Graph Prefetching Using Data Structure Knowledge

Sam Ainsworth
University of Cambridge
sam.ainsworth@cl.cam.ac.uk

Timothy M. Jones
University of Cambridge
timothy.jones@cl.cam.ac.uk

ABSTRACT

Searches on large graphs are heavily memory latency bound, as a result of many high latency DRAM accesses. Due to the highly irregular nature of the access patterns involved, caches and prefetchers, both hardware and software, perform poorly on graph workloads. This leads to CPU stalling for the majority of the time. However, in many cases the data access pattern is well defined and predictable in advance, many falling into a small set of simple patterns. Although existing implicit prefetchers cannot bring significant benefit, a prefetcher armed with knowledge of the data structures and access patterns could accurately anticipate applications' traversals to bring in the appropriate data.

This paper presents a design of an explicitly configured prefetcher to improve performance for breadth-first searches and sequential iteration on the efficient and commonly-used compressed sparse row graph format. By snooping L1 cache accesses from the core and reacting to data returned from its own prefetches, the prefetcher can schedule timely loads of data in advance of the application needing it. For a range of applications and graph sizes, our prefetcher achieves average speedups of 2.3×, and up to 3.3×, with little impact on memory bandwidth requirements.

CCS Concepts

•**Computer systems organization** → Special purpose systems;

Keywords

Prefetching, Graphs

1. INTRODUCTION

Many modern workloads are heavily memory latency bound, due to the von Neumann bottleneck in conventional multiprocessors. This is especially true for graph workloads [1, 34], which require vast amounts of data to be fetched from high latency main memory.

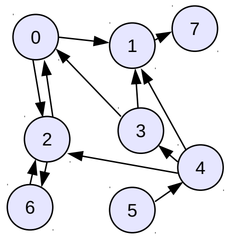
ICS '16, June 01-03, 2016, Istanbul, Turkey
© 2016 ACM. ISBN 978-1-4503-4361-9/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2925426.2926254

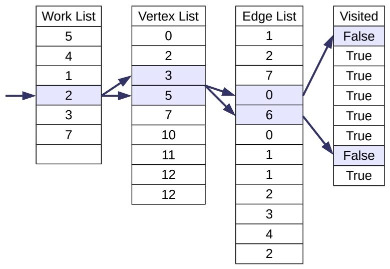
Graph computation is employed in a wide variety of disciplines, for example routing, network analysis, web crawling, bioinformatics, marketing and social media analysis [14, 32], where the aim is to extract meaningful measurements from real-world "Big Data" to make judgments about the systems they are based on [39]. Analysing the vast amounts of data contained in these graphs is highly time consuming, difficult to parallelise and requires data-dependent traversal methods [34], making these workloads extremely inefficient using current computation paradigms.

A common way to improve throughput for memory-bound workloads is prefetching, where hardware learns the access patterns of computation kernels, and predicts future accesses, issuing them before the data is requested so that the data is in the low latency, high throughput cache when required. However, since graph access patterns tend to be data-dependent and non-sequential, current hardware prefetchers perform very poorly; stride prefetchers are unable to follow the data-dependent access patterns [12], those that correlate sequences of visited addresses suffer from both a large storage cost and an inability to improve non-repeated computation [12], and pointer fetchers [6] are unable to target indirect array accesses. Similarly, frameworks such as MapReduce are inefficient for graph workloads as they aren't inherently embarrassingly parallel, and require many iterations, resulting in significant overheads [30, 27].

However, despite existing hardware's inability to implicitly find structure within accesses, in common graph workloads the data access pattern is typically well defined and predictable in advance: a great deal of computations fall into simple patterns such as breadth-, depth-, and best-first search [34]. Although implicit prefetchers, such as those used to pick up stride accesses on conventional microprocessors, cannot aid these workloads, their performance could be improved through the use of an explicit prefetcher, which is armed with knowledge of the data structures being used and the traversals being performed. Using this information, the prefetcher could anticipate the data required by the application in the near future and, crucially, know where to look for it. Further, it obviates the need to learn access patterns, resulting in less state to maintain and earlier prefetching decisions.

This paper presents a hardware prefetcher for breadth-first searches on graphs stored as compressed sparse row structures. Breadth-first search is a highly common compute kernel used in social media analysis [32], web crawling [33, 40], model checking [3] and many other fields, as it can be used as the basis of a large number of algorithms [34]. Com-


(a) Graph



(b) CSR breadth-first search

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

```
1  Queue workList = {startNode}
2  Array visited[startNode] = true
3  while $worklist \neq \varnothing$ do
4      Node N = workList.dequeue()
5      foreach Edge $E \in N$ do
6          if visited[E.to] is false then
7              workList.enqueue(E.to)
8              visited[E.to] = true
9          end
10     end
11 end
```

Algorithm 1: Breadth-first search.

pressed sparse row format is the de facto standard representation of sparse graphs for high-performance compute due to its highly efficient storage format: most current work on high performance breadth-first search also focuses on this [34, 32]. Although other representations (such as GraphLab [26]) have good distribution properties, they perform more poorly due to additional overheads, such as data duplication [30].

We introduce a prefetcher that snoops reads of an in-memory queue to calculate and schedule highly accurate and timely loads of edge and vertex information into the L1 cache. For computation based around breadth-first search on compressed sparse row format graphs, our prefetcher achieves average speedups of 2.3×, and up to 3.3×, across a range of applications and graph sizes. We also extend the prefetcher for sequential iteration on such graphs, as used in PageRank [35], which achieves average speedups of 2.4× and up to 3.2×.

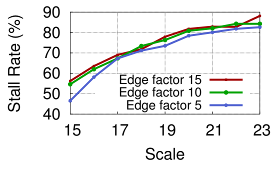
2. BACKGROUND

Compressed sparse row (CSR) data structures are an efficient sparse matrix representation that are commonly used for in-memory sparse graphs [1, 34] due to their compact nature. They can be used to store adjacency information for graphs by using two arrays: the edge list, which stores the non-zero elements of the adjacency matrix as a one dimensional array; and the vertex list, which contains the start edge list array index for each vertex. An example graph is shown in figure 1(a); the CSR format vertex list and edge list are shown in figure 1(b). Note that the CSR structures contain indices, not pointers, into the data arrays.

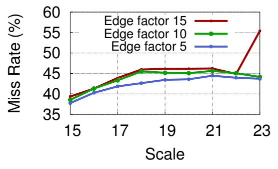
2.1 Breadth-First Search

Breadth-first search is a common access pattern in graph workloads: it can be used as a basic computation kernel to perform unweighted distance calculations, connected components [34], maximum flows via the Edmonds-Karp algorithm [11], optimal decision tree walking in AI [7], betweenness centrality [5], and many other algorithms. More recently, the concept has been applied to very large graphs as a kernel within many sub-linear algorithms, that only operate on a small fraction of the input data [37].

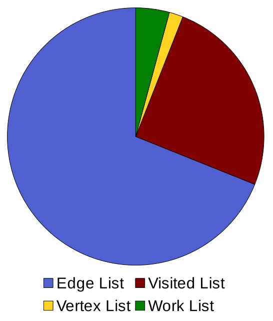
An overview of breadth-first search is shown in algorithm 1 and an example of one iteration of the outer loop shown in figure 1(b). From the starting vertex, computation moves through the graph adding vertices to a FIFO work list queue (implemented as an array) in the order observed via the edges out of each node. For example, stating at vertex 5 in figure 1(a), nodes are visited for breadth-first search in the order 5, 4, 1, 2, 3, 7, 0, 6.



(a) Stall rate



(b) L1 miss rate



(c) Source of misses

Figure 2: Core stall rate and L1 cache read miss rate for Graph 500 search. Loads from the edge and visited lists account for 94% of the misses.

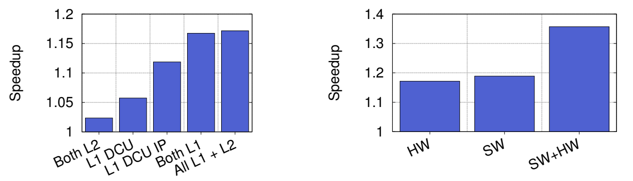
2.2 Stalling Behavior

The main issue with breadth-first search is that there is no temporal or spatial locality in accesses to the vertex list, and only locality in edge list accesses for a single vertex, meaning that graphs larger than the last-level cache get little benefit from caching. Figure 2(a) shows that the Graph 500 search benchmark [32], running on an Intel Core i5 4570 processor, experiences stall rates approaching 90%, increasing with graph size.[1] This is due to L1 cache misses approaching 50%, as can be seen in figure 2(b). Figure 2(c) shows the breakdown of the extra time spent dealing with misses for different types of data, using gem5 for the same benchmark with scale 16 and edge factor 10. The majority of additional time is due to edge list misses (69%), because the edge list is twenty times larger than the vertex list. In addition, the array that records whether each vertex has been visited or not is also a significant source of miss time (25%). Although this is the same size as the vertex list, it is accessed frequently (once for each edge into a vertex) in a seemingly random order.

2.3 Conventional Prefetching Techniques

Stride Prefetching In today's conventional commodity processors, stride-based prefetchers generally pervade [42]. These prefetchers work well for sequential accesses through arrays and matrices, but for irregular, data-dependent access

[1]Section 4 gives more detail on benchmarks, graphs and experimental setup.



(a) Hardware prefetchers (b) Hardware vs software

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

patterns, as in breadth-first search, these stride patterns do not appear.

Figure 3(a) shows the impact of existing prefetchers on the Graph 500 search benchmark on a Core i5 4570. L2 prefetchers, which consist of two distinct prefetchers, bring almost no benefit, and all combined bring only 17% improvement. The largest contribution is from the L1 data cache's DCU IP prefetcher, which prefetches based on sequential load history and the instruction pointer. This increases performance by 12%, most likely from prefetching edge list data that is stored contiguously for each vertex.

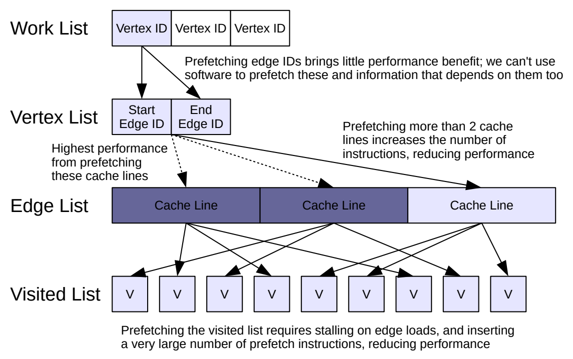
Software Prefetching In contrast to hardware techniques, software prefetch instruction can be inserted into the code when the programmer can give an indication of the most critical data. Figure 4 shows in detail the data that is profitably prefetched in software. Since prefetches cannot use the results of a prior prefetch, any loads to obtain data require stalling; i.e.; to software prefetch the visited list, we must issue real loads from the work, vertex and edge lists, causing stalls. Therefore there is a trade-off between prefetching information close to the work list to reduce loads (e.g., the vertex list) and the larger amount of information further away, (e.g., the visited list).

Although we wish to prefetch every edge for each vertex, this results in too many additional instructions, swamping the benefits. In addition, we cannot efficiently analyse the prefetch distance in software, meaning we must use a fixed distance even though the workload characteristics change throughout execution. Combined, these limitations mean that the best strategy was to add software prefetch between lines 4 and 5 in algorithm 1, to fetch in the first two cache lines containing edge list information for a vertex at an offset of 4 on the work list. Varying offsets and the number of cache lines prefetched gave no additional increase in performance, and attempting to prefetch other data structures in advance, such as the vertex list and work list, reduced performance.

Combining prefetchers, in figure 3(b), shows that over 35% performance improvement can be achieved through software prefetch of the edge list for future vertices, but this still leaves significant performance on the table: the processor is still stalled 80% of the time at this graph scale.

2.4 Opportunity

Although breadth-first searches currently have poor performance due to high L1 cache miss rates, and existing prefetchers are unable to bring significant benefit, the nature of the search algorithm does lend itself to a different type of prefetching. A key feature that distinguishes breadth-first searches from many other data-dependent traversals is that the nodes to be visited are generally known a great deal of time in advance [34]: upon each visit to a node we add



Work List
Vertex ID
Vertex ID
Vertex ID
Prefetching edge IDs brings little performance benefit; we can't use software to prefetch these and information that depends on them too
Vertex List
Start Edge ID
End Edge ID
Prefetching more than 2 cache lines increases the number of instructions, reducing performance
Highest performance from prefetching these cache lines
Edge List
Cache Line
Cache Line
Cache Line
Visited List
V
V
V
V
V
V
V
V
V
Prefetching the visited list requires stalling on edge loads, and inserting a very large number of prefetch instructions, reducing performance

Figure 4: Loads as a result of visiting a node in a breadth first search. Those which can be prefetched in software with any observable benefit are shown with dark shading.

its neighbors to a FIFO queue. This known order can be exploited by a prefetcher with knowledge of the traversal.

However, to determine the addresses to load requires multiple loads of values itself, making the prefetcher non-trivial. For example, the first neighbor of a node, n, in the work list, is obtained by first loading n from the work list, then using the result to index into the vertex list, and finally using this data to index into the edge list. This requires three load requests and a number of address calculations ($\mathit{array\text{-}base} + \mathit{index} \times \mathit{data\text{-}size}$). Thus a prefetcher needs to be able to deal with this inherent complexity, and be able to use data loaded from addresses in memory.

3. A GRAPH PREFETCHER

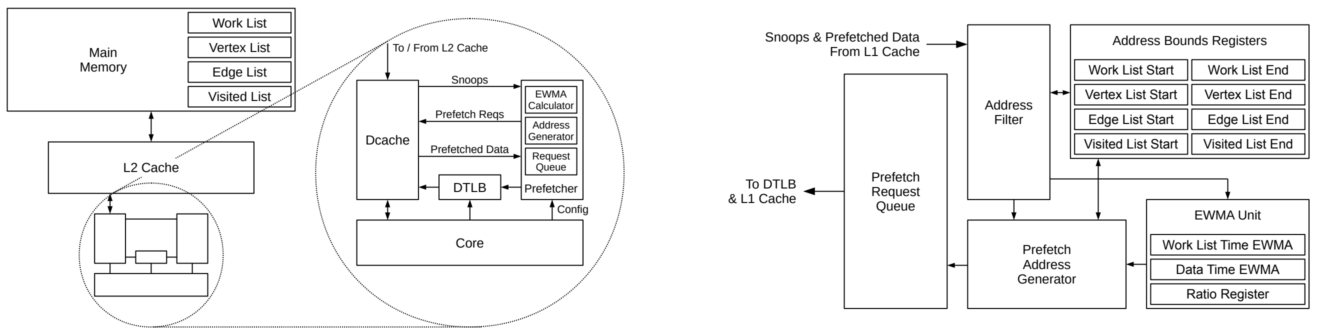
We present a prefetcher for traversals of graphs in CSR format, which snoops loads to the cache made by both the CPU and the prefetcher itself to drive new prefetch requests. Figure 5 gives an overview of the system, which sits alongside the L1. Although it is more common to target the L2 cache, prefetching into the L1 provides the best opportunity for miss-latency reduction, and modern cores include prefetchers at both the L1 and L2 levels [42]. The prefetcher also has a direct connection from the CPU to enable configuration, and another to the DTLB to enable address translation, since our prefetcher works on virtual addresses. Virtual address prefetchers have been proposed previously [43] and implemented in the Itanium 2 on the instruction side [29].

As described in section 2.2, the majority of the benefits come from prefetching the edge and visited lists. However, these are accessed using the work list and vertex list. Therefore, the prefetcher is configured with the address bounds of all four of these structures (needed so that it can calculate addresses from indices), and prefetches issued for each, so a side effect of bringing in the data we care most about is that we also prefetch work list and vertex list information.

3.1 Basic Operation

When the application thread is processing vertex n from the work list, we need to prefetch data for vertex $n + o$, where o is an offset representing the distance ahead that we wish to fetch, based on our expected ratio of fetch versus traversal latencies. Section 3.2 gives more information about the calculation of o. To prefetch all information related to the search, the prefetcher needs to perform a fetch of

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



(a) System overview (b) Prefetcher microarchitecture detail

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

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]`

Large-Vertex Mode

Observation	*Action*
Prefetch `vid = workList[n]`	Prefetch `vertexList[vid]`
Prefetch `eid = vertexList[vid]`	Prefetch `edgeList[eid]` to `edgeList[eid + 8*lineSize - 1]`
Load from `edgeList[eid]` where `(eid % (4*lineSize)) == 0`	Prefetch `edgeList[eid + 4*lineSize]` to `edgeList[eid + 8*lineSize - 1]`
Prefetch `vid = edgeList[eid]`	Prefetch `visited[vid]`
Prefetch `edgeList[vertexList[vid+1]]`	Prefetch `workList[n+1]`

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

for all edges out of this node. Prefetching the first data, `workList[n+o]`, gives the vertex ID, v, of the node and `vertexList[v]` brings in the start edge index. The end edge index (`vertexList[v+1]`) is usually in the same cache line; if not then we estimate that there will be two cache lines of edge data for the vertex. For each edge, e, prefetching `edgeList[e]` gives the node ID of a neighbouring vertex to v, which is also the index into the visited list.

The prefetcher snoops L1 accesses by the core. Observation of an access to `workList[n]` triggers a chain of dependent prefetches for node v, starting with the generation of a prefetch to `workList[n+o]`, which the L1 issues when an MSHR is available. The prefetcher snoops the memory bus and detects the return of the data, which it copies. It can then calculate the address in the vertex list to access, and issue a prefetch for that. Similar actions are performed to generate prefetches for the edge and visited lists.

3.2 Scheduling Prefetches

The key questions with any prefetcher are what to prefetch and when. In the ideal case, we prefetch all the information for the node at offset o from the current node on the work list using equation 1, where $work\_list\_time$ is the average time between processing nodes on the work list and $data\_time$ is the average time to fetch in data required by a single vertex. In other words, all the data for node $n + o$ on the work list will arrive in the cache just in time for it to be required. This technique was proposed by Mowry et al. [31] to set prefetch distances in a static compiler pass. Here we provide a dynamic implementation to make use of runtime data.

$$o * work\_list\_time = data\_time \tag{1}$$

Since $work\_list\_time$ and $data\_time$ can vary wildly both between and within applications, depending on the number of edges out of each node in the graph, we use exponentially weighted moving averages (EWMAs) to estimate their values for any given point in time. Equation 2 gives the generalised EWMA equation. We use $\alpha = 8$ to estimate $work\_list\_time$ and $\alpha = 16$ to estimate $data\_time$, which is more heavily dampened to avoid chance edges in the L2 from reducing the estimate too dramatically. We evaluate the impact of altering α in section 5.

$$avg\_time_{new} = \frac{new\_time + (\alpha - 1) avg\_time_{old}}{\alpha} \tag{2}$$

The EWMA approach works well for graphs of different sizes, as well as those with a highly-variable number of edges per vertex. Due to the bias of breadth-first search [20], a search is more likely to visit larger vertices first and smaller ones towards the end, and thus the search proceeds in phases.

Vertex-Offset Mode When $data\_time > work\_list\_time$

Structure	Configuration
Core	3-Wide, out-of-order, 3.2GHz
ROB	40 Entries
L/S Queues	16 / 16 Entries
Issue Queue	32 Entries
Registers	128 Int, 128 FP
ALUs	3 Int, 2 FP, 1 Mult/Div
Branch Pred.	Tournament with 2048-entry local, 8192-entry global, 2048-entry chooser, 2048-entry BTB, 16-entry RAS
L1 TLB	64 entry, fully associative
L2 TLB	4096 entry, 8-way assoc, 8-cycle hit lat
Page Table Walker	3 simultaneous walks
L1 Caches	32kB, 2-way, 2-cycle hit lat, 12 MSHRs
L2 Cache	1MB, 16-way, 12-cycle hit lat, 16 MSHRs
Memory	DDR3-1600 11-11-11-28 800MHz
Prefetcher	200-entry queue, BFS prefetcher
Operating System	Ubuntu 14.04 LTS

Table 2: Core and memory experimental setup.

then we use equation 3 to prefetch at an offset from the current node on the work list, where k is a multiplicative constant to mitigate the fact that an average always underestimates the maximum time to fetch (2 in our simulations), and also to bias the timeliness of the prefetcher to make it more conservative, ensuring data arrives in the cache before it is requested.

$$o = 1 + \frac{k * data\_time}{work\_list\_time} \qquad (3)$$

The vertex-offset mode is used when prefetching all information for a node on the work list takes more time than the application takes to process each node. In this situation we need to start prefetches for several vertices in advance, in order to ensure the data is in the cache when the program wants to use it.

Large-Vertex Mode On the other hand, when $data\_time < work\_list\_time$, then each vertex takes longer to process than the time to load in all data for the next. Prefetching at a simple offset of 1 from the work list runs the risk of bringing data into the L1 that gets evicted before it is used. In this case we enter large-vertex mode, where we base our prefetches on the progress of computation through the current vertex's edges. As we know the range of edge list indices required, we prefetch 21 cache lines' worth of data, followed by prefetches of stride size 14 upon read observation. In other words, we continually prefetch

```
firstLine = edgeList[idx + 14*lineSize]
```

where *idx* is the current edge list index being processed, and *lineSize* is the size of a cache line. This means we have a constant, small fetch distance in these situations.

We schedule a fetch for the next vertex in the work list when we are four cache lines away from the end of the current vertex's edge list. Although we could use a variable distance based on past history, this access pattern involves comparatively few cache lines at once, so we can afford to be conservative, targeting the case where little work is done between edges, and all other cases will be adequately accommodated as a result.

3.3 Implementation

Given the two modes of operation described in section 3.2, the prefetcher can be implemented as several finite state machines that react to activity in the L1 cache that it snoops. Table 1 shows the events that the prefetcher observes, along with the actions it takes in response.

Configuration Unfortunately it is too complex for the prefetcher to learn the address bounds of each list in memory, therefore the application must explicitly specify these as a configuration step prior to traversing the graph. Although this requires a recompilation to make use of the prefetcher, functionality can be hidden in a library call and for high performance applications this is unlikely to be a major hurdle.

Operation Whenever an address from a load or prefetch is observed, it is compared to each of the ranges to determine whether it is providing data from one of the lists. If so, then an appropriate prefetch can be issued to bring in more data that will be used in the future. For example, when in vertex-offset mode, a load from the work list kicks off prefetching data for a later vertex on the work list using the offset calculated in section 3.2. On the other hand, observation of a prefetch from the work list means that the prefetcher can read the data and proceed to prefetch from the vertex list.

The prefetcher assumes that consecutive values in the vertex list are available in the same cache line, which greatly reduces the complexity of the state machine as it never needs to calculate on data from multiple cache lines at the same time. The downside is that it reduces the capability of the prefetcher in cases where the start and end index of a vertex actually are in different cache lines. In these cases we assume all edge list information will be contained in two cache lines and, if we're in large-vertex mode, then we correct this information once the true value has been loaded in by the application itself.

3.4 Hardware Requirements

Our prefetcher consists of 5 structures, as shown in figure 5a. Snooped addresses and prefetched data from the L1 cache are processed by the address filter. This uses the address bounds registers to determine which data structure the access belongs to, or to avoid prefetching based on L1 accesses to memory outside these structures. We require 8 64-bit registers to store the bounds of the 4 lists when traversing a CSR graph.

Accesses that pass the address filter move into the prefetch address generator. This contains 2 adders to generate up to two new addresses to prefetch, based on the rules shown in table 1. In addition, for prefetches to the work list, it reads the values of the three registers from within the EWMA unit. The output is up to two prefetch addresses which are written into the prefetch request queue.

Alongside the three registers (two EMWAs and one ratio), the EMWA unit contains logic for updating them. The EWMAs are efficient to implement [8], requiring an adder and a multiplier each, and can sample accesses to the lists to estimate their latencies. The ratio register requires a divider, but as this is updated infrequently it need not be high performance.

In total, the prefetcher requires just over 1.6KB of storage (200 $\times$ 64-bit prefetch request queue entries and 11 *times* 64-bit registers), 4 adders, 2 multipliers and a divider. This compares favorably to stride prefetchers (typically 1KB storage) and history-based prefetchers, such as Markov [15], which require large stores (32KB to MBs[12]) of past information to predict the future.

Benchmark	Source	Name
Connected components	Graph 500	CC
Search	Graph 500	Search
Breadth-first search	Boost graph library	BFS
Betweenness centrality	Boost graph library	BC
ST connectivity	Boost graph library	ST
PageRank	Boost graph library	PR
Sequential colouring	Boost graph library	SC

Table 3: Benchmarks.

Graph	Nodes	Edges	Size	Field
s16e10	65,536	1,310,720	10MB	Synthetic
s19e5	524,288	5,242,880	44MB	Synthetic
s19e10	524,288	10,485,760	84MB	Synthetic
s19e15	524,288	15,728,640	124MB	Synthetic
s21e10	4,194,304	83,886,080	672MB	Synthetic
amazon0302	262,111	1,234,877	11MB	Co-purchase
web-Google	875,713	5,105,039	46MB	Web graphs
roadNet-CA	1,965,206	5,533,214	57MB	Roads

Table 4: Synthetic and real-world input graphs.

3.5 Generalised Prefetching

While our prefetcher has been designed to accelerate sequential breadth-first search, it can also be used for a parallel search or other traversals on CSR graphs.

Parallel Breadth-First Search For graphs with many edges and low diameters, it may be beneficial to parallelise the whole breadth-first search on multiple cores [24]. This exchanges the FIFO queue assumed above for a bag, where multiple threads can access different areas of the structure, which are conceptually smaller queues. Our prefetcher works without modification because each individual thread still reads from the bag with a sequential pattern. Therefore we can have multiple cores with multiple prefetchers accessing the same data structure. With multiple threads on a single core, we simply use separate EWMAs to predict per-thread queue access times.

Sequential Iteration Prefetching Another common access pattern for graphs is sequential movement through the vertex and edge data, typically for iterative calculations such as PageRank [35]. For the actual access of the edge and vertex information, a traditional stride prefetcher will work well, however such workloads typically read from a data structure indexed by the vertex value of each edge, which is a frequent, data-dependent, irregular access where a stride prefetcher cannot improve performance. Our prefetcher views this as the same problem as fetching the visited list for breadth-first searches. By reacting to edge list reads instead of work list reads, we can load in the vertex-indexed "visited-like" data at a given offset. This results in the same strategy described for the large-vertex mode.

Other Access Patterns More generally, a similar technique can be used for other data formats and access patterns. The prefetcher relies on inferring progress through a computation by snooping accesses to preconfigured data structures, a technique that can be easily applied to other traversals. For example, a best-first search could be prefetched by observing loads to a binary heap array, and prefetching the first N elements of the heap on each access.

For different access patterns (e.g., array lookups based on hashes of the accessed data [18]), hardware such as the prefetch address queue, which isn't traversal specific, could be shared between similar prefetchers, with only the address generation logic differing. This means that many access patterns could be prefetched with a small amount of hardware.

3.6 Summary

We have presented a prefetcher for traversals of graphs in CSR format. The prefetcher is configured with the address bounds for the graph data structures and operates in two modes to prefetch information in reaction to L1 accesses by the core. Our prefetcher is designed to avoid explicitly stating which vertex traversal starts from. This information is inferred from reads of the lists: we assume that at any point we read the work list, we are likely to read successive elements. This makes the prefetcher both more resilient against temporary changes in access pattern, and also increases its generality: it can also accelerate algorithms which aren't pure breadth-first searches, such as ST connectivity.

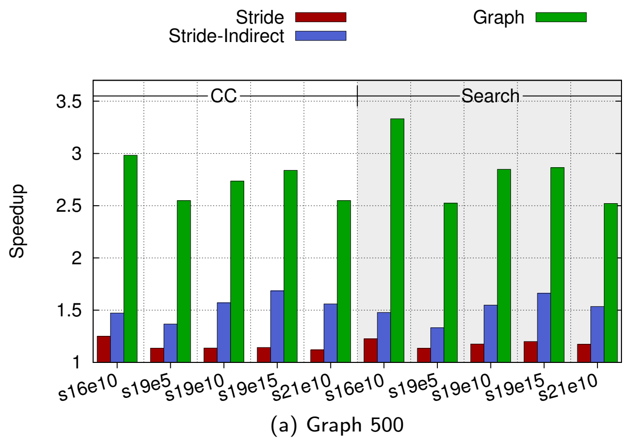
4. EXPERIMENTAL SETUP

To evaluate our prefetcher we modelled the system using the gem5 simulator [4] in full system mode with the setup given in table 2 and the ARMv8 64-bit instruction set. Our applications are derived from existing benchmarks and libraries for graph traversal, using a range of graph sizes and characteristics. We simulate the core breadth-first search based kernels of each benchmark, skipping the graph construction phase.

Our first benchmark is from the Graph 500 community [32]. We used their Kronecker graph generator for both the standard Graph 500 search benchmark and a connected components calculation. The Graph 500 benchmark is designed to represent data analytics workloads, such as 3D physics simulation. Standard inputs are too long to simulate, so we create smaller graphs with scales from 16 to 21 and edge factors from 5 to 15 (for comparison, the Graph 500 "toy" input has scale 26 and edge factor 16).

Our prefetcher is most easily incorporated into libraries that implement graph traversal for CSR graphs. To this end, we use the Boost Graph Library (BGL) [41], a C++ templated library supporting many graph-based algorithms and graph data structures. To support our prefetcher, we added configuration instructions on constructors for CSR data structures, circular buffer queues (serving as the work list) and colour vectors (serving as the visited list). This means that any algorithm incorporating breadth-first searches on CSR graphs gains the benefits of our prefetcher without further modification. We evaluate breadth-first search, betweenness centrality and ST connectivity which all traverse graphs in this manner. To evaluate our extensions for sequential access prefetching (section 3.5) we use PageRank and sequential colouring.

Inputs to the BGL algorithms are a set of real world graphs obtained from the SNAP dataset [25] chosen to represent a variety of sizes and disciplines, as shown in table 4. All are smaller than what we might expect to be processing in a real system, to enable complete simulation in a realistic time-frame, but as figure 2(a) shows, since stall rates go up for larger data structures, we expect the improvements we attain in simulation to be conservative when compared with real-world use cases.



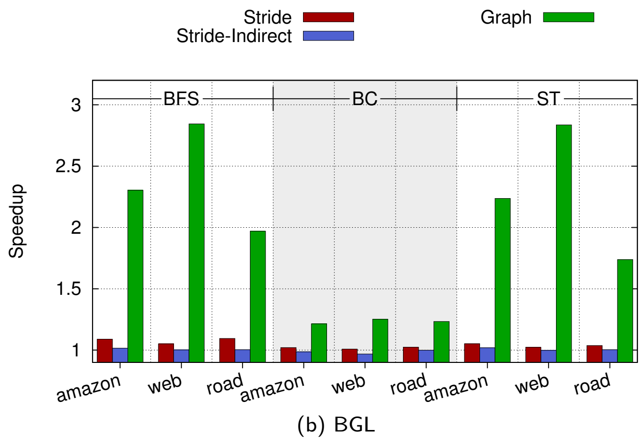


Figure 6: Speedups for our hardware graph prefetcher against stride and stride-indirect schemes.

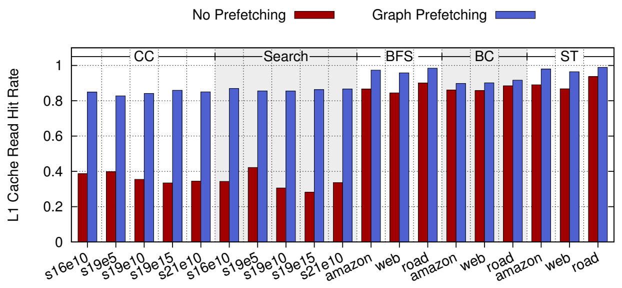


Figure 7: Hit rates in the L1 cache with and without prefetching.

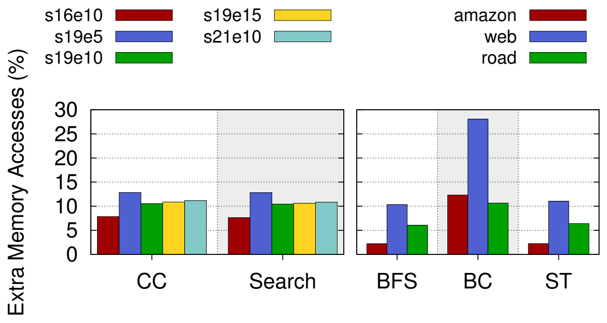


Figure 8: Percentage of additional memory accesses as a result of using our prefetcher.

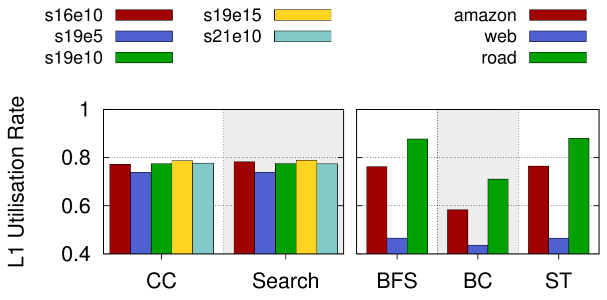


Figure 9: Rates of prefetched cache lines that are used before leaving the L1 cache.

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.

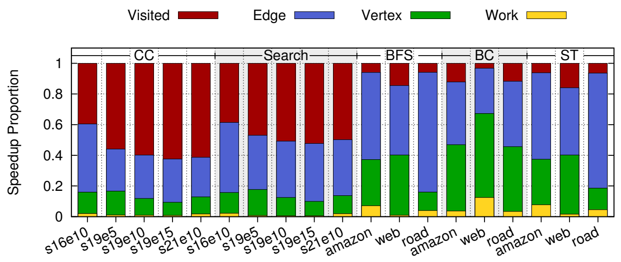


Figure 10: The proportion of speedup from prefetching each data structure within the breadth first search.

L1 Cache Read Hit Rates Our hardware prefetcher boosts L1 hit rates, and even small increases can result in large performance gains. In Graph 500 benchmarks, the baseline hit rates are mostly under 40% and these increase to over 80%. However, in BGL algorithms, baseline hit rates are already high at 80% or more, due to a large number of register spills. These result in loads to addresses that are still in the L1 cache, which are relatively unimportant in the overall run time of the program. Our prefetcher increases the hit rate marginally, but crucially these additional hits are to time-sensitive data from the edge and visited lists, resulting in significant speedups.

Memory Accesses If a prefetcher fetches too much incorrect data from main memory, then a potentially severe inefficiency comes about in terms of power usage. To this end, any prefetcher targeting the reduction of energy usage by reducing stalls needs to keep such fetches to a minimum. Figure 8 shows the percentage increase in memory bytes accessed from main memory for each of the benchmarks and graphs we tested. The average is 9%, which translates into 150MB/s (or approximately 3 cache lines every 4,000 cycles) extra data being fetched. Betweenness Centrality on the web input suffers from the most extra accesses: as it has very low locality and a small number of edges per vertex, the assumption that we will access every edge in a loaded cache line shortly after loading is incorrect. Indeed, this input receives only minor benefit from prefetching visited information, as can be seen in figure 10; without visited prefetching we gain 1.24× for 2% extra memory accesses.

L1 Prefetch Utilisation Figure 9 shows the proportion of prefetches that are used before eviction from the L1 cache. These values are more dependent on timeliness than the number of extra memory accesses: for a prefetched cache line to be read from the L1 it needs to be fetched a short time beforehand. However, even when prefetched data is evicted from the L1 before being used, we still gain the benefits of having it in the L2 cache instead of main memory.

The vast majority of prefetched cache lines are read at least once from the L1 for most test cases. A notable exception is the web input for BGL algorithms, where around half of prefetches aren't used before being evicted from the L1. Still, significant performance improvement is observed for this input; the prefetcher's fetches stay in the L2 and improve performance through avoiding main memory accesses.

Breakdown of Speedup Figure 10 characterises where performance improvement is being observed from within each benchmark. The Graph 500 based benchmarks gain significantly more speedup from visited information than the BGL based algorithms do: this is because Graph 500 stores 64 bit information per vertex (the parent vertex and the component, for search and connected components respectively), whereas the Boost Graph Library code stores a 2 bit colour value for visited information. This means that the Boost code's visited information is more likely to fit in the last level cache. However, as the data size increases, this will not be the case, so for larger graphs a more significant speedup from visited information prefetching will be observed.

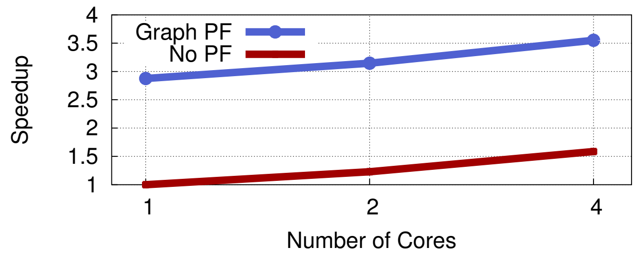


Figure 11: Speedup relative to 1 core with a parallel implementation of Graph500 search with scale 21, edge factor 10 using OpenMP.

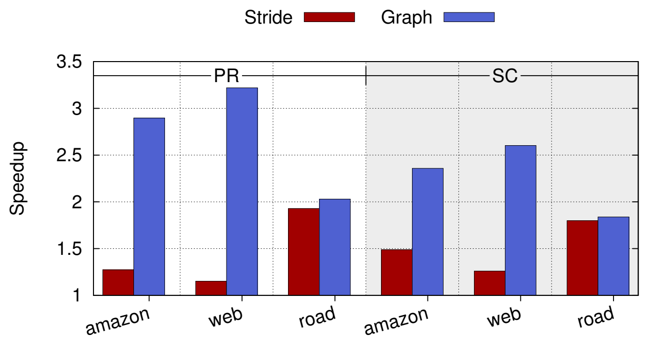


Figure 12: Speedup for different types of prefetching when running PageRank and Sequential Colouring.

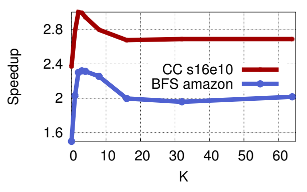
5.3 Generalised Prefetching

We now show how our prefetcher can be used to accelerate other traversals on CSR graphs, as described in section 3.5.

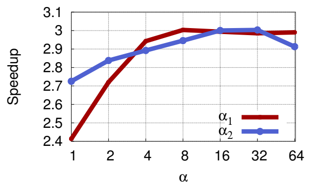
Parallel Breadth-First Search Figure 11 shows the performance of our prefetcher on a parallel implementation of Graph500 search using OpenMP, with a separate prefetcher per core. Each prefetcher works independently, but all are accessing the same data structures. We attain similar speedup to using the sequential algorithm, showing that our prefetcher can aid both single-threaded and multithreaded applications. In addition, the speedups scale at the same rate both with and without prefetching, but prefetching is significantly more beneficial than parallelising the algorithm: 4 cores with no prefetching brings a speedup of 1.6× whereas a single core with our graph prefetcher achieves 2.9×.

Sequential Iteration Prefetching Figure 12 shows the performance of our extension for sequential-indirect access patterns, along with the same stride baseline setup from section 5.1. As this pattern is very predictable, few prefetches are wasted: all of our simulations resulted in under 0.1% extra memory accesses, with an average utilisation rate of 97% for prefetches in the L1 cache.

Notably, though the performance differential between stride and our prefetcher for the web and amazon graphs is very large, it is much smaller for the road-based graph. This reflects the latter's domain: roads tend to have very localised



(a) Varying k



(b) Varying α

Figure 13: Performance as parameters are varied.

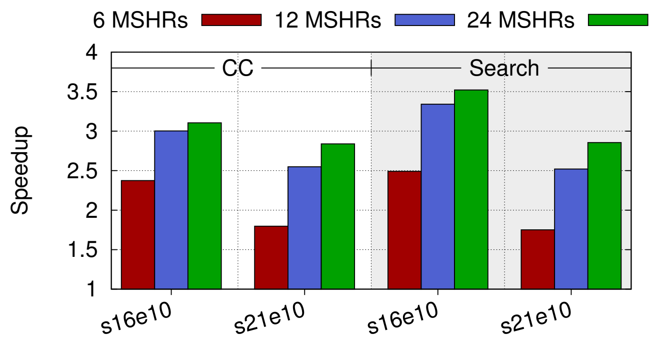


Figure 14: Speedup from the prefetcher with varying numbers of MSHRs for the L1 cache.

structure, so stride fetching for the data-dependent rank indexing still works relatively well.

5.4 Impact of Parameters

We next evaluate the impact of changing the parameters of our prefetcher, showing that there is a sweet spot in the distance weighting factor, and that the EWMA weights and prefetch queue size must be chosen carefully.

Distance Weighting Factor Figure 13(a) shows the performance for two different graphs and benchmarks with varying values for k, the weighting factor from equation 3 in section 3.2. Other applications follow a similar pattern. Both benchmarks see peaks at low values of k (2 and 3 respectively), although there is high performance even with large values. This is because a) we always transition into large-vertex mode at the same time, so only vertex-offset mode is affected, and b) when in vertex-offset mode, even though data is likely to be evicted before it is used, it is likely in the L2 instead of main memory when we need it.

EWMA Weights For our weighted moving averages, we need to strike a balance between a useful amount of history for high performance and ease of computation in setting α. For the latter, we need to set α to a power of two, so that the divide operation is just a shift. Figure 13(b) shows the performance impact for the choice of α for each. For the former, performance is maximal at $\alpha_1 = 8$, and the latter at $\alpha_2 = 32$.

Number of MSHRs Our baseline core cannot use more than six MSHRs due to the size of its load/store queues. However, as the prefetcher can issue its own loads, this no longer becomes the case and lack of available MSHRs is a significant constraint on the number of outstanding prefetches that can be maintained. Figure 14 shows the performance gained with various setups for the L1 cache, showing that 12 MSHRs achieves most of the performance gains, with a little more available with 24, particularly for larger graphs.

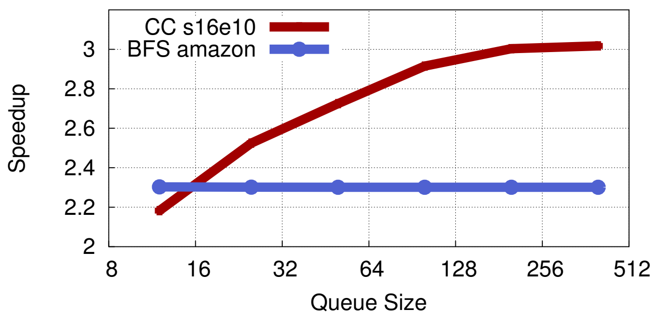


Figure 15: Performance for different queue sizes for two benchmarks and graphs.

Queue Size As prefetches to main memory take a long time to complete, a large queue of addresses is beneficial to deal with the large number of requests created. Figure 15 shows the effect of queue size for Graph 500 on the s16e10 graph and Boost BFS on amazon. Although performance for the former improves with larger queues, more conservatively sized address queues have minor performance impact. Therefore, the storage requirements of the prefetcher could be reduced with only minor performance degradation.

5.5 Summary

We have evaluated our prefetcher on a variety of applications and input graphs, showing that it brings average speedups of 2.2× for breadth-first-search-based algorithms and 2.4× for sequential iteration access patterns. For breadth-first search applications, the prefetcher incurs only 9% additional memory accesses and 70% of the prefetched cache lines are read directly from the L1.

6. RELATED WORK

There has been significant interest in prefetching in the literature, although little work that specifically focuses on graphs. We start with research that targets graphs and broaden out to other types of recently-proposed prefetcher.

6.1 Graph Prefetching

Peled et al. [36] achieve a 1.1× speedup for Graph 500 search using the CSR graph format. Yu et al. [44] target breadth first search with a stride-indirect prefetcher, prefetching visited information based on striding in the edge list. However, performance is limited since the access pattern of a breadth-first search is not a stride-indirect pattern. Spare-register-aware prefetching [22] targets GPUs, detecting load pairs, like the edge list value and associated visited list address, giving an average 10% improvement. Prefetching has also been used for graph algorithms to move data from SSD to main memory. PrefEdge [34] exploits the predictable access patterns of breadth-first search by calculating a lookahead function to determine the memory-mapped addresses that will be loaded in the near future.

6.2 Explicit Hardware Prefetchers

Most current work on hardware support for application-based data fetching involves highly specialised fetcher units. SQRL [19] is an explicitly-configured hardware prefetching accelerator, for collecting values for vectors, key-value structures and BTrees. However, SQRL only works for iterative computations and thus sequential accesses to those three data structures. Similarly, dark silicon accelerators for database indexing [17, 18] look at using explicitly-addressed fetch units for database queries, where the main CPU sleeps on specialised database hash table load requests, which are handled by accelerators.

Al-Sukhni et al. [2] consider prefetching linked-list-style dynamic data structures in hardware. They use special "Harbinger" instructions to specify the exact layout of the structures involved. This paper only looks at sequential traversals, which it predicts in hardware. Fuchs et al. [13] use explicit compiler annotations to control the aggressiveness of the prefetch, and thus serves as an example of software-level control of prefetchers, as is required in our scheme.

6.3 Pointer Fetchers

Several attempts at fetching linked data structures have been made, although these do not work for graphs in CSR format which store array indices, not pointers. Cooksey et al. [6] fetch any plausible-looking address that is loaded as data from a cache line. However, unless the data structure being observed is a list of pointers, all of which will be referenced, such schemes over-fetch dramatically, causing cache pollution and performance degradation. Other work attempts to control this by selectively enabling the technique through compiler configuration [2, 10], but the benefits are still limited to arrays of pointers and linked lists.

Similarly, dependence prefetchers [38] prefetch linked data structures by analysing code at runtime to generate prefetches based on load pairs, and the hardware-based pointer data prefetcher [21] provides a system to prefetch linked lists in hardware, by storing caches of pointer locations. However, these strategies don't generalise to indirect array accesses, due to them exhibiting values being added to base addresses rather than repeated loaded pointers, or complicated conditional access patterns.

6.4 Thread-Based Software Prefetchers

As well as more rigid hardware approaches, considerable work has focused on using software-based approaches to improve performance with prefetching.

Malhotra and Kozyrakis [28] utilise software prefetching on separate threads by inserting explicit instructions into libraries. Kim and Yeung [16] discuss thread-based prefetching, by using "pre-execution threads" generated by the compiler automatically from computation thread code written in C, using profiling information to find frequent misses. These schemes require no extra hardware, but add inefficiency by pulling extra instructions through the memory hierarchy, and wastes a CPU or thread, which is likely to be power inefficient. The lack of good methods for finding cache miss information, and the inherent complexity of inferring which misses are caused by pollution in software also limit the ability to respond to dynamic behaviour of the system.

Advanced architectures have been designed that contain small programmable cores tightly coupled to larger cores to improve load performance. Lau et al. [23] suggest having a small, simple core very tightly coupled to a larger core, with the smaller one running a "helper thread" to fetch data in advance. The tight coupling allows for events to be passed from the main processor cheaply, and the small, low power nature of the secondary core avoids wasting as many large resources on chip, and also makes the idea more efficient. However, this still incurs the inefficiencies of using a general programmable device to do the prefetching; it still has to pull lots of instructions through an expensive memory system. Similarly, Evolve [9] includes a section on using small cores in a many core architecture to do custom caching and prefetching for JPEG encoding.

7. CONCLUSION

Common graph workloads tend to be heavily inefficient due to low performance of the memory system. Traditional hardware prefetchers don't deal well with the accesses required, due to their irregularity, and software prefetching also works poorly, due to its inherent inefficiency and inability to react dynamically. However, since such workloads tend to follow a very small amount of different patterns through the data, we propose using a hardware solution with explicit configuration of the data structure and search pattern to improve throughput and thus performance. We have designed an explicitly controlled prefetcher for breadth first searches and sequential accesses on compressed sparse row graphs, based on snooping reads of a search queue, achieving average speedups of 2.3×, and up to 3.3×, across a range of applications and graph sizes.

The general design paradigm of such a prefetcher is likely to apply to many other workloads: any regular search pattern with the property that we can easily look ahead in the computation, and many different data structures should be amenable to this kind of prefetching. Future work should focus on exploiting other workloads that can be specified to such prefetchers.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant references EP/K026399/1 and EP/M506485/1, and ARM Ltd. Additional data related to this publication is available in the data repository at https://www.repository.cam.ac.uk/handle/1810/254642.

8. REFERENCES

[1] D. Ajwani, U. Dementiev, R. Meyer, and V. Osipov. Breadth first search on massive graphs. In *9th DIMACS Implementation Challenge Workshop: Shortest Paths*, 2006.

[2] Hassan Al-Sukhni, Ian Bratt, and Daniel A. Connors. Compiler-directed content-aware prefetching for dynamic data structures. In *PACT*, 2003.

[3] Jiří Barnat and Ivana Černá. Distributed breadth-first search ltl model checking. *Form. Methods Syst. Des.*, 29(2), September 2006.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), August 2011.

[5] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[6] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS*, 2002.

[7] Ben Coppin. *Artificial Intelligence Illuminated.* Jones and Bartlett Publishers, 2004.

[8] P. Demosthenous, N. Nicolaou, and J. Georgiou. A hardware-efficient lowpass filter design for biomedical applications. In *BioCAS*, 2010.

[9] Jonathan Eastep. Evolve : a preliminary multicore architecture for introspective computing. Master's thesis, MIT, 2007.

[10] E. Ebrahimi, O. Mutlu, and Y.N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA*, 2009.

[11] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2), April 1972.

[12] Babak Falsafi and Thomas F. Wenisch. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture*, 9(1), 2014.

[13] Adi Fuchs, Shie Mannor, Uri Weiser, and Yoav Etsion. Loop-aware memory prefetching using code block working sets. In *MICRO*, 2014.

[14] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *ASPLOS*, 2012.

[15] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ISCA*, 1997.

[16] Dongkeun Kim and Donald Yeung. Design and evaluation of compiler algorithms for pre-execution. *SIGPLAN Not.*, 37(10), October 2002.

[17] Onur Kocberber, Babak Falsafi, Kevin Lim, Parthasarathy Ranganathan, and Stavros Harizopoulos. Dark silicon accelerators for database indexing. In *1st Dark Silicon Workshop (DaSi)*, 2012.

[18] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *MICRO*, 2013.

[19] Snehasish Kumar, Arrvindh Shriraman, Vijayalakshmi Srinivasan, Dan Lin, and Jordon Phillips. SQRL: Hardware accelerator for collecting software data structures. In *PACT*, 2014.

[20] M. Kurant, A. Markopoulou, and P. Thiran. On the bias of BFS (breadth first search). In *ITC*, 2010.

[21] Shih-Chang Lai and Shih-Lien Lu. Hardware-based pointer data prefetcher. In *ICCD*, 2003.

[22] N.B. Lakshminarayana and Hyesoon Kim. Spare register aware prefetching for graph algorithms on gpus. In *HPCA*, 2014.

[23] Eric Lau, Jason E. Miller, Inseok Choi, Donald Yeung, Saman Amarasinghe, and Anant Agarwal. Multicore performance optimization using partner cores. In *HotPar*, 2011.

[24] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA*, 2010.

[25] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[26] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *VLDB*, 2012.

[27] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[28] V. Malhotra and C. Kozyrakis. Library-based prefetching for pointer-intensive applications. Technical report, Computer Systems Laboratory, Stanford University, 2006.

[29] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. In *MICRO*, 2003.

[30] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what cost? In *HotOS XV*, May 2015.

[31] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, 1992.

[32] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the graph 500. *Cray User's Group (CUG)*, May 5, 2010.

[33] Marc Najork and Janet L. Wiener. Breadth-first crawling yields high-quality pages. In *WWW*, 2001.

[34] Karthik Nilakant, Valentin Dalibard, Amitabha Roy, and Eiko Yoneki. Prefedge: Ssd prefetcher for large-scale graph traversal. In *SYSTOR*, 2014.

[35] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

[36] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *ISCA*, 2015.

[37] Dana Ron. Algorithmic and analysis techniques in property testing. *Foundations and Trends in Theoretical Computer Science*, 5(2):73–205, 2009.

[38] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *ASPLOS*, 1998.

[39] Nadathur Satish, Changkyu Kim, Jatin Chhugani, and Pradeep Dubey. Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing. In *SC*, 2012.

[40] V. Shkapenyuk and Torsten Suel. Design and implementation of a high-performance distributed web crawler. In *ICDE*, 2002.

[41] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[42] Vish Viswanathan. Disclosure of h/w prefetcher control on some intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, September 2014.

[43] Chia-Lin Yang and Alvin R. Lebeck. Push vs. pull: Data movement for linked data structures. In *ICS*, 2000.

[44] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. IMP: Indirect memory prefetcher. In *MICRO*, 2015.