32768 and 131072 nodes), we observed that the required cache sizes were 5120 and 7776 bytes, respectively. Based on this result, we suspect the relationship between node count and cache size is logarithmic, but we would need to perform further experiments with many differently sized graphs in order to rigorously show this.

This reasoning also shows that sparse graphs, which have fewer different degree values among their nodes, take less cache space than dense graphs. In particular, we would like to highlight an extreme example of this effect we observed while using two graphs which represent real-world road maps. The nature of these graphs makes it extremely rare to find nodes of high degree (try searching for a 126-way intersection the next time you are out in your car). Specifically, nearly all nodes in these graphs have degree less than five, meaning we can get away with an extremely small metadata cache for these graphs. Despite the fact the graphs have nearly 2 million nodes, nearly double that of the dense graphs shown in Figure 4, we determine that NOE for these graphs requires less than 128 bytes of metadata.

5.3 Using NOE as a Software Solution

We can also take advantage of the properties of the NOE cache in software to achieve a significant performance improvement even with no hardware prefetcher available. Here, we introduce Software NOE, or SNOE, which achieves a 20% average speedup over CSR across our benchmarks. SNOE works by replacing the first array of CSR with a NOE post cache to achieve better spatial locality and cache peformance when loading neighbor lists.

Recall that we expect sorted graphs to contain long series of consecutive nodes with the same degree, since most nodes have low degree. For this reason, the first array in CSR, which is responsible for enumerating the location and distance between the neighbor lists in the second array, will contain large regions that give little useful information. Specifically, in a long sequence of nodes with degree D, whose first node has its neighbor list begin at index N in the second array, the section of the first array responsible for those nodes will just contain the values N, $N+D$, $N+2D$, $N+3D$, ... , since every node in the region has a neighbor list of length D.

For this reason, we can greatly improve our performance by replacing the first CSR array with a software implementation of the NOE post cache. In our previous example, instead of storing the redundant values N, $N+D$, $N+2D$, $N+3D$, ..., we could instead just store the information that all the nodes beginning with the first node in the region have degree D, and the starting node has neighbor offset N. Unlike the first array of CSR, which suffers frequent cache evictions because it is too large to fit in the cache all at once, the NOE post cache is very compact, taking only 17-18K of space as we showed earlier, and thus we expect it to achieve much better cache performace. Each cache hit we get this way allows us to load a node's neighbor list in only one DRAM latency cycle instead of two, achieving the same optimization as when NOE is used as a hardware prefetcher.

5.4 Limitations of NOE

5.4.1 Limit on Number of Prefetched Cache Lines on Neighbor Trigger

In nodes with very high degree, we may encounter cases where it is infeasible in terms of memory bandwidth to immediately prefetch the entire neighbor list of the node we are traversing to. Because we observe that most of our benchmarks iterate through the neighbor lists they load in a loop, we plan in our future hardware implementation of NOE to place a hard limit on the number of cachelines fetched immediately when the

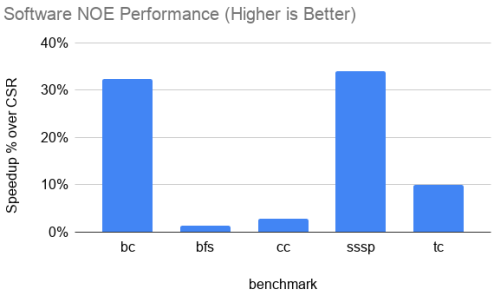

Figure 5: Performance Speedup achieved using SNOE across our benchmarks.

trigger occurs, and then load the rest as needed with a basic next-line or stream [9] prefetcher. We are also interested in exploring the idea of tracking the tempo of the loops through the neighbor lists and dynamically adjusting the number of cache lines fetched on a trigger in order to prefetch just enough lines to keep the prefetches timely. Such a solution may achieve the same performance benefits while reducing cache pollution and preventing early eviction of the prefetched neighbors.

5.4.2 Limitation in Invertible Directed Graphs

NOE is also unable to fully optimize directed graphs that need both the graph and its inverse; that is, the same graph with all edge directions reversed. Because nodes in these graphs can have different degrees in the standard and inverse graphs, sorting both by degree would make the IDs of the same node different in the two directions. For this reason, it is impossible to make the nodes appear in sorted order in both directions and maintain the same ID for each node across both, meaning only one direction can be optimized.

5.5 NOE Evaluation

We evaluate NOE on the GAP benchmark suite [1], using the Linux `perf` tool [19], on a standard notebook machine with an Intel Core i5 (8th generation) CPU with a 1.7GHz clock speed, 8 logical cores, 32K L1d cache, 32K L1i cache, 256K L2 cache, and 6144 L3 cache. Although Intel does not publish the type of prefetcher available in these cores, a bench test we performed showed there is likely no prefetcher present; the `perf` [19] tool showed a cache miss rate well over 90% when looping through a long array in order, which even a basic next-line prefetcher should be able to optimize. For these experiments, we utilized a variety of graphs. We use fifteen graphs on the orde of 1 million node and 4 billion edges, as well as the two road map graphs and two smaller automatically generated graphs mentioned in section 5.2.

Figure 5 shows the performance increase of the software implementation of NOE versus standard CSR across our benchmarks. It achieves a 34% speedup in `sssp` (single-source shortest path), a 33% speedup in `bc` (betweenness centrality), and a 10% speedup in `tc` (triangle counter), along with marginal gains in `bfs` (breadth-first search) and `cc` (connected components). The `pr` (PageRank) benchmark was excluded from our test because it uses invertible directed graphs and makes use of both directions equally. For this reason, we do not believe NOE will improve performance in this benchmark.

The gains we see are a direct result of achieving fewer cache misses when traversing edges. The minimal effect we see in breadth-first search and connected components can be attributed to excessive hardware cache misses in the post cache, which SNOE relies on achieving good cache performance in order to see a significant speedup. Figure 6 shows the fraction of the memory latency in each benchmark attributed to hardware cache

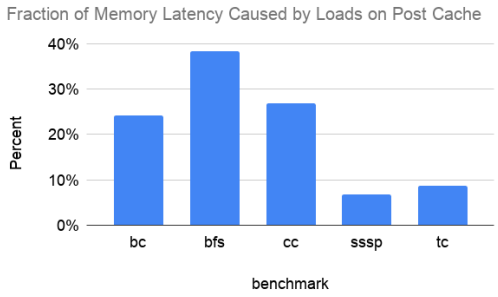

Figure 6: Fraction of each benchmark's total memory latency attributed to hardware cache missed in the post cache.

misses in the post cache. Breadth-first search and connected components see the worst hardware cache performance, with BFS specifically seeing 38% of its memory latency attributed to misses in the post cache. This is consistent with our earlier results, showing that poor hardware cache locality is indeed the cause of `bfs` and `cc` not achieving a significant speedup. We believe the higher-than-expected latency in the post cache is attributed to long reuse distances in CSR, which [4] determined occurs through profiling experiments. The high reuse distances cause the cache lines within the post cache to suffer frequent evictions from the hardware cache, which causes additional latency when the corresponding nodes are loaded again.

Although most of these rates are below that of CSR, whose first array accounts for between 30-45% of the memory latency across the same benchmarks, the frequent hardware cache misses in the post cache suggests we should see even better performance gains by implementing NOE as a hardware prefetcher. In the future, we would like to run the prefetcher in a hardware simulator, and compare the performance results with the benefits we see in SNOE.

6 Merged CSR

In Merged CSR, we take a completely different approach to reducing the latency of loading neighbor lists, which, unlike NOE, does not require sorting the graph first. Instead of storing metadata indicating the address of each neighbor list, we instead *merge* the two CSR arrays, such that the data for each node originally in both arrays is instead placed in one contiguous block. This allows all the data from the two CSR arrays to achieve good spatial locality. We then reassign the IDs of every node within the neighbor lists to reflect the index in the Merged CSR array at which that node begins.

The pointer from the first CSR array is repurposed to instead point to the next node in memory, since its original purpose, referencing its neighbors, is now unnecessary, since the neighbors always begin one index above the pointer. What we need to store, is some information about the degree of our node, or, equivalently, how far forward to go in memory before the next node begins. In CSR this was achieved by taking the difference between two consecutive neighbor offsets in the first array, which is no longer an option for us because the cells are no longer adjacent in Merged CSR.

Another way to think about how Merged CSR functions is that we make the neighbors faster to access by replacing the values in each neighbor list to point directly to the neighbor list of the neighboring node rather than indirectly referencing the neighbors through the first array. One tempting question to ask, therefore, is, "Is the first array in CSR necessary at all?", The answer to this question is more complicated than it appears.

As far as traversing the graph's edges is concerned, the answer is no. If we replace each node ID in the second CSR array with the index in that same second array of the neighbors of the node with that ID, we can

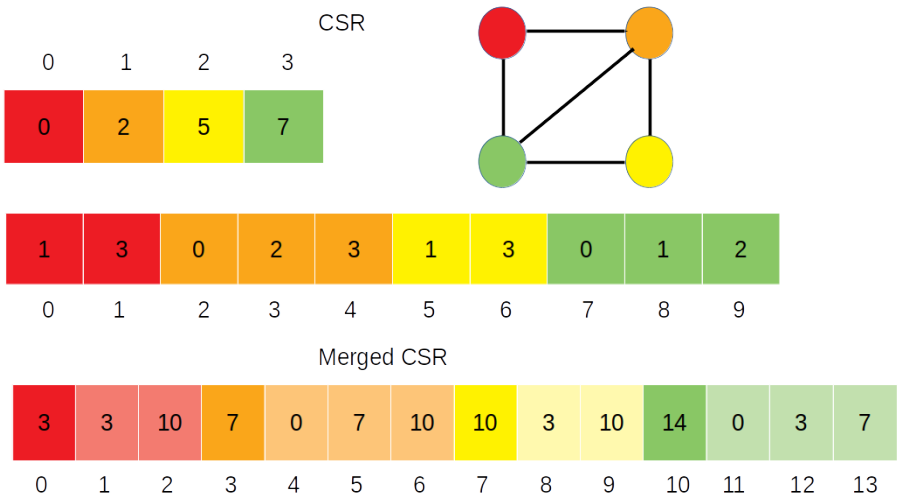

Figure 7: Example of converting the graph in Figure 2 from standard CSR to Merged CSR. Observe that the node IDs change in Merged CSR because they reflect the nodes' start indices in the Merged CSR array. Each node in Merged CSR has a single entry (full opacity) which references the start index of the next node, and the remaining entries contain the node IDs that the node is connected to by an edge.

in fact reach the next node's neighbors without any need for the first array. However, the internal node data is typically stored in arrays that are indexed by the node ID, and this behavior requires the node IDs to be consecutive values. If we allow the start index of a node in the single Merged CSR array to logically become that node ID, the IDs will no longer be consecutive. Now we have the problem of needing to store some information about how to find the node data. We propose several methods for doing this.

6.1 Locating Sequentially-Indexed Node Data

Since Merged CSR reassigns the nodes' IDs to be non-consecutive values, we now have an issue when needing to access internal node data stored in arrays indexed by a node's original ID it had in standard CSR, which we call the node's *sequential ID* (versus the *non-sequential ID* used to reference a node's position in the Merged CSR array). We now introduce several strategies for efficiently accessing data indexed by sequential IDs using Merged CSR.

6.1.1 The Lazy Approach

The simplest solution we could employ to handle the non-symmetric IDs is to just accept that our data will be stored in arrays with large gaps. The trouble with this method is that the data memory usage takes $O(V+E)$ space complexity rather than $O(V)$. Although the DRAM latency cycle analysis from earlier shows that we should be able to efficiently load the data even if it exhibits poor spatial locality, this method will be particularly heavy on the OS page allocator because the array requires so many blank indices that serve no purpose but to waste enough memory so the real data appears in the correct index. For very large graphs, we may also begin to see performance losses due to the increased amount of page swapping required to maintain the large memory overhead. In addition, the amount of time to initialize the memory may also significantly increase, hurting performance even further.

6.1.2 Sparse Array Notation

Typically, the problem of storing arrays with many blank indices is solved by using a dynamically allocated *sparse array* data structure. Unfortunately, this solution will not always work in our case, because a sparse array is implemented using indirection using an array of pointers to dynamically allocated sub-arrays. Just like the indirection through the first array to find the neighbor list in CSR, having to pass through this indirection to find the node data would almost certainly cause an additional cache miss per access because of the graph's lack of spatial locality.

6.1.3 Active vs. Passive Data Dependencies

At first glance, it appears that solutions such as sparse array notation for translating non-sequential node IDs to sequential data will never achieve good performance because the incur an additional cache miss to reach the data. However, unlike the neighbor list, the node data is not necessarily needed immediately to perform a node traversal; this depends largely on the graph algorithm, and, as we will see soon with the following three techniques, we will want to choose different time-space trade-offs depending on the node data's role in the algorithm. In the case of sparse array notation, we will certainly suffer performance losses in graph workloads where the internal node data is a direct prerequisite to perform the neighbor traversal. On the other hand, a workload which uses data for other purposes but not to decide neighbor traversals may be able to swallow the additional latency.

We define an *active data dependency* as any use of node data where the data is required to advance to the next node in the traversal, and loads to the next node must block until the data returns. An example of an active data dependency is choosing a child in a BST `find()` operation. In this instance, we cannot know whether to choose the left or right child until we learn how the value stored in the current node compares with our search key. Depending on the implementation, A* [18] can also contain active data dependencies, as it must compare the potential value with that of the target node to decide where to search next. (Note, however, that in our evaluations using the GAP benchmark suite [1], the node potential value is stored as part of the node structure, so it is not subject to this data dependency limitation. In CSR it is stored with the neighbor pointer, and in Merged CSR it is stored with the node degree).

Conversely, we define a *passive data dependency* as a use of internal node data that does not block a traversal to neighbor nodes. An example of this type of dependency is in breadth-first search. Consider a BFS implementation which searches for a specific value or values in the internal node data to know if a node is a match. In this instance, the node data is used only to find whether or not the algorithm can stop because we found the node we were looking for. The algorithm is perfectly capable of traversing to more neighbor nodes while it is waiting to find out while the current node is a match. In hardware, this behavior naturally occurs because the match check load will sit in the CPU's reorder buffer, while the neighbor loads can move along (although we expect they will incur a cache miss themselves). If we find out later after loading several neighbors that the current node is in fact a match, then we can stop the algorithm with no issues. Passive data dependencies like this one are more tolerant to long wait times for node data because the software can parallelize further edge traversals while waiting for the data to return.

We now propose three additional strategies for solving the problem of fetching sequentially-ordered node data. A general rule of thumb is that if a workload contains active data dependences as a key component of the algorithm, it is best to choose a format which produces the most timely loads for node data. If a workload contains only passive data dependencies, one may find it more useful to choose a format which less aggressively optimizes data loads at the benefit of space savings.

6.1.4 The Double-Node Method

In this strategy, we insert an additional value with a node's structure data, alongside the degree and neighbors, which stores the node's sequential ID. Adding the sequential ID allows us to point directly to the node data from the node structure data, allowing us to return the node data to the sequential array notaion used in standard CSR. However, the major caveat to this strategy is that we can no longer parallelize loading the node's neighbors and internal data as CSR does, because the logical pointer to the data is stored with the neighbors. Instead, we would have to load the node's degree, data pointer, and neigbors first using a pointer from the previous node's neighbors, and then load the data once we receive the data pointer. For workloads with active data dependencies, this method is not recommended, because data-dependent traverals will require two dependent cache misses per traversal, which is the same as standard CSR. On the other hand, workloads with only passive data dependencies will likely see less performance loss from the extra cache miss, and will benefit from this structure only requiring one additional value stored per node over what is required in standard CSR. As the name suggests, this method requires $2V + E$ values stored in the Merged CSR array, versus $V + E$ for standard CSR.

6.1.5 The Double-Edge Method

This method intends to mitigate the Double Node Method's issue that workloads with active data dependencies will incur an additional load-load dependency when loading the internal node data that directly delays the traversal to the next node. We can avoid this problem by storing the node's sequential ID, which acts as the logical pointer to the internal node data, alongside the node's non-sequential ID in the neighbor array in every node which has our node as a neighbor. This eliminates the load-load dependency we cited in the previous strategy, because now it is possible to fetch both the node's neighbor list (using the non-sequential ID) and the node's internal data (using the sequential ID) in the same DRAM latency cycle. The downside, however, is the increased space cost required. Because we now have to store two values for each edge, the number of indices in the structure array is up to $V + 2E$, which is quite expensive for dense graphs.

6.1.6 Weaving the Internal Data into the Structure Array

The primary logic behind Merged CSR was to reduce memory latency by placing data which is likely to be accessed together in the same or adjacent cache lines. For particularly data-bound traversals, we may find it useful to apply this same logic to the internal node data as well. This strategy involve leaving a fixed amount of space between each of nodes' neighbor lists in the structure array to store node data. Because the data appears alongside the neighbors, we should have no trouble loading it without a cache miss when the node is first loaded, provided we use an aggressive enough next-line prefetcher to ensure all the data is covered. In addition, since each node's data slice is the same size, we could forgo having to store its size alongside the neighbor list, and thus incur no additional memory overhead versus CSR. That is, the structure array size $V + E$, excluding the node data, which is stored separately in CSR.

The trade-off in this case, though, is the flexibility the programmer has with the node data slices. Because the data slices are interweaved into the graph structure, the slice size must be known at load-time, and it is not possible to deallocate internal node data once it is no longer in use. Accordingly, although this method is the most space efficient in the best case, in workloads which use many temporary data stores, one will likely find that this method ends up using more memory, because the memory for the temporary stores must stick around after it is no longer used. Luckily, it is entirely possible to use a hybrid scheme where temporary data stores that cause only passive data dependencies are still stored using separate arrays, like in standard CSR, and the weaved data slices are reserved for long-term data that causes active data dependencies, such as A*'s [18] node potential values.

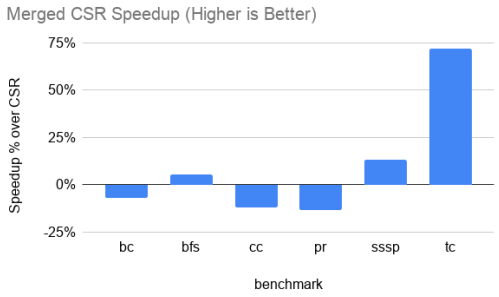

Figure 8: The speedup achieved using Merged CSR compared to standard CSR

6.2 Evaluation of Merged CSR

We evaluate Merged CSR on the GAP benchmark suite [1], using the Linux `perf` tool [19] on the same machine specs and graph set specified in section 5.5.

The control group in our experiments was the standard CSR implementation provided by the GAP benchmark suite [1]. Our evaluations for our experimental group were run on a reimplementation of that same data structure. Our modified Merged CSR implementation follows the design described by section 6.1.5. We chose this version largely because of our processor's lack of prefetching. In order to obtain useful results, we needed to stray away from techniques like the one in section 6.1.1, because merely initializing an array with length proportional to $V + E$ will perform extremely poorly. On an actual server in practice, it is unreasonable to assume we will have no prefetcher at all, and even a next-line prefetcher would almost completely eliminate the latency in initializing such an array. In our testing, just attempting to initialize an array of that length with no prefetcher was solely reponsible for over 45% of the total cache misses for the experiment, which is easily enough to shadow any performace differences we would hope to observe between Merged CSR and standard CSR. In order to obtain useful data which was also the most applicable to a server with prefetching, we needed to use one of the CSR techniques which allows the internal node data to retain its original structure. Of the three such designs we suggested in section 6.1, we chose the design in 6.1.5 for our evaluations because it should give us the best chance of observing the potential gains for benchmarks such as `sssp` (single-source shortest path) which contain many active data dependencies. Figure 8 shows the performance gains we achieve using Merged CSR versus standard CSR. By far, the benchmark which achieves the largest performace gain is `tc` (triangle counter). We believe this is because this benchmark had the most headroom to save in the first place. The main loop of triangle counter's implementation is extremely traversal-heavy, and it performs little expensive computation in between the traversals. Contrast this with `sssp` (single-source shortest path), where we saw a 13% speedup on average. In our latency analysis, we see over 40% of `sssp`'s memory latency stems for misses on other data structures, such as the queue, and various node ID-indexed data arrays. Because of this additional latency, the optimizations Merged CSR provides affect a much smaller fraction of the total memory latency, and thus we do not see the same dramatic speedup we see in `tc`.

The benchmarks for `bc` (betweenness centrality), `cc` (connected components), and `pr` (PageRank) instead see losses in performance as a result of using Merged CSR. These benchmarks all have one important characteristic in common. All of these applications' main loops iterate through the graph's nodes in memory order. With no prefetching, this operation is much faster in standard CSR than in Merged CSR, because in Merged CSR we must traverse the nodes like a linked list. Moreover, the nodes are more spread apart than in the first array of CSR, because the neighbor lists are inserted in between the nodes. This means that the fraction of the time we incur a cache miss when traversing from one node to the next will be significantly higher in Merged

CSR than in standard CSR, which puts a significant hindrance on our performance.

The `bfs` (breadth-first search) benchmark also includes a main loop which iterates through the graph's nodes in memory-order, but in its case we instead see marginal gains. Importantly, though, the performance improvements we saw were almost entirely in the smaller and sparser graphs we tested. In the graphs we used that represent road maps of the USA, we see upwards of a 70% speedup, and the smaller graphs we tested (~100,000 nodes), we saw an average of about a 45% speedup. However, on large, dense graphs, Merged CSR tends to suffer minor performance losses versus standard CSR. One possible explanation for this behavior is that in sparse graphs, the fraction of the time we encounter a cache miss upon performing each linked list traversal to the next node is smaller, since the neighbor lists between the traversal points will be smaller.

The results for these last four benchmarks show that we may be able to improve performance even further by introducing a simple prefetcher alongside Merged CSR. Specifically, we suspect that most of the performace gains Merged CSR achieves by reducing the cache misses incurred in loading node neighbors is cancelled out by the increased number of misses suffered when traversing the graph in memory-order as a linked list with no prefetching. This hypothesis is supported by another observation we made during our evaluations: these benchmarks encountered a much higher fraction of their overall cache misses in the code function we used to traverse to the next node in memory.

Since the stream only goes forward in memory, we believe that introducing an aggressive next-line prefetcher could lead to significant performance gains. In other words, even though changing from CSR to Merged CSR in these benchmarks did not improve performance with no prefetching, it did translate a major fraction of the memory latency to operations which are much easier to predict using previous hardware prefetching schemes. In the future, we believe it is critical to discover if a simple prefetching scheme can work in tandem with the improved memory locality introduced by Merged CSR to achieve singificant performance gains. Therefore, it would be a useful experiment to run the Merged CSR benchmarks using a hardware simulator where we can adjust the type and lookahead of the prefetcher, and analyze whether or not prefetching can help alleviate the high miss rate caused by these traversals.

7 Conclusions

In this paper, we demostrated the ability to improve graphs' spacial locality in order to reduce memory latency. We demostrated the viability and practicality of our new Neighbor Offset Extrapolation prefetcher, and showed that its software variant SNOE can significantly improve performance despite the poorer-than-expected cache locality of the NOE post cache when implemented in software. We also demonstrated that Merged CSR is able to improve spatial locality between nodes and their neighbors, leading to performance gains in benchmarks whose main loop largely consists of edge traversals. However, we obeserve slowdown in benchmarks whose main loop instead loops through nodes in memory order. This paper follows up on a longstanding chain of work focusing on graph optimizations in the prefetching domain, and we demonstrate the ability to apply techniques useful in hardware solutions to software optimizations in order to see moderate performance improvements even with no prefetcher present.

8 Future Work

Our work paves the way for several future experiments and directions for exploration.

Firstly, we would really like to answer the question of how well NOE performs in hardware versus its software variant SNOE. The memory latency data in section 5.5 shows that we would likely be able to see further performance improvements in hardware, since the NOE prefetcher uses a dedicated cache for posts and will not suffer from the frequent post evictions that SNOE does. We plan to implement the NOE

prefetcher in a hardware simulator to test this hypothesis and gather additional data about NOE's prefetch timeliness and accuracy.

Secondly, we observe that prefetching the entire neighbor list for high degree nodes may be infeasible in NOE. We are interested in analyzing the performance costs to ignoring high-degree nodes. If the cost is significant, we would be interested in exploring methods for reducing the prefetch overhead for high-degree nodes, possibly by placing a hard cap on the number of lines to prefetch, and then handling the remainder using a different prefetching scheme after the neighbor loop begins. We may also choose to explore dynamically adjusting the cap based on observed tempo of previous neighbor loops. Such a solution may improve performance by reducing cache pollution and preventing early eviction of the fetched neighbors.

Thirdly, the performance issues Merged CSR suffers from in benchmarks whose main loop performs an in-memory traversal of the graph's nodes suggests that it may significantly benefit from a hardware prefetcher. We are interested in experimenting with Merged CSR in a hardware simulator as well to test different prefetchers to see if we can improve on our performance results.

Finally, we are interested in exploring the general domain of how hardware and software solutions can work together. By considering both techniques simultaneously, we may come across combinations of schemes that work to achieve greater performance gains than either one alone.

9 Acknowledgements

I would like to thank my research advisors Dr. Calvin Lin and Dr. Akanksha Jain for the frequent advice and ideas in developing the concepts in this paper. I would also like to thank Matthew Pabst for providing some of the benchmark scripts used for testing different prefetchers and helping debug a few issues, and Ishan Shah for providing the set of graph inputs used in our evaluations.

References

[1] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite,"arXiv preprint arXiv:1508.03619, 2015.

[2] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in International Symposium on Microarchitecture, pp. 178–190, ACM, 2015.

[3] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '18.New York, NY, USA: Association for Computing Machinery, 2018, p. 578–592.[Online]. Available: https://doi.org/10.1145/3173162.3173189

[4] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2019, pp. 373–386.

[5] Chao Zhang, Yuan Zeng, John Shalf, and Xiaochen Guo, "RnR: A Software-Assisted Record-and-Replay Hardware Prefetcher", 53th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, Oct. 2020.

[6] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In Proceedings of the 38th International Symposium on Computer Architecture, 2010.

[7] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C.Steely Jr., and J. Emer. SHiP: Signature-based hit predictorfor high performance caching. In Proceedings of the 44thInternational Symposium on Microarchitecture, 2011.

[8] A. Jain and C. Lin, “Back to the future: Leveraging belady’s algorithm for improved cache replacement,” in Proceedings of the International Symposium on Computer Architecture (ISCA), June 2016.

[9] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In ISCA, 1990.

[10] P. Michaud, "Best-offset hardware prefetching," in 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016, pp. 469–480.

[11] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In Proceedings of the Tenth Symposium on High-Performance Computer Architecture, Feb. 2004.

[12] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In ISCA’06: Proceedings of the 33rd Annual International Symposium on Computer Architecture, pages 252–263,2006.

[13] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical off-chip meta-data for temporal memory streaming. In HPCA, pages 79–90,2009.

[14] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In ISCA, pages 69–80, 2009.

[15] A. Jain and C. Lin, “Linearizing irregular memory accesses for improved correlated prefetching,” in 46rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), December 2013.

[16] Hao Wu, Krishnendra Nathella, Akanksha Jain, Dam Sunwoo, and Calvin Lin. 2019. Efficient Metadata Management for Irregular Data Prefetching. In the 46th International Symposium on Computer Architecture (ISCA).

[17] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal Prefetching Without the Off-Chip Metadata," In MICRO ’52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 996-1008, 2019.

[18] P. E. Hart, N. J. Nilsson, B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and Cybernetics (Volume: 4, Issue: 2), 1968.

[19] The Linux Perf Tool. https://perf.wiki.kernel.org/index.php/Main_Page