Appears at the 1st International Workshop on Architectures for Graph Processing (AGP 2017), held in conjuntion with ISCA 2017

Cache-Guided Scheduling: Exploiting Caches to Maximize Locality in Graph Processing

Anurag Mukkara
MIT CSAIL
anuragm@csail.mit.edu

Nathan Beckmann
CMU SCS
beckmann@cs.cmu.edu

Daniel Sanchez
MIT CSAIL
sanchez@csail.mit.edu

ABSTRACT

Graph processing algorithms are currently bottlenecked by the limited bandwidth and long latency of main memory accesses. On-chip caches are of little help when processing large graphs because their irregular structure leads to seemingly random memory references. However, most real-world graphs offer significant potential locality—it is just hard to predict ahead of time. In practice, graphs have well-connected regions where relatively few vertices share edges with many common neighbors. If these vertices were processed together, graph algorithms would enjoy significant reuse. But finding this cache-friendly schedule is hard from the processor side, which is oblivious to cache contents.

Our insight is that *the cache knows exactly which vertices are cached at any given time, so it is in an ideal position to find a schedule that maximizes locality.* We present our ongoing work on Cache-Guided Scheduling (CGS), a technique that exploits this insight by adding a specialized engine to the last-level cache that dynamically finds a schedule that minimizes cache misses.

We present a limit study of CGS through two idealized implementations. This limit study reveals CGS's large potential: CGS reduces memory accesses by 5.8× gmean on a set of large graphs. Though promising, CGS would incur high overheads if implemented this way. We discuss several paths towards a practical implementation.

1 INTRODUCTION

Graph analytics is an increasingly important workload domain. While graph algorithms are diverse, most have a common characteristic: they are dominated by expensive main memory accesses. Three factors conspire to make graph algorithms memory-dominated. First, these algorithms have low compute-to-communication ratio, as they execute very few instructions (usually less than 10) for each vertex and edge they process. Second, they suffer from poor temporal locality, as the irregular structure of graphs results in seemingly random accesses that are hard to predict ahead of time. Third, they suffer from poor spatial locality, as they perform many sparse accesses to small (e.g., 4 or 8-byte) objects.

Much effort in graph-analytics architectures has focused on specialized processing elements that make compute more efficient [7, 15, 32–34]. While these systems may be beneficial on small graphs that can fit in on-chip storage, on large graphs, main memory accesses dominate performance and energy consumption *even on power-hungry general-purpose processors* [11, 40]. Amdahl's Law thus demands that we focus our efforts on the memory system.

The conventional wisdom has been that graph algorithms have essentially random accesses [5, 17, 24]. This misconception partially stems from limited evaluations that use synthetic, randomly-generated graphs. However, a more detailed analysis reveals that real-world graphs have abundant structure. Specifically, many real-world graphs have power-law degree distributions where a few vertices are much more popular, and accessed more frequently than others [3]. They also have strong community structure corresponding to communities that exist in some meaningful sense in the real world [21]. Graph algorithms thus have abundant locality [4] which, although irregular and difficult to predict, can be exploited.

Most graph processing frameworks use various preprocessing techniques to exploit the structure available in real-world graphs [42, 44]. Preprocessing techniques change the order in which the graph's vertices and edges are stored in memory. Although preprocessing improves locality, it is very expensive. It often takes much longer than the graph algorithm itself, making it impractical for many important scenarios, such as dynamically-updated graphs and graph-processing pipelines where a graph is operated on only once and discarded.

The key idea of this paper is to *exploit the cache hierarchy to find a graph traversal schedule that results in high locality.* The cache has plentiful information about what data it has currently cached and can guide the schedule to yield better locality. Specifically, for most graph algorithms, a vertex is cheap to process when its neighbors are cached, and expensive when its neighbors are not. Hence, scheduling vertices based on what data is in the cache can significantly improve locality. Moreover, as fetching data from memory consumes much more energy than processing it, there is an opportunity to reduce overall system energy by spending more energy in the processor to better schedule memory references.

We demonstrate the potential of this approach, which we call Cache-Guided Scheduling (CGS), through two *idealized* schemes that perform scheduling at the granularity of individual vertices and edges. By decoupling the scheduling algorithm from the associated overheads, these schemes reveal insights about the tradeoffs of CGS and help us analyze its potential benefits. On a set of large real-world graphs, these schemes show up to 5.8× fewer main memory accesses than a locality-agnostic baseline and closely match or outperform state-of-the-art preprocessing techniques. Moreover, our evaluation shows that CGS can be combined with preprocessing to further improve locality. Finally, we present a preliminary hardware design to accelerate the core software routines needed by CGS, discuss the limitations of the current design, and sketch potential ways to sidestep these limitations.

2 BACKGROUND AND MOTIVATION

In this section, we motivate the benefits of Cache-Guided Scheduling and discuss the limitations of previous work. We provide background on software graph processing frameworks (Sec. 2.1) and discuss the impact of scheduling on locality (Sec. 2.2). We show that preprocessing techniques, although beneficial to locality, are very expensive, making them impractical in many important scenarios (Sec. 2.3). Finally, we discuss how previous work on graph-processing architectures fails to alleviate the memory bottleneck of graph processing (Sec. 2.4).

2.1 Software Graph Processing Frameworks

Graphs are a natural abstraction to represent and analyze many kinds of information, such as social network interactions, web page links, road networks, and user ratings in online services [16, 35]. With ever-increasing datasets, there is a growing interest in software graph processing frameworks that improve programmer productivity while achieving good performance and scalability.

Prior work has proposed abstractions and runtimes for graph processing on both shared-memory multicores [20, 31, 37, 41, 42] and distributed systems [13, 14, 23, 25, 29, 38]. Shared-memory graph processing has become very attractive because the limited network bandwidth of distributed systems makes distributed graph processing very inefficient [27], and increasing main memory capacities have made it possible to fit most large real-world graphs on a single machine [41].

Most graph frameworks provide a simple interface that lets application programmers specify algorithm-specific logic for performing operations on graph vertices and edges. The runtime is then responsible for scheduling these operations as defined by the execution model. Many graph frameworks use a Bulk Synchronous Parallel (BSP) [43] model, where the execution is divided into iterations separated by barriers and updates to algorithm-specific data are made visible only at the end of each iteration. The runtime manages a set of active vertices in each iteration and performs application-specific operations on them, until there are no more active vertices or a termination condition (e.g., a number of iterations) is reached.

2.2 Impact of Scheduling on Locality

Many graph algorithms are unordered and the runtime has complete freedom on how to schedule the processing of active vertices in each iteration. The schedule does not affect the correctness of the algorithm, but it has a large impact on locality. All-active algorithms, in which all vertices are processed in each iteration, give more opportunities to perform locality-aware scheduling. Many important algorithms like `PageRank`, `Collaborative Filtering`, and `Label Propagation` are all-active algorithms. Algorithms where only a subset of vertices are active on each iteration, like `BFS`, offer reduced opportunities to improve locality.

State-of-the-art graph processing frameworks like Ligra [41] and GraphMat [42] follow a *vertex-ordered schedule*, a simple technique that achieves spatial locality in accesses to edges but results in hard-to-cache accesses to vertices. Before analyzing the tradeoffs of vertex-ordered scheduling, we first describe the format in which graphs are stored in memory (see Fig. 1).

Assuming the graph has V vertices and E directed edges, there is a destination array of E elements and an offset array of V + 1 elements. The destination array stores the destination vertex id of each directed edge in the graph, after sorting the edges by their source vertex. For each vertex, the offset array stores the offset of its first edge in the destination array. The last element of the offset array is set to V, the number of vertices. Application-specific data, one element for each vertex, is stored in a separate array. For example, in `PageRank`, vertex data stores the score of each vertex. This format, known as Compressed Sparse Row (CSR), is used by many graph processing frameworks [41, 42] since it is simple and space-efficient. CGS can be used with other formats that provide fast lookup of a vertex's neighbors, such as adjacency matrices.

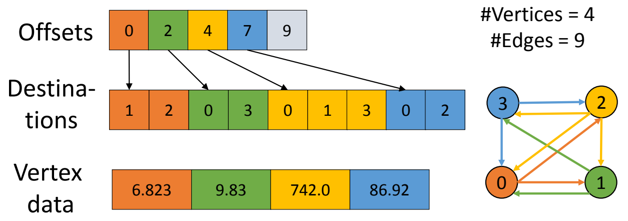


Figure 1: Graph storage format. The destination array stores the destination vertex id of each edge, and the offset array stores the offset of the first edge of each vertex. Application-specific vertex data is stored in a separate array.

Real-world graphs often have tens or even hundreds of edges per vertex, so the edge list is much larger than the vertex data. A vertex-ordered schedule simply processes the active vertices in order of vertex id, and processes all the edges of each vertex consecutively. Processing an edge usually involves accessing the vertex data of both the source (active) vertex and the destination vertex.

This vertex-ordered schedule results in sequential accesses to the large edge list, which yield good spatial locality and can be prefetched well. However, it causes seemingly random accesses to vertex data, as shown in the top half of Fig. 2. If vertex data does not fit in on-chip caches, as is the case for all but the smallest real-world graphs, this will cause many expensive accesses to main memory.

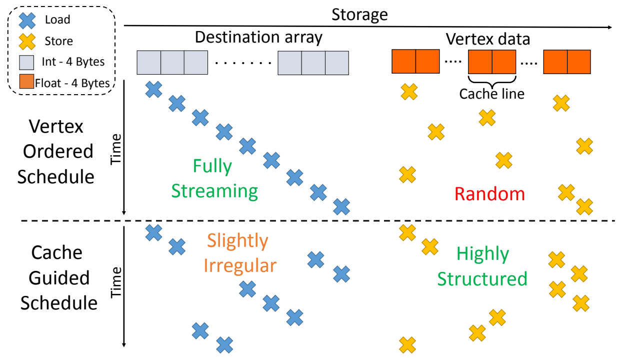


Figure 2: Effect of vertex processing order on locality in accesses to different data structures.

On the graphs we evaluate, accesses to vertex data account for about 90% of total main memory accesses. While latency-hiding techniques such as hardware prefetching [2, 45] can reduce the impact of these accesses on performance, system energy is dominated by main memory accesses. Fig. 3 shows the energy breakdown of `PageRank` on several large graphs using vertex-ordered scheduling (see Sec. 4.1 for methodology). On average, 51% of energy is spent

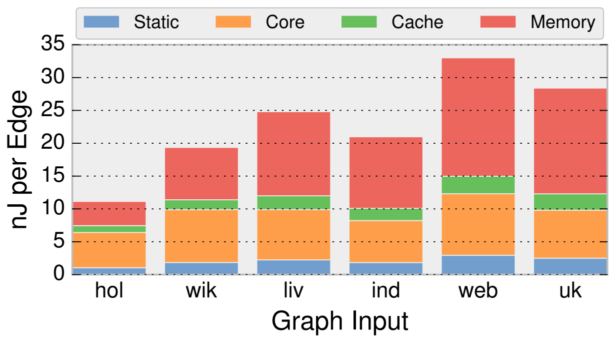


Figure 3: Energy breakdown of `PageRank` on a system with lean out-of-order (Silvermont-like) cores.

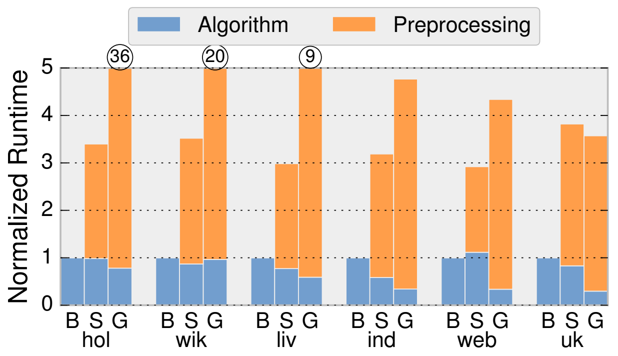


Figure 4: The benefits and overheads of various preprocessing techniques: Baseline (B) with no preprocessing, Slicing (S), and GOrder (G).

on main memory accesses, and this fraction increases to 57% for larger graphs (web) with 100 million vertices.

CGS improves locality in accesses to vertex data by scheduling vertices in an order that exploits the graph's structure. Since vertices are not processed sequentially, this loses some spatial locality in accesses to the edge list (bottom half of Fig. 2). The access to the first edge always touches a new cache line, but accesses to the remaining edges still have spatial locality. Since many real-world graphs have high average degree (more than 10 edges per vertex for the graphs we evaluate), the overall loss of spatial locality is not significant and is offset by the much higher locality we can achieve on accesses to vertex data.

2.3 Graph Preprocessing

Prior work on improving locality in graph algorithms has focused on *graph preprocessing techniques*. These techniques change the graph layout (i.e., the order in which vertices and edges are stored in memory) to improve locality. Some of these techniques are designed to work well with a vertex-ordered schedule, while others also require a different scheduling strategy.

Although preprocessing effectively improves temporal and spatial locality, it is very expensive. It requires doing multiple passes over the entire graph and changing the graph layout to get these benefits. As a result, preprocessing often takes longer than the graph algorithm itself.

Fig. 4 illustrates this tradeoff for `PageRank`. We evaluate two representative preprocessing techniques:

(1) Graph Slicing [15, 42], which partitions the graph into multiple sub-graphs that fit in the cache, improving temporal locality.
(2) GOrder [44], which reorders the vertices in memory through a novel heuristic that is shown to outperform others like sorting vertices by degree, sorting by breadth-first or depth-first traversal order of the graph, etc.

Fig. 4 shows the breakdown of execution time for the preprocessing step and the algorithm itself on several large graphs. We run 20 iterations of single-threaded `PageRank` on an Intel Xeon E5-2658 v3 (Haswell) processor running at 2.2 GHz with 30 MB of last-level cache.

Fig. 4 shows that, although preprocessing significantly reduces `PageRank`'s runtime (by up to 3×), end-to-end runtime is dominated by preprocessing, which often takes an order of magnitude more time than the algorithm. In all cases, baseline execution time with no preprocessing is better than the best preprocessing technique when overhead is considered. Furthermore, preprocessing overhead is magnified for faster algorithms like `Breadth-First Search` that only need to traverse the graph once.

Prior work in this area ignores preprocessing overheads because, it argues, graphs can be reused across many algorithms. But in many real-world situations this is not the case: the graph changes over time or is produced by another algorithm, and is used once or at most a few times [26]. Therefore, while the effectiveness of preprocessing shows that there is abundant structure in graphs, the overheads of preprocessing are unacceptable.

2.4 Accelerators for Graph Analytics

Recent work [7, 15, 32–34] has proposed specialized graph-processing accelerators that use both compute and memory system specialization to achieve large performance and energy efficiency gains.

On the compute side, the proposed accelerators observe that graph algorithms use simple vertex and operations that are similar across many algorithms. Hence, they use specialized pipelines to perform these operations cheaply, and either leverage reconfigurable logic or introduce limited programmability to support multiple algorithms.

While specialized graph-processing engines are effective on small graphs that fit on chip, their effectiveness is limited on large graphs, where main memory accesses dominate performance and energy even with general-purpose cores. Fig. 5 demonstrates this by showing the impact of core configuration on system energy, both with and without preprocessing. Compared to lean out-of-order cores, wide out-of-order (Nehalem-like) cores increase compute energy but barely help performance, because the system is bandwidth-bound. Specialized pipelines reduce compute energy significantly, by more than 10× [15]. While this improves system energy, it increases the relative importance of main memory accesses, which now become the main cost. Preprocessing techniques (shown on the right of Fig. 5) improve locality and reduce memory energy, but even then, memory accesses dominate energy consumption

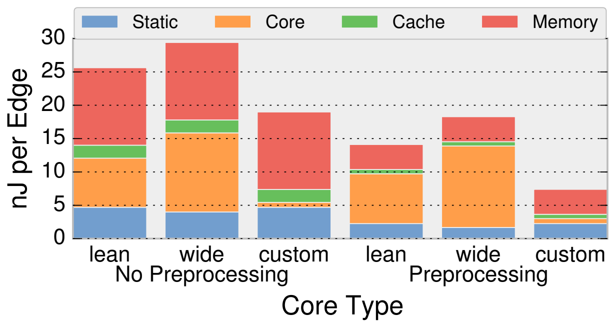


Figure 5: Energy breakdown of `PageRank` **on various core configurations, without and with GOrder preprocessing, averaged across all graph inputs.**

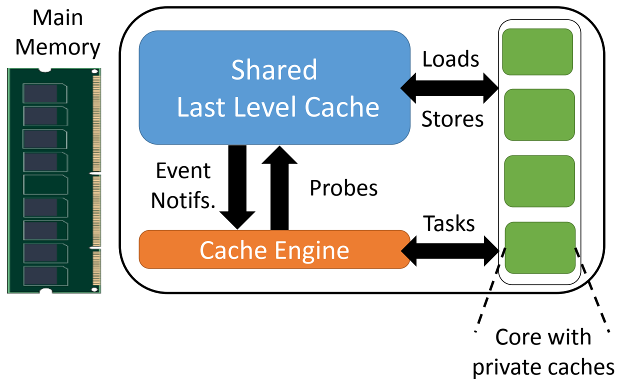


Figure 6: Key hardware components of CGS. The cache engine is a simple core that monitors cache contents and reacts to insertion and eviction events to find a cache-friendly schedule. The main processor, which is general-purpose, enqueues and dequeues tasks from the cache engine.

when specialized compute engines are used. Hence, we focus on techniques to use the memory system better.

Prior work that specializes the memory system for graph processing follows two broad techniques. First, such designs often tune the memory hierarchy to the needs of graph applications, e.g., by using separate scratchpads to hold vertex and edge data and matching their word sizes to the object sizes [34], or by adding a large on-chip eDRAM scratchpad to hold larger graphs [15] than is possible with SRAM. Second, prior work has proposed near-data processing designs [1, 12, 30] that execute most graph-processing operations in logic close to main memory, reducing the cost of memory accesses.

Cache-Guided Scheduling complements this prior work by leveraging caches to dynamically find a schedule that makes better use of limited on-chip capacity. Although we describe CGS in the context of a general-purpose system with lean cores, we expect it to be even more beneficial with specialized processors.

3 CGS DESIGN

In this section, we sketch CGS's hardware support and hardware-software interface. Note that *this is just an implementation sketch and not the final design*, which will require a more comprehensive study of design alternatives and overheads. We then discuss the two idealized schemes that let us study the potential of CGS.

3.1 Hardware Support

Graph processing is memory-bound, so it is possible to spend extra instructions to perform locality-aware scheduling without hurting end-to-end performance. However, implementing CGS completely in software would incur prohibitive overheads. Hence, we propose simple hardware support to accelerate CGS.

Cache engine: As shown in Fig. 6, we add a programmable *cache engine* at the last-level cache that is responsible for monitoring cache contents and finding a cache-friendly schedule.

We envision a simple, specialized core in the style of MAGIC [19] or Typhoon [36] that can inspect the cache's contents and react to events like cache line insertions and evictions. Although we propose using the cache engine to enable cache-friendly graph analytics, it could be used to accelerate other functions.

The primary function of the cache engine is to maintain a *worklist* of tasks to feed to cores. A task consists of processing a set of active vertices or edges in the graph. The *worklist* ranks the available tasks by assigning each task a score that indicates how beneficial it is to execute. For example, when the task involves processing a vertex, the score could be how many of its neighbors are currently cached. We call the metric used to compute task scores the *scoring metric*.

The cache engine sends the tasks with the highest scores to cores for processing. We assume small hardware task queues between the cache engine and the cores, similar to Active Messages [10] or Carbon [18].

In our current, idealized implementation of CGS, the worklist stores all active vertices and edges, which is expensive. Sec. 5 discusses potential strategies to use a small worklist while retaining most of the benefits of an unbounded worklist. We envision a design where the cache engine adaptively expands the worklist by exploring the graph and bringing in graph vertices when the worklist size falls below a certain threshold.

The scoring metrics that we use require simple integer arithmetic operations. Hence, this core only needs simple functional units and we expect its area footprint to be a small fraction of the multi-megabyte last-level caches in current processors. To compute these scores, the cache engine uses two interfaces to the last-level cache: probes and cache event notifications, which we detail below.

Probes: The cache engine routinely checks whether an address is present in the last-level cache. A probe is simply a cache tag lookup that has no side effects: no main memory accesses are triggered if the address is not cached, and cache replacement state is not updated if the line is cached. Probe operations can reuse existing tag ports, since last-level caches are multi-banked and have plentiful tag array bandwidth.

The cache engine uses probes to rank tasks, e.g., by checking the neighbors of a vertex to see how many are present in the cache.

Cache event notifications: As software executes tasks, the cache's contents keep changing as new data is brought into the cache and old data is evicted. To keep the worklist's scores up-to-date, the cache engine is notified of evictions and insertions. A notification consists of a line address and an event type.

The cache engine maintains metadata about which tasks in the worklist are waiting on a particular cache line and, on a notification, queries this metadata to see whether any score needs to be updated. For example, when the worklist consists of active vertices, the cache engine maintains an auxiliary hash table of neighbors that are not yet in the cache. On each insertion, the cache engine queries the hash table to find whether any task scores should be updated. To maintain this metadata, the cache engine may perform additional memory accesses to the graph (e.g., in our example, to fetch the edge list of each vertex it inserts into the worklist). Thus the tradeoff is between the extra memory accesses to the graph and keeping the scores accurate in order to improve the locality of accesses to vertex data.

Overheads and idealizations: The main overheads of this design are *(1)* the additional processing performed by the engine to build the worklist and keep task scores updated, and *(2)* the space overheads of the worklist and the auxiliary metadata, which will take some of the available cache capacity. In our limit study, we ignore both these overheads, which allows us to study the potential benefits of CGS. Though we have not modeled these overheads in detail, given experimental data we expect them to be significant. Therefore, in Sec. 5 we discuss some potential strategies to reduce them. We now describe two variants of CGS and evaluate them in Sec. 4 with these idealizations.

3.2 Cache-Guided Scheduling of Vertices (CGS-V)

In CGS-V, each task processes one active vertex in the graph. At the beginning of an iteration, all active vertices are inserted into the worklist. Tasks are then dequeued from the worklist and processed until the worklist is empty. Processing a task involves sequentially processing all the edges with the vertex as the source.

Choosing the right scoring metric to rank tasks in the worklist can have a significant impact on the performance of CGS-V. Empirically, we have found that ranking vertices by hit ratio (e.g., number of cached neighbors / number of neighbors) works well. This is intuitive, since selecting the vertices that will cause the highest hit ratio will tend to maximize the cache's performance. In Sec. 4.3 we show that this approach outperforms other intuitive scoring metrics, such as ranking by most-cached or fewest-uncached neighbors.

3.3 Cache-Guided Scheduling of Edges (CGS-E)

In CGS-E, each edge in the graph maps to a task rather than a vertex. The main difference between CGS-V and CGS-E is that in CGS-V, all the edges with a common source vertex are processed sequentially before moving to the next vertex. This property is not guaranteed in CGS-E since each edge is scheduled independently. CGS-E manages computation at a finer granularity than CGS-V and has more opportunities to improve locality. However, it has higher overheads since edges outnumber vertices by more than an order of magnitude in most real-world graphs.

The scoring metric tracks how many of the two endpoint vertices of an edge are currently in the cache. Hence, it can take only three values: 0 if none of the vertices are cached, 1 if either source or destination is cached, and 2 if both vertices are cached. Since the goal is to maximize hits, a higher score is better.

Similar to CGS-V, all the active edges are added to the worklist at the start of each iteration and processed in the order of their scores. If the initial score is 2, the edge is immediately processed since doing so causes no cache misses. Otherwise, the edge is inserted into the worklist to be processed later, when it is likely to have a better score.

When the worklist runs out of edges with score 2, it executes an edge with score 1. Among all the edges with a score of 1, it chooses the edge, one of whose endpoints is connected to the most number of edges with score 1 in the worklist. The intuition is that processing this edge brings in new data into the cache, that increases the score of the maximum number of edges from 1 to 2. If there are no edges with a score of 1, the worklist executes an edge with score 0, breaking ties in a similar manner.

4 EVALUATION

We now present our evaluation methodology including the graph algorithms, datasets, and simulation infrastructure. We then demonstrate the benefits of the vertex- and edge-based cache-guided scheduling schemes compared to a locality-agnostic baseline. We end the section with several sensitivity studies.

4.1 Methodology

Applications: We evaluate CGS on `PageRank` and `Collaborative Filtering` (CF), which are both all-active algorithms but with small (16-byte) and large (256-byte) object sizes respectively. We use the implementations from Ligra [41] as our locality-agnostic baselines with vertex-ordered scheduling. All-active algorithms are generally executed for a fixed number of iterations or until a convergence condition is reached. However, to avoid long simulation times we only report results for the first iteration. This does not significantly affect the relative performance of different schemes as the algorithms that we evaluate have the same memory reference pattern in each iteration and the graphs we use are so large that there is negligible reuse across iterations.

We implement several scheduling schemes in our runtime without changing the application code. This makes our techniques transparent to changes in application-level logic and makes it easier to extend them to other graph algorithms. In particular, we expect similar gains for other all-active algorithms like `Triangle Counting`, `Label Propagation`, etc. In addition, CGS also works with traversal algorithms like `BFS` or asymmetric algorithms like `Incremental PageRank`. For such algorithms, CGS can reuse the active vertex bit vector to rank only the active vertices.

Graph datasets: We use several large real-world graphs from various domains like social networks, web crawls, movie ratings, etc., detailed in Table 1. With 16-byte and 256-byte objects for `PageRank`, `Collaborative Filtering` respectively, the graph sizes are much larger than the last-level cache size. We represent graphs in memory in Compressed Sparse Row (CSR) format, similar to Ligra. Since some of the graphs we use were already preprocessed, we randomize their vertex ids before running the graph algorithm.

Application	Graph	Vertices (M)	Edges (M)	Description
PageRank	hol	1.1	113	Hollywood actor network [8]
	wik	3.5	45	Wikipedia page links [8]
	liv	4.8	69	LiveJournal follower network [8]
	ind	7.4	194	Crawl of Indochina network [8]
	uk	19	298	Crawl of .uk domain [8]
	web	118	1020	Cross-domain crawl by Webbase [8]
Collaborative Filtering	nfl	0.50	100	Netflix movie ratings [6]
	yms	0.55	61	Yahoo Music ratings [9]

Table 1: Real-world graph datasets used.

L1 cache	32 KB 8-way set-associative
L2 cache	8 MB 16-way hashed set-associative
Replacement policy	LRU with L2 bypassing for streaming data

Table 2: Configuration of the simulated cache hierarchy.

Simulation infrastructure: We developed an in-house cache simulator to get first-order estimates of the cache performance of different schemes. We insert hooks into the graph processing runtime that profile accesses to all the important data structures through the cache simulator. While the cache simulator does not do detailed instruction-level simulation, we verified that the memory and cache statistics that it reports are within 5% of those reported by zsim [39]. This simulator is simpler and faster than zsim, so it accelerates design-space exploration but does not provide end-to-end performance results. We simulate the cache hierarchy shown in Table 2 and report last-level cache misses only. Since graph processing is memory latency-bound even when using out-of-order core processors [4], reduction in main memory accesses should translate to improved performance.

To get the system energy breakdown numbers in Sec. 2, we simulate (using zsim [39]) a system with 32 lean out-of-order (Silvermont-like) cores, 16MB of last-level cache, and 6 DDR3 memory controllers with 12.8GB/s bandwidth per controller. We use McPAT [22] to derive the energy numbers of chip components at 22 nm, and Micron DDR3L datasheets [28] to compute main memory energy.

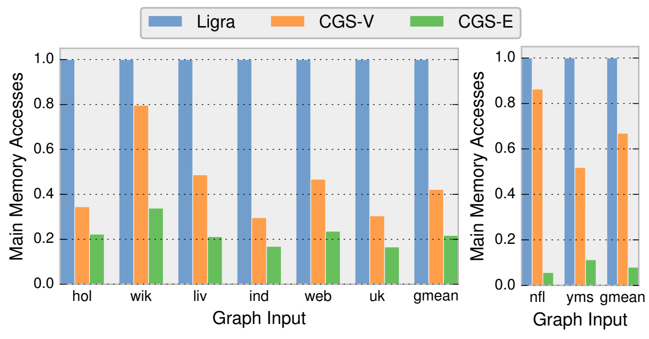
4.2 Benefits of Cache-Guided Scheduling

Fig. 7 shows the memory accesses of cache-guided scheduling schemes compared to Ligra on `PageRank` and `Collaborative Filtering` (CF). Both cache-guided scheduling schemes show significant reduction in main memory accesses with CGS-E performing better than CGS-V in general. On average across both applications, CGS-V and CGS-E reduce memory accesses by 2.1× and 5.8×, respectively.

For `PageRank`, CGS-V and CGS-E achieve 2.4× and 4.6× lower memory accesses than Ligra on average, respectively. The gap between CGS-V and CGS-E widens for `Collaborative Filtering`: while CGS-V only gets up to 1.9× lower accesses (1.5× on average), CGS-E achieves up to 17× reduction (12× on average). Due do the larger objects sizes and high average vertex degree for `Collaborative Filtering`, on average, a few kilobytes of data are accessed in processing a single vertex. By scheduling such large chunks of work at once, CGS-V loses significant opportunities for fine-grain adaptation to cache contents.

4.3 Sensitivity Studies

For the results in the rest of the section, we focus on `PageRank`.



(a) PageRank. **(b) CF.**

Figure 7: Main memory accesses of cache-guided scheduling schemes normalized to Ligra (lower is better).

Cache-guided scheduling with preprocessing: Fig. 8 compares CGS with preprocessing techniques and shows that they are complementary. We compare Ligra, Slicing, GOrder (described in Sec. 2.3), and CGS-V and CGS-E both with and without GOrder preprocessing. While CGS-E's gains are similar to using preprocessing with Ligra, preprocessing the graph before applying CGS-E further reduces main memory accesses. Thus, in addition to matching the performance of state-of-the-art preprocessing techniques without their often-impractical overheads, CGS can also be combined with preprocessing when practical to reap additional benefits.

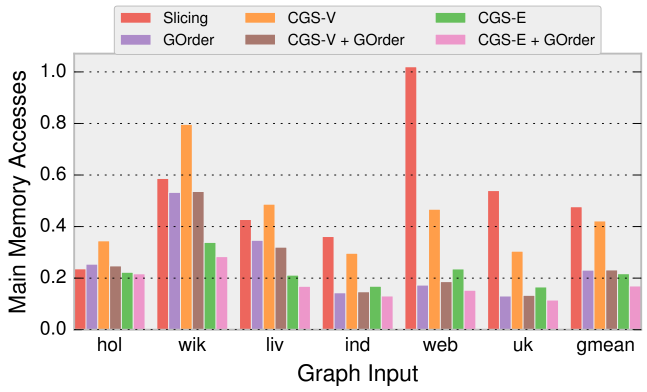


Figure 8: Effect of applying GOrder preprocessing on the graph before running the main algorithm using CGS. Main memory accesses are normalized to Ligra (lower is better).

Breakdown of accesses by data structure: Fig. 9 shows the breakdown of accesses to the major data structures: edge data, source vertex data, and destination vertex data. While Ligra achieves great locality in accesses to edge and source vertex data, it suffers from many random accesses to destination vertices. Preprocessing improves the locality of accesses to destination vertex data without impacting accesses to source vertex data. Both CGS-V and CGS-E hurt locality of accesses to source vertices somewhat, but improve that to destination vertices much more, giving a large overall reduction in cache misses.

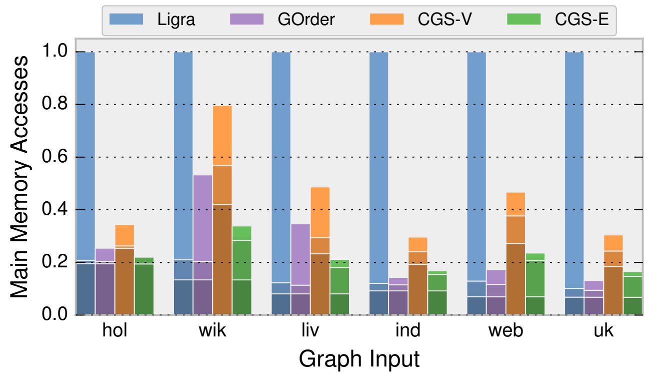


Figure 9: Breakdown of main memory accesses between edge data, source vertices, and destination vertices (from bottom to top). All schemes are normalized to Ligra (lower is better).

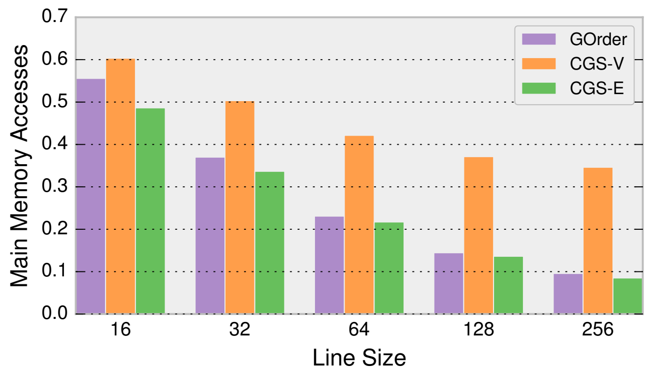


Figure 10: Impact of cache line size. Main memory accesses are normalized to Ligra (lower is better).

Impact of cache line size: Fig. 10 shows the impact of line size on the benefits of CGS, by varying the line size from 16 bytes (which matches the object size of `PageRank`) to 256 bytes. At each line size, we show the reduction in memory accesses over Ligra averaged across all graphs. CGS-E slightly outperforms preprocessing at all line sizes and its benefits increase with larger line sizes. CGS-V performs similarly to the other techniques at small line sizes, but CGS-V's benefits increase more slowly with line size, so with large lines the gap with CGS-E and GOrder broadens.

Scoring metric for CGS-V: Fig. 11 shows the impact of scoring metric on the benefits of CGS-V. We consider the following scoring metrics: most hits (i.e., most neighbors cached), fewest misses (i.e., fewest neighbors uncached) and highest hit ratio (our default metric). For each scoring metric, we show the reduction in memory accesses of CGS-V over vertex-ordered scheduling.

The first two metrics perform almost the same on average. For graphs with lower baseline cache hit ratio (hol, wik, liv) minimizing misses performs better and the trend reverses for graphs with higher baseline cache hit ratio (ind, web, uk). The third metric, maximizing hit ratio, consistently outperforms the other two metrics and achieves up to 29% (14% on average) lower memory accesses.

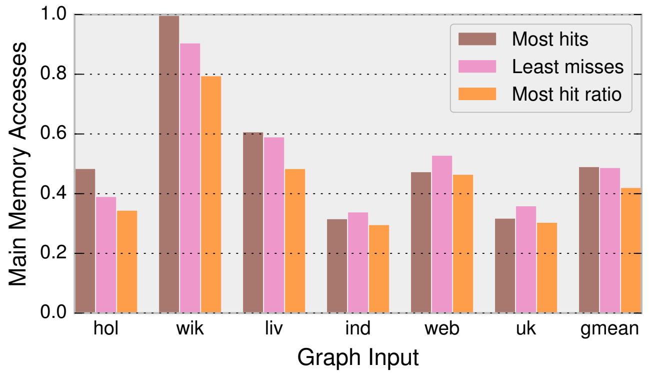


Figure 11: Impact of scoring metric of CGS-V. Main memory accesses are normalized to Ligra (lower is better).

We also evaluated the effect of valuing hits and misses differently in the scoring metric. Specifically, we evaluated the metric:

$$a \cdot hits - (1 - a) \cdot misses \text{ for } a \in [0, 1]$$

Note that $a = 1$ is equivalent to maximizing hits and $a = 0$ is equivalent to minimizing misses. We observed that the value of a that performs best changes across graphs, and its performance closely matches the performance of the hit ratio metric.

5 REDUCING THE OVERHEADS OF CGS

In this section, we discuss some potential ways to reduce the overheads of CGS as it currently implemented.

Reducing storage overheads: In the current CGS design, the major overhead comes from the storage required for the tasks in the worklist and the auxiliary metadata needed to track scores accurately. In particular, maintaining all the active vertices in the worklist is very expensive. To mitigate this, we need to be able to store only a small fraction of the active vertices in the worklist and still capture most of the locality benefits of CGS. Our initial experiments suggest that for a small worklist to be effective, the order in which vertices are added to the worklist is crucial. For example, inserting vertices in the order of their id would be ineffective as the algorithm would end up exploring disjoint regions of the graph simultaneously. Preliminary results suggest that filling the worklist by exploring the graph in a depth-first fashion is a plausible way to capture the locality achieved with an unbounded worklist while maintaining a small worklist. We will quantify this tradeoff in depth and explore alternative policies in future work.

Reducing processing overheads: Beyond storage overheads, scheduling costs must also be kept small, since each task is a few tens of instructions. A simple approach would be to fix the scheduling logic in hardware, but this would forgo the programmability of the cache engine, making the system hard to adapt to new algorithms. Before rushing towards a fixed-function scheduler, we should consider algorithmic simplifications to CGS.

First, since nearby vertices tend to be explored at the same time, *grouping vertices* as they are explored can lower overheads while preserving locality. Processing groups of vertices as a single task and only tracking the scores for a few vertices from each group greatly reduces the number of vertices that must be tracked. Since the work involved in processing a task correlates with the number of edges rather than vertices, it might be important to balance the number of edges across groups of vertices.

Second, some vertices have many neighbors—often hundreds or even thousands—so it may be sufficient to sample a few neighbors to infer their hit rate. This *sampling neighbors* strategy would reduce the number of cache lookups for each vertex.

Third, some cache lines are accessed by many tasks, when the vertices it holds have a large in-degree. When such lines are evicted or inserted into the cache, the scores of most tasks in the worklist are updated by the same value. Thus, *skipping updates* in such cases does not significantly change the relative order of tasks in the worklist. The choice of the degree threshold that identifies such large vertices is crucial for this optimization to be effective.

Finally, we could devise more *efficient strategies to update scores* than by tracking insertions and evictions. For example, we could sample evictions and insertions, or devise an aging mechanism that achieves similar effects. We note that, although updating scores is not cheap, we have found that it is necessary: disabling updates makes CGS lose a large fraction of its benefits.

6 CONCLUSION

Graph processing suffers from excessive main memory accesses, causing poor performance and energy efficiency on current systems. We have presented our ongoing work in cache-guided scheduling, a novel technique that improves locality by exploiting information about the contents of the cache hierarchy to improve locality. We demonstrated the potential of cache-guided scheduling through two idealized schemes that reduce memory accesses by 5.8× gmean over a locality-agnostic baseline. We will next work on developing a practical implementation of cache-guided scheduling with the required hardware support.

ACKNOWLEDGMENTS

We sincerely thank Maleen Abeydeera, Mark Jeffrey, Hyun-Ryong Lee, Suvinay Subramanian, Po-An Tsai, Victor Ying, Guowei Zhang, and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grant CAREER-1452994, a Samsung research grant, and a grant from the Qatar Computing Research Institute.

REFERENCES

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. ISCA-42*, 2015.
[2] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proc. ICS'16*, 2016.
[3] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, 1999.
[4] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an Ivy Bridge server," in *Proc. IISWC*, 2015.
[5] S. Beamer, K. Asanovic, and D. Patterson, "Reducing Pagerank Communication via Propagation Blocking," in *Proc. IPDPS*, 2017.
[6] J. Bennett and S. Lanning, "The Netflix prize," in *KDD cup*, 2007.
[7] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph processing framework on FPGA—a case study of breadth-first search," in *Proc. FPGA'16*, 2016.
[8] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM TOMS*, vol. 38, no. 1, 2011.
[9] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The Yahoo! Music Dataset and KDD-Cup'11." in *KDD Cup*, 2012.
[10] T. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proc. ISCA-19*, 1992.
[11] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, and S. Katti, "Parallel graph processing on modern multi-core servers: New findings and remaining challenges," in *Proc. MASCOTS-24*, 2016.
[12] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proc. PACT-24*, 2015.
[13] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. OSDI-10*, 2012.
[14] J. Gonzalez, R. Xin, A. Dave, D. Crankshaw, M. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. OSDI-10*, 2012.
[15] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. MICRO-49*, 2016.
[16] K. Kambatla, G. Kollias, V. Kumar, and A. Grama, "Trends in big data analytics," *JPDC*, vol. 74, no. 7, 2014.
[17] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with milk," in *Proc. PACT-25*, 2016.
[18] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *Proc. ISCA-34*, 2007.
[19] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo *et al.*, "The Stanford FLASH multiprocessor," in *Proc. ISCA-21*, 1994.
[20] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: large-scale graph computation on just a PC," in *Proc. OSDI-10*, 2012.
[21] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Statistical properties of community structure in large social and information networks," in *Proc. WWW'08*, 2008.
[22] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. MICRO-42*, 2009.
[23] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, 2012.
[24] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, 2007.
[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser *et al.*, "Pregel: a system for large-scale graph processing," in *Proc. SIGMOD'10*, 2010.
[26] A. McGregor, "Graph stream algorithms: a survey," *ACM SIGMOD Record*, vol. 43, no. 1, 2014.
[27] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at what COST?" in *Proc. HotOS-15*, 2015.
[28] Micron, "1.35V DDR3L power calculator (4Gb x16 chips)," 2013.
[29] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proc. SOSP-24*, 2013.
[30] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks," in *Proc. HPCA-23*, 2017.
[31] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. SOSP-24*, 2013.
[32] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe *et al.*, "Graphgen: An FPGA framework for vertex-centric graph computation," in *Proc. FCCM-22*, 2014.
[33] T. Oguntebi and K. Olukotun, "GraphOps: A dataflow library for graph analytics acceleration," in *Proc. FPGA'16*, 2016.
[34] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns *et al.*, "Energy efficient architecture for graph analytics accelerators," in *Proc. ISCA-43*, 2016.
[35] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem *et al.*, "The tao of parallelism in algorithms," in *Proc. PLDI*, 2011.
[36] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-level shared memory," in *Proc. ISCA-21*, 1994.
[37] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric graph processing using streaming partitions," in *Proc. SOSP-24*, 2013.
[38] S. Salihoglu and J. Widom, "GPS: a graph processing system," in *Proc. SSDBM-25*, 2013.
[39] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. ISCA-40*, 2013.
[40] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan *et al.*, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proc. SIGMOD'14*, 2014.
[41] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. PPoPP*, 2013.
[42] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi *et al.*, "GraphMat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, 2015.
[43] L. G. Valiant, "A bridging model for parallel computation," *Comm. ACM*, vol. 33, no. 8, 1990.
[44] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup Graph Processing by Graph Ordering," in *Proc. SIGMOD'16*, 2016.
[45] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proc. MICRO-48*, 2015.