Graph Prefetching Using Data Structure Knowledge

SAM AINSWORTH, TIMOTHY M. JONES

Background and Motivation

Graph applications are memory latency bound

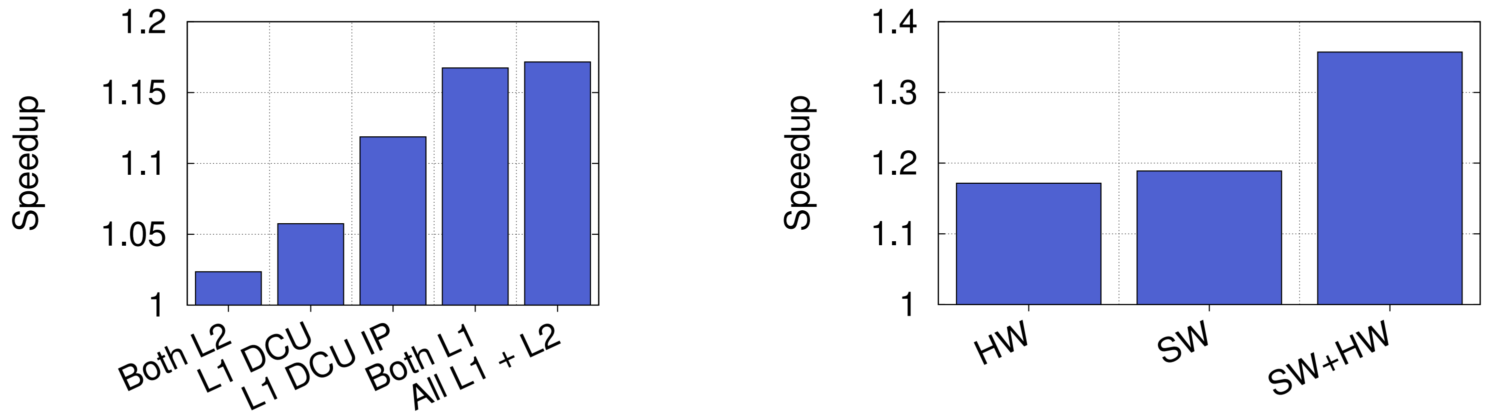
- Caches & Prefetching are existing solution for memory latency
- However, irregular access patterns hinder their usefulness

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

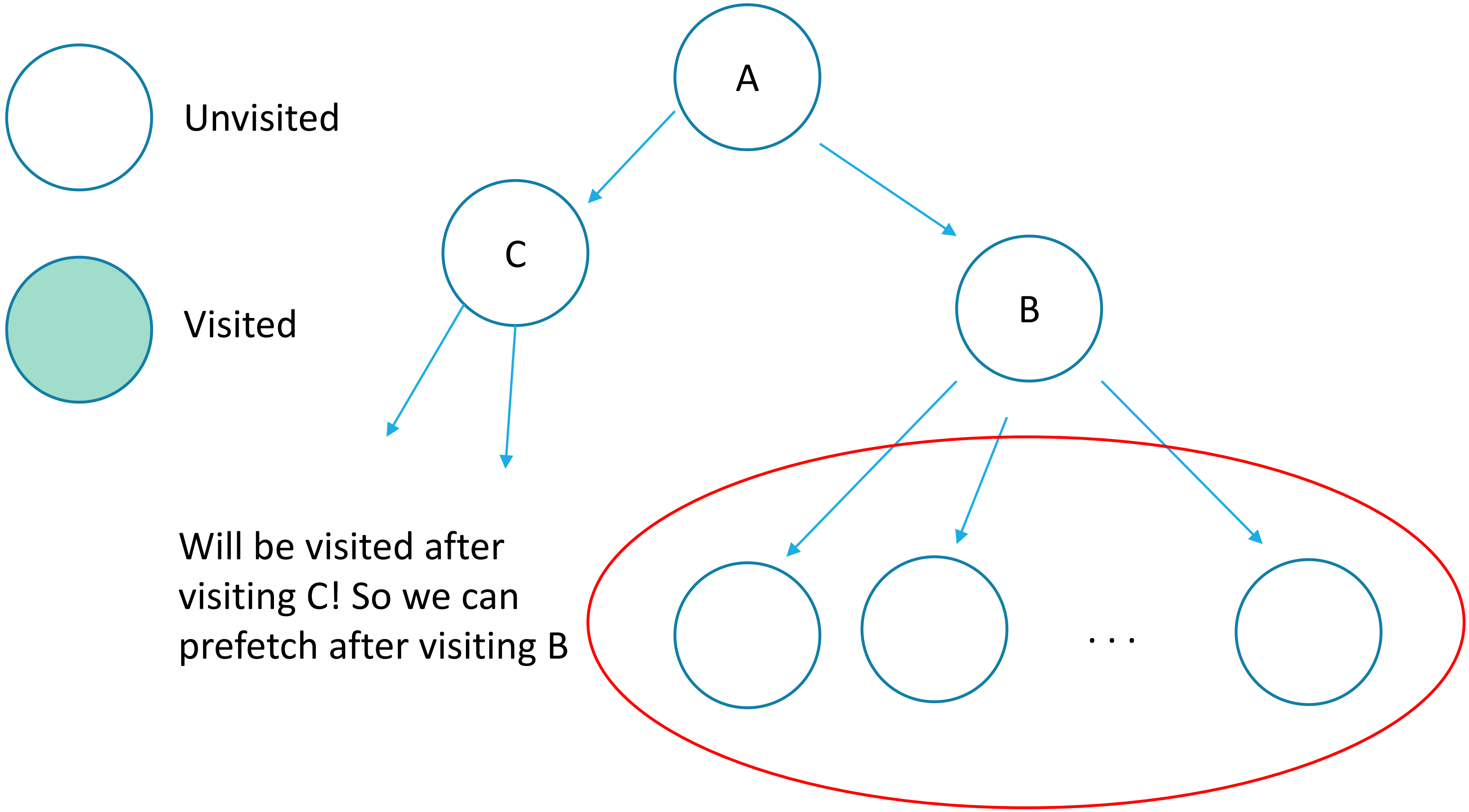
Existing SW/HW prefetching is insufficient



(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



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**

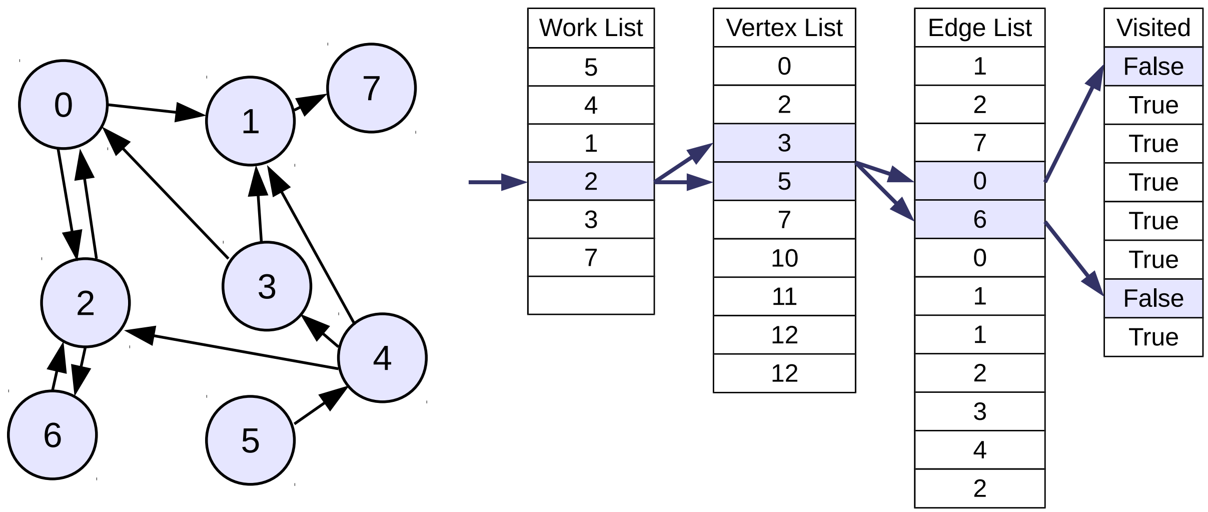
- Target BFS, but can support a wider range of algorithms/access patterns
- Specific to Compressed Sparse Row (CSR) format
- Prefetcher snoop reads/writes from L1 cache

Achieve an average of 2.3x speedup

Review: Compressed Sparse Row (CSR)

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

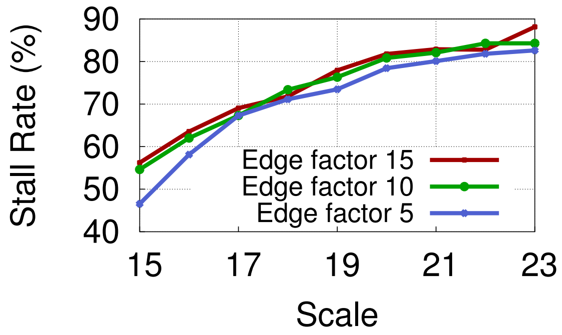
- Authors add a *visited list* and *work list* specifically for BFS



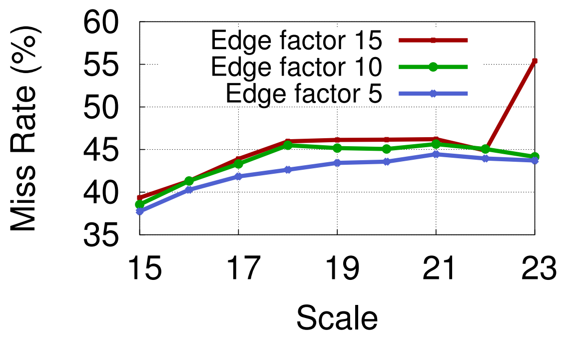
(a) Graph (b) CSR breadth-first search

Figure 1: A compressed sparse row format graph and breadth-first search on it.

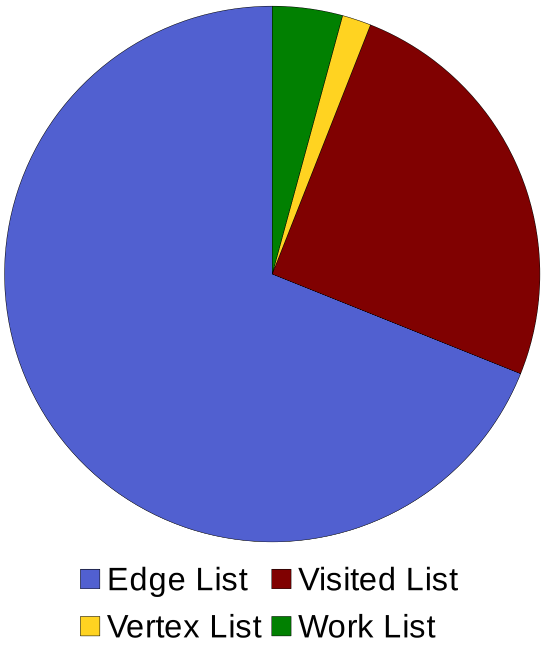
Poor Locality of Accesses in Graphs



(a) Stall rate



(b) L1 miss rate



(c) Source of misses

Overview of Approach

Prefetch all relevant data of *o*-distance away from the current worklist entry:

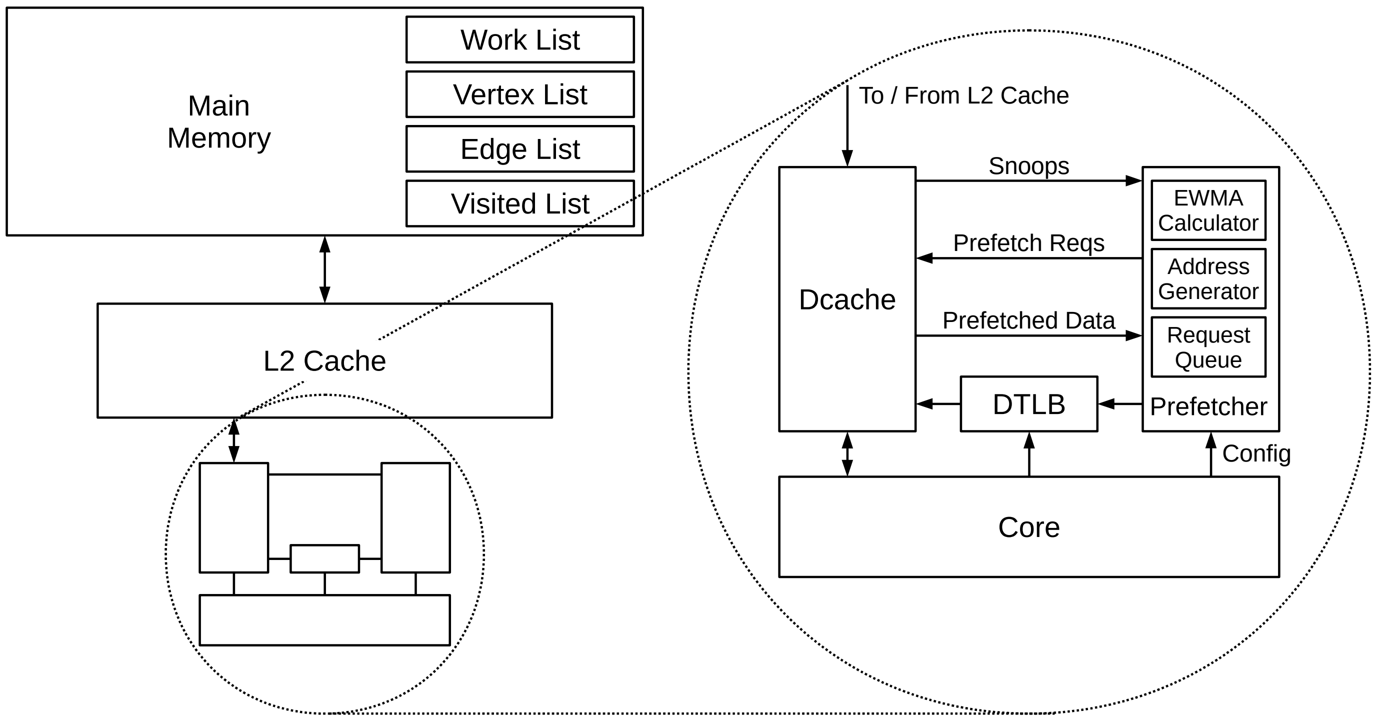
```
visited[edgeList[vertexList[workList[n+o]]]]
```

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

Vertex-Offset Mode

Observation	*Action*
Load from `workList[n]`	Prefetch `workList[n+o]`
Prefetch `vid = workList[n]`	Prefetch `vertexList[vid]`
Prefetch from `vertexList[vid]`	Prefetch `edgeList[vertexList[vid]]` to `edgeList[vertexList[vid+1]]` (12 lines max)
Prefetch `vid = edgeList[eid]`	Prefetch `visited[vid]`

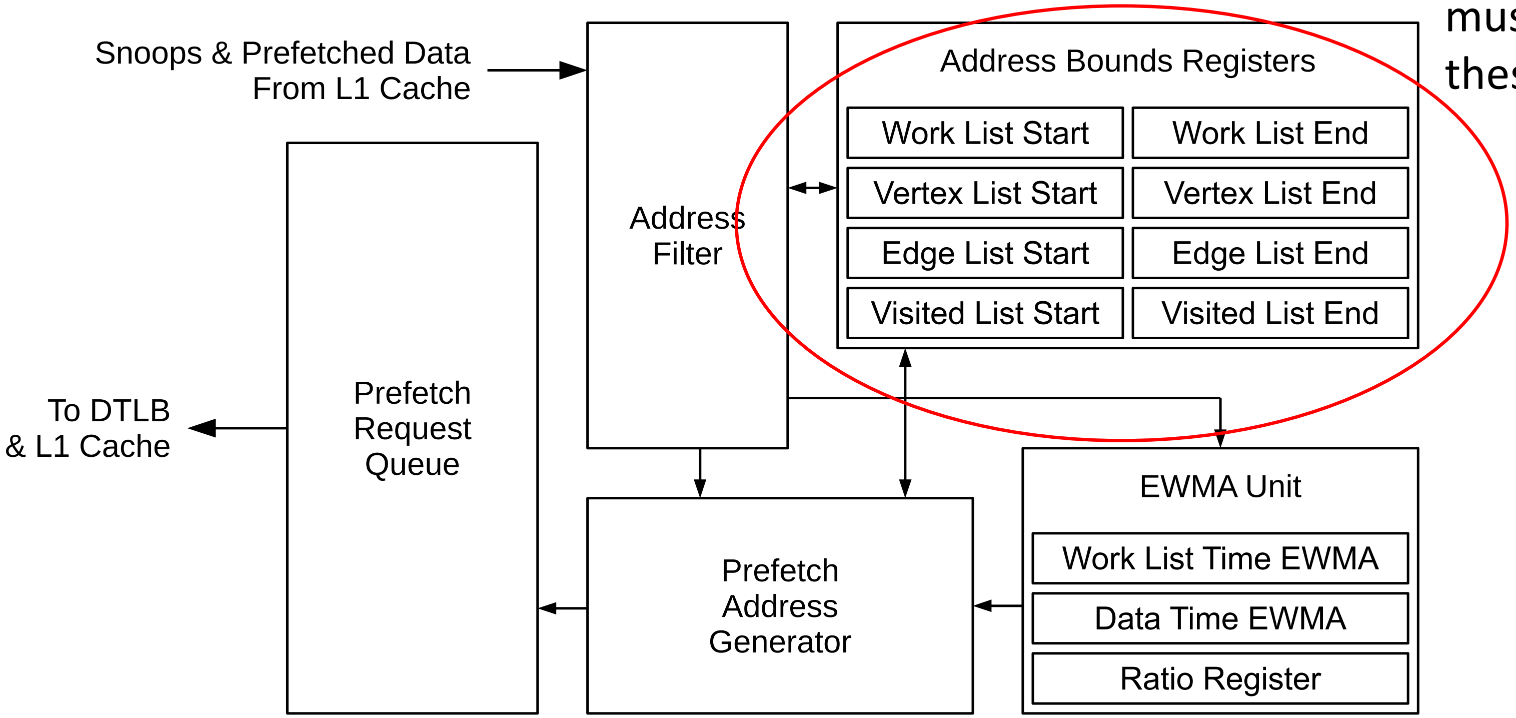
System Architecture



(a) System overview

Prefetcher Microarchitecture

Programmer must specify these bounds



(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*)

$$o * work\_list\_time = data\_time$$

- *work_list_time* and *data_time* vary wildly => use exponentially weighted moving averages (EWMA)
- Use a safe bound because EWMA often underestimates *data_time*:

$$o = 1 + \frac{k * data\_time}{work\_list\_time}$$

Determining Prefetch Distance

Problem: *work_list_time > 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 edgeList for larger vertex
- Fetch need vertex in 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)

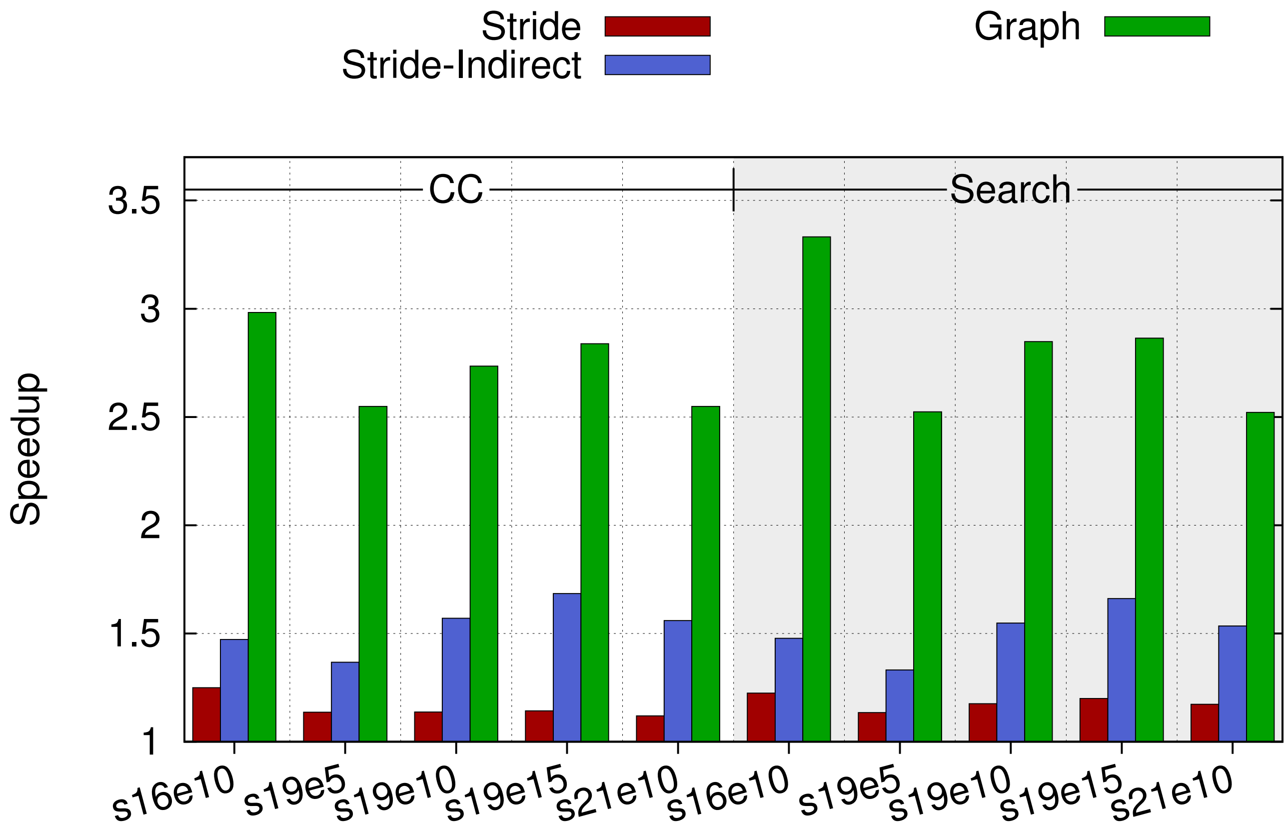
Methodology

gem5 simulator

Set of algorithms from Graph500 and the Boost Graph Library:

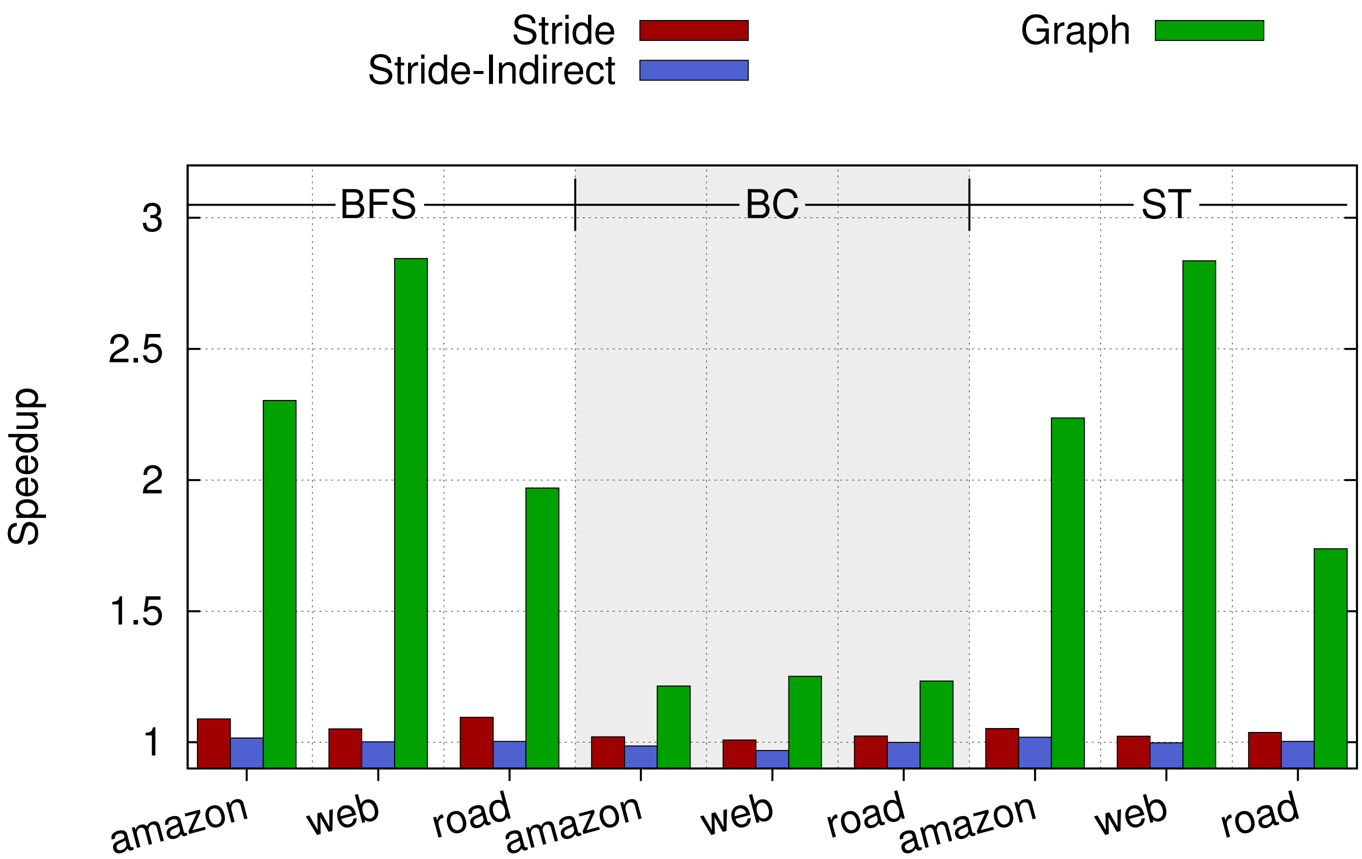
- BFS-like traversal: Connected components, BFS, betweenness-centrality, ST connectivity
- Sequential access: PageRank, sequential coloring

Evaluation - BFS-like traversal



(a) Graph 500

Evaluation - BFS-like traversal



(b) BGL

Significantly improved L1 hit-rate

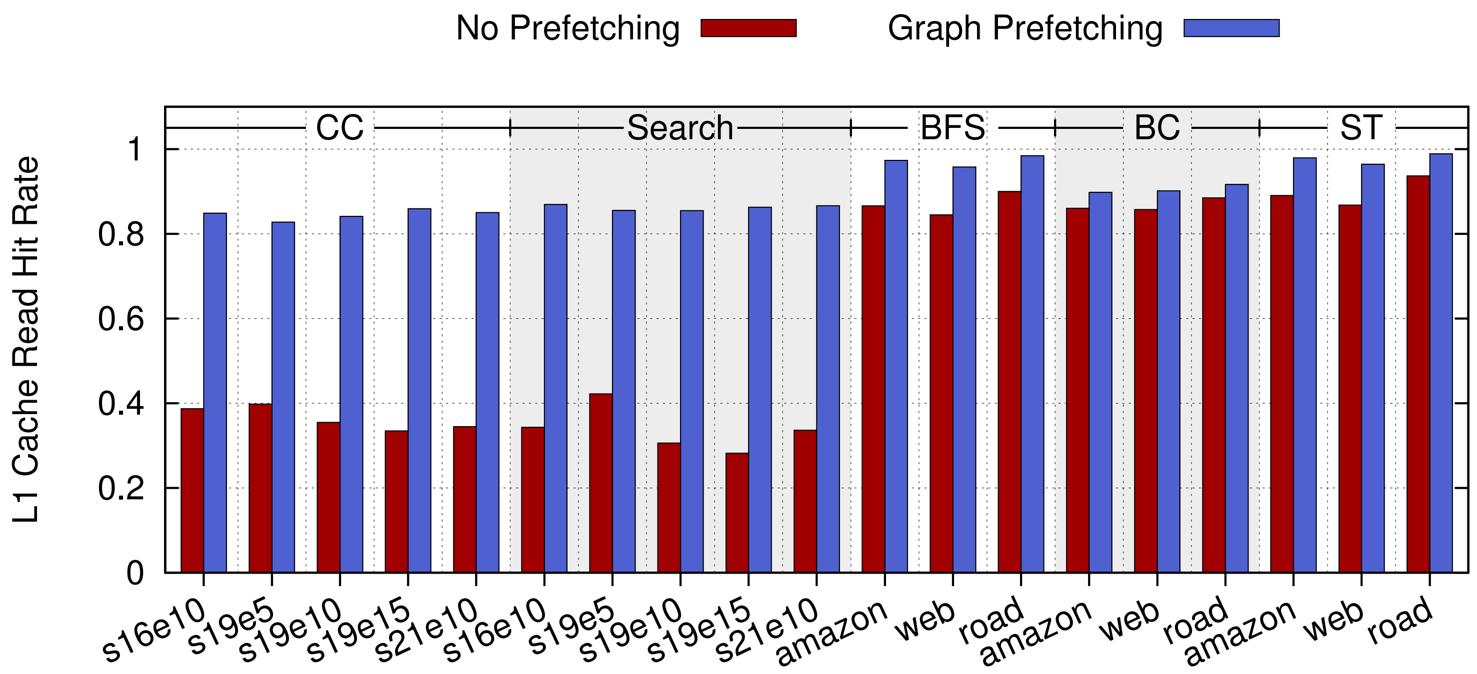


Figure 7: Hit rates in the L1 cache with and without prefetching.

Prefetching has low overheads

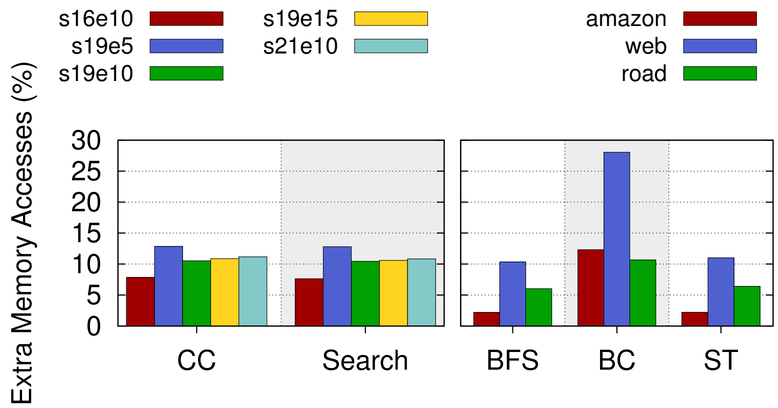


Figure 8: Percentage of additional memory accesses as a result of using our prefetcher.

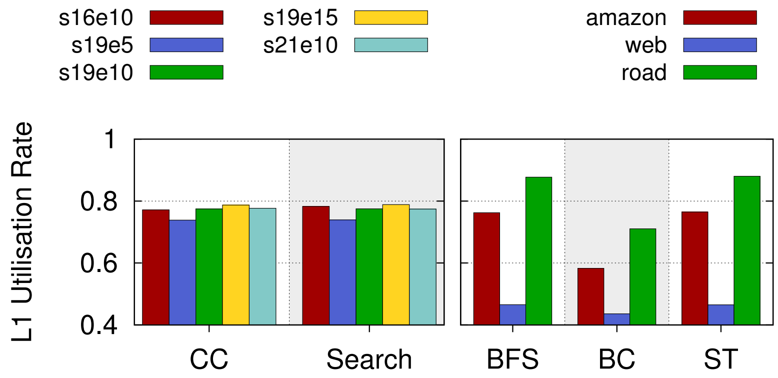


Figure 9: Rates of prefetched cache lines that are used before leaving the L1 cache.

Prefetching Analysis

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

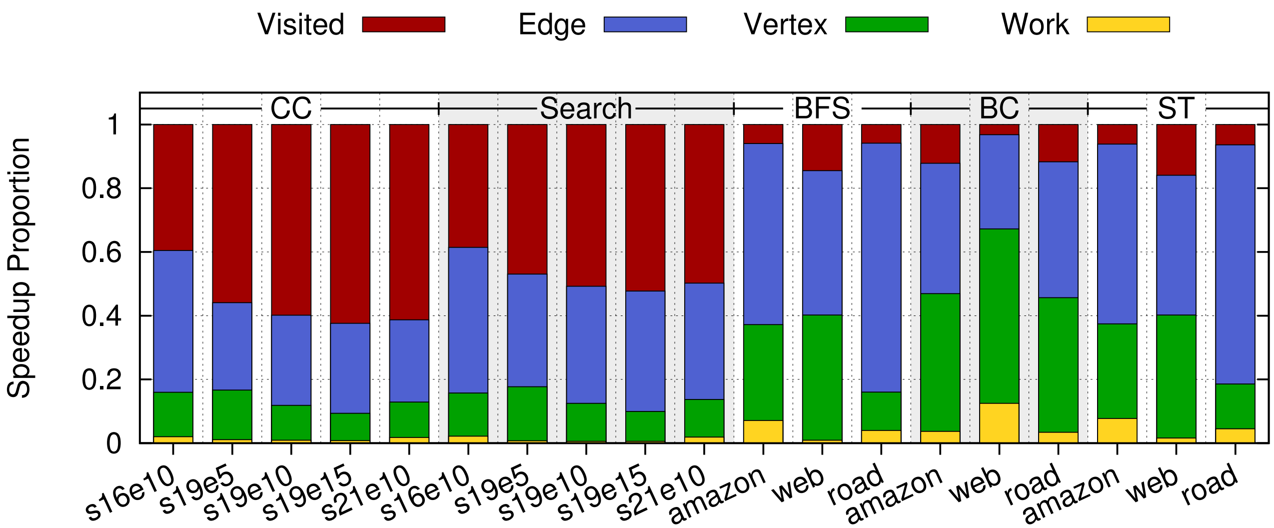


Figure 10: The proportion of speedup from prefetching each data structure within the breadth first search.

Prefetching works for other traversal types

Example: parallel BFS

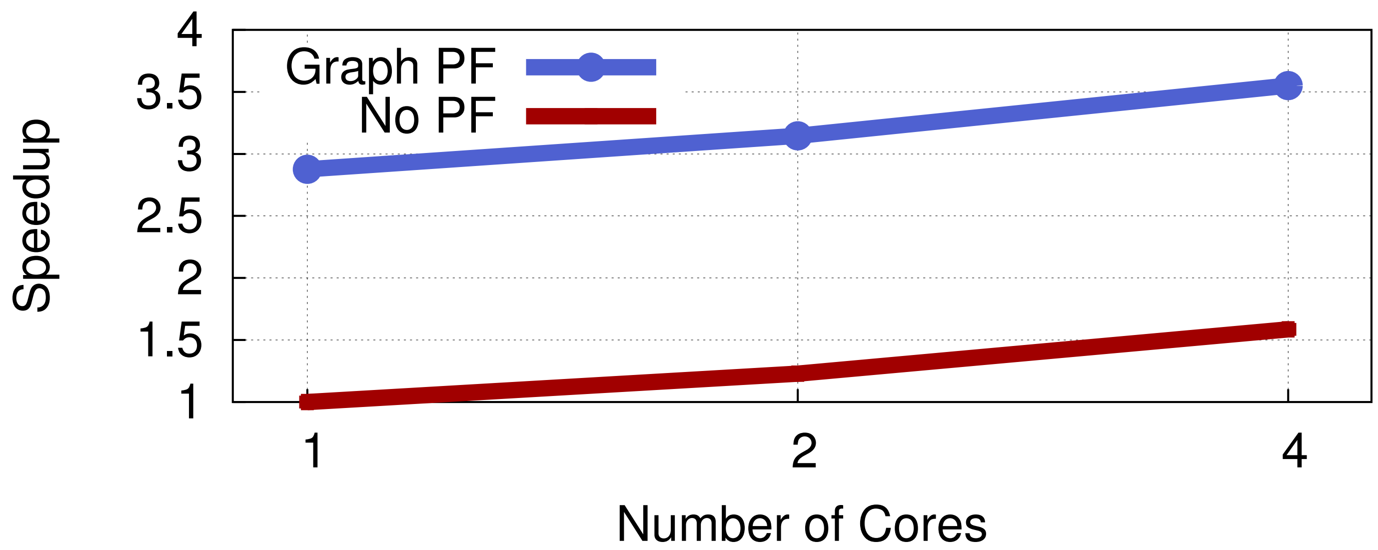


Figure 11: Speedup relative to 1 core with a parallel implementation of Graph500 search with scale 21, edge factor 10 using OpenMP.

Prefetching works for other traversal types

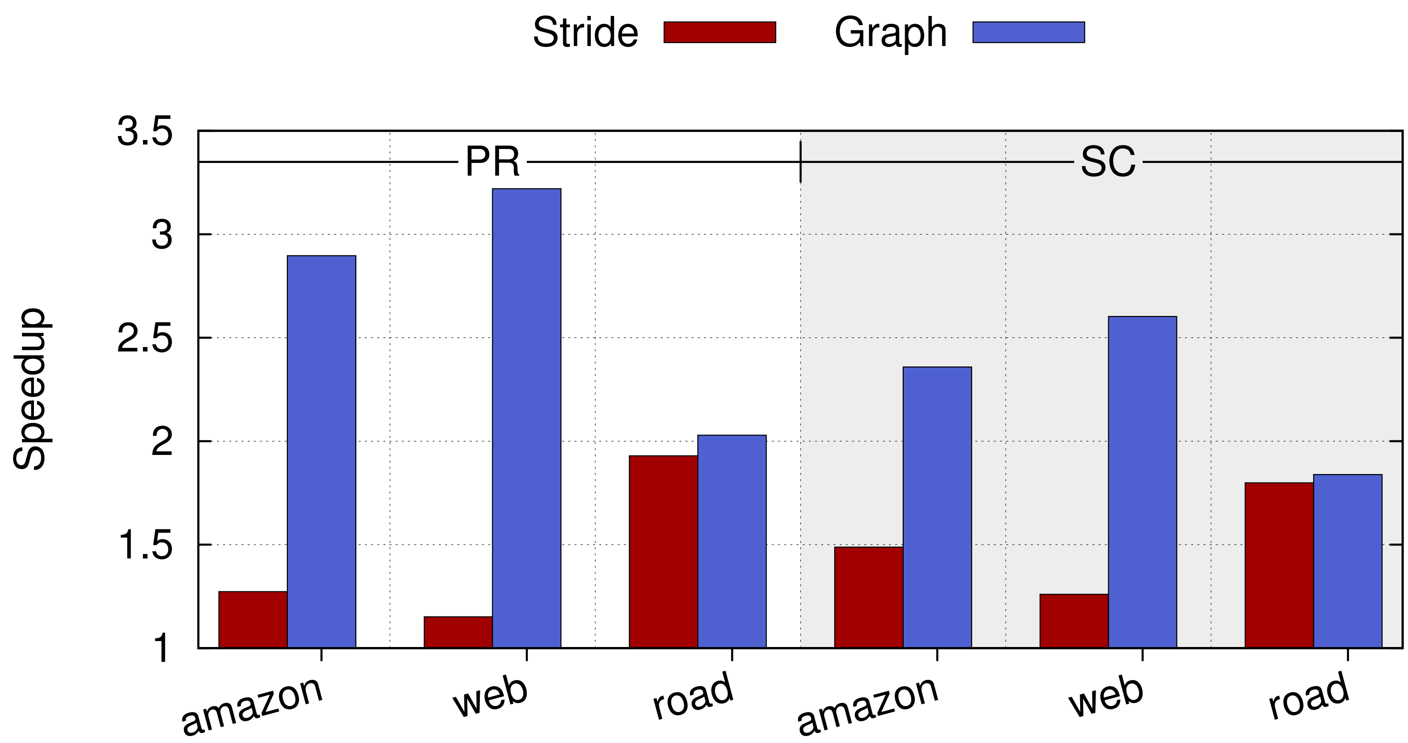


Figure 12: Speedup for different types of prefetching when running PageRank and Sequential Colouring.

Conclusion

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

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