Graph Prefetching Using Data Structure Knowledge

SAM AINSWORTH, TIMOTHY M. JONES
Background and Motivation

Graph applications are memory latency bound
- Caches & Prefetching are existing solution for memory latency
- However, irregular access patterns hinder their usefulness

Key insight: accesses seem irregular at individual load/store level, but have **predictable structure when we consider the high-level algorithm**
- e.g. Breadth-first search (BFS)
Background and Motivation

Existing SW/HW prefetching is insufficient

(a) Hardware prefetchers

(b) Hardware vs software

Figure 3: Hardware and software prefetching on Graph 500 search with scale 21, edge factor 10.
Background and Motivation - BFS

Unvisited

Visited

Will be visited after visiting C! So we can prefetch after visiting B
Prefetching with Algorithmic Knowledge

Design a hardware prefetcher that relies on access patterns specific to algorithms

- Target BFS, but can support a wider range of algorithms/access patterns
- Specific to Compressed Sparse Row (CSR) format
- Prefetcher snoop reads/writes from L1 cache

Achieve an average of 2.3x speedup
Review: Compressed Sparse Row (CSR)

Sparse representation, with a vertex list indirection to an edge list

- Authors add a *visited list* and *work list* specifically for BFS

![Graph](image)

![CSR breadth-first search](image)

*Figure 1: A compressed sparse row format graph and breadth-first search on it.*
Breadth-first search is a common access pattern in graph algorithms. It involves visiting all the nodes in a graph in breadth-first order, which can be implemented using a queue. The algorithm typically starts with a root node and iteratively visits all the nodes at the current level before moving to the next level.

```
Algorithm 1: Breadth-first search

startNode = root
workList = Queue
end

while workList is not empty do
    N = workList.dequeue()
    if visited[N] is false then
        visited[N] = true
        for each adjacent E of N do
            if visited[E.to] is false then
                workList.enqueue(E.to)
        end
    end
end
```

This algorithm is efficient for graphs with a tree-like structure, where each node has a limited number of direct neighbors. However, it can suffer from poor locality of access if the graph has a highly connected structure or if the data is stored in a format that does not support efficient traversal.

In today’s conventional commodity hardware, graphs larger than the L1 cache can result in a significant impact on performance, especially for graph algorithms that traverse the graph in breadth-first order. The data access patterns for breadth-first search access data in a seemingly random order, which is detrimental to modern cache hierarchies.

Poor Locality of Accesses in Graphs

![Graph 500 search](image.png)

The images above show the stall rate and L1 miss rate for different scales and edge factors. The pie chart indicates the source of misses, with the edge list being the largest component.

**Figure 1:** A compressed sparse row format graph and an example of one iteration of the outer loop shown in the figure.
Overview of Approach

Prefetch all relevant data of \( o \)-distance away from the current worklist entry:

\[
\text{visited}[\text{edgeList}[\text{vertexList}[\text{workList}[n+o]]]]
\]

Prefetcher snoops the core-to-L1 mem. accesses to determine which data to prefetch

<table>
<thead>
<tr>
<th>Observation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load from \text{workList}[n]</td>
<td>Prefetch \text{workList}[n+o]</td>
</tr>
<tr>
<td>Prefetch \text{vid} = \text{workList}[n]</td>
<td>Prefetch \text{vertexList}[\text{vid}]</td>
</tr>
<tr>
<td>Prefetch from \text{vertexList}[\text{vid}]</td>
<td>Prefetch \text{edgeList}[\text{vertexList}[\text{vid}]] to \text{edgeList}[\text{vertexList}[\text{vid}+1]] (12 lines max)</td>
</tr>
<tr>
<td>Prefetch \text{vid} = \text{edgeList}[\text{eid}]</td>
<td>Prefetch \text{visited}[\text{vid}]</td>
</tr>
</tbody>
</table>

**Vertex-Offset Mode**

**Table 1**: Actions taken by the prefetcher in response to observations on L1 activity.
System Architecture

(a) System overview

Figure 5: A graph prefetcher, configured with in-memory data structures, to which it snoops accesses.

Table 1: Actions taken by the prefetcher in response to observations on L1 activity.
Prefetcher Microarchitecture

(b) Prefetcher microarchitecture detail
Determining Prefetch Distance

Easy Case: Time to process a vertex \((work\_list\_time)\) is less than time to pre-fetch the next vertex \((data\_time)\)

\[
o \times work\_list\_time = data\_time
\]

\(work\_list\_time\) and \(data\_time\) vary wildly \(\Rightarrow\) use exponentially weighted moving averages (EWMA)

Use a safe bound because EWMA often underestimates \(data\_time\):

\[
o = 1 + \frac{k \times data\_time}{work\_list\_time}
\]
Determining Prefetch Distance

Problem: \textit{work\_list\_time} > \textit{data\_time}

- Pre-fetched data is not used timely, might get kicked out of cache before it is used!
- Happens with high-degree vertices

Solution: Large vertex mode

- Base prefetch on how far along we have processed the high-degree vertex
  
  » Possible because we know the range of the edge indices
- Prefetch within \textit{edgeList} for larger vertex
- Fetch need vertex in \textit{worklist} when almost done with current vertex’s edges
Extensions

Technique can be extended to other algorithms:
- Parallel BFS
- Sequentially scanning vertex and edge data (e.g. PageRank)
Methodology

gem5 simulator

Set of algorithms from Graph500 and the Boost Graph Library:

- BFS-like traversal: Connected components, BFS, betweenness-centrality, ST connectivity
- Sequential access: PageRank, sequential coloring
Evaluation - BFS-like traversal

5. EVALUATION

We first evaluate our prefetcher on breadth-first-search-based applications and analyse the results. Then we move on to algorithms that perform sequential access through data structures, and parallel breadth-first search.

5.1 Performance

Our hardware prefetcher brings average speedups of 2.8× on Graph 500 and 1.8× on BGL algorithms. Figure 6 shows the performance of the breadth-first search (BFS) hardware prefetcher against the best stride scheme under simulation, and a stride-indirect scheme as suggested by Yu et al. [44], which strides on the edge list into the visited list. Stride prefetching performs poorly, obtaining an average of 1.1×. Stride-indirect performs only slightly better with an average of 1.2×, as breadth first searches do not exhibit this pattern significantly, causing a large number of unused memory accesses. For comparison, under the same simulation conditions, augmenting binaries with software prefetching gave speedups of no more than 1.1×.

Our hardware prefetcher increases performance by over 2× across the board for Graph 500. In the BGL algorithms, basic breadth-first searches perform comparably to Graph 500's search, but betweenness centrality achieves a much smaller performance increase, averaging 20%, due to significantly more calculation and non-breadth-first-search data accesses. In fact, the Boost betweenness centrality code involves data-dependent accesses to various queue structures and dependency metrics, which are only accessed on some edge visits and are not possible to prefetch accurately. This algorithm also accesses two data structures indexed by the edge value: the visited list, and also a distance vector. For evaluation, we implemented an extension for the prefetcher to set two “visited” lists, allowing both to be prefetched, improving on average by an extra 5%.

Around 20% of our benefit comes from prefetching TLB entries; due to the heavily irregular data accesses observed, and the large data size, many pages are in active use at once. However, by virtue of prefetching these entries when performing prefetching of the data itself, these entries should be in the L2 TLB when the main thread reaches a given load, avoiding stalls on table walks.

5.2 Analysis

We now analyse the effect of our prefetcher on the system, considering the changes in L1 hit rates, memory accesses and utilisation of prefetched data, shown in figures 7 to 9.
Evaluation - BFS-like traversal

We first evaluate our prefetcher on breadth-first-search-based applications and analyse the results. Then we move on to algorithms that perform sequential access through data structures, and parallel breadth-first search.

5.1 Performance

Our hardware prefetcher brings average speedups of 2.8 $\times$ on Graph 500 and 1.8 $\times$ on BGL algorithms. Figure 6 shows the performance of the breadth-first search (BFS) hardware prefetcher against the best stride scheme under simulation, and a stride-indirect scheme as suggested by Yu et al. [44], which strides on the edge list into the visited list. Stride prefetching performs poorly, obtaining an average of 1.1 $\times$. Stride-indirect performs only slightly better with an average of 1.2 $\times$, as breadth first searches do not exhibit this pattern significantly, causing a large number of unused memory accesses. For comparison, under the same simulation conditions, augmenting binaries with software prefetching gave speedups of no more than 1.1 $\times$.

Our hardware prefetcher increases performance by over 2 $\times$ across the board for Graph 500. In the BGL algorithms, basic breadth-first searches perform comparably to Graph 500's search, but betweenness centrality achieves a much smaller performance increase, averaging 20%, due to significantly more calculation and non-breadth-first-search data accesses. In fact, the Boost betweenness centrality code involves data-dependent accesses to various queue structures and dependency metrics, which are only accessed on some edge visits and are not possible to prefetch accurately. This algorithm also accesses two data structures indexed by the edge value: the visited list, and also a distance vector. For evaluation, we implemented an extension for the prefetcher to set two “visited” lists, allowing both to be prefetched, improving on average by an extra 5%.

Around 20% of our benefit comes from prefetching TLB entries; due to the heavily irregular data accesses observed, and the large data size, many pages are in active use at once. However, by virtue of prefetching these entries when performing prefetching of the data itself, these entries should be in the L2 TLB when the main thread reaches a given load, avoiding stalls on table walks.

5.2 Analysis

We now analyse the effect of our prefetcher on the system, considering the changes in L1 hit rates, memory accesses and utilisation of prefetched data, shown in figures 7 to 9.
Significantly improved L1 hit-rate

Figure 7: Hit rates in the L1 cache with and without prefetching.
Prefetching has low overheads

Figure 8: Percentage of additional memory accesses as a result of using our prefetcher.

Figure 9: Rates of prefetched cache lines that are used before leaving the L1 cache.
Prefetching Analysis

Most of the benefit comes from prefetching visited & edge lists -> as expected!

Figure 10: The proportion of speedup from prefetching each data structure within the breadth first search.
Prefetching works for other traversal types

Example: parallel BFS

Figure 11: Speedup relative to 1 core with a parallel implementation of Graph500 search with scale 21, edge factor 10 using OpenMP.
Prefetching works for other traversal types

Figure 12: Speedup for different types of prefetching when running PageRank and Sequential Colouring.
Conclusion

Prefetching with knowledge of the graph traversal order significantly improves its performance

◦ Works for different traversal types (BFS, sequential scan, ...)

22