Appears in the Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018

Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling

Anurag Mukkara* Nathan Beckmann† Maleen Abeydeera* Xiaosong Ma‡ Daniel Sanchez*
**MIT CSAIL* *†CMU SCS* *‡QCRI*
{anuragm, maleen, sanchez}@csail.mit.edu, beckmann@cs.cmu.edu, xma@hbku.edu.qa

***Abstract*—Graph processing is increasingly bottlenecked by main memory accesses. On-chip caches are of little help because the irregular structure of graphs causes 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 processing would enjoy significant data reuse. Hence, a graph's *traversal schedule* largely determines its locality.**

This paper explores *online* traversal scheduling strategies that exploit the community structure of real-world graphs to improve locality. Software graph processing frameworks use simple, locality-oblivious scheduling because, on general-purpose cores, the benefits of locality-aware scheduling are outweighed by its overheads. Software frameworks rely on offline preprocessing to improve locality. Unfortunately, preprocessing is so expensive that its costs often negate any benefits from improved locality. Recent graph processing accelerators have inherited this design. Our insight is that this misses an opportunity: Hardware acceleration allows for more sophisticated, online locality-aware scheduling than can be realized in software, letting systems significantly improve locality without any preprocessing.

To exploit this insight, we present bounded depth-first scheduling (BDFS), a simple online locality-aware scheduling strategy. BDFS restricts each core to explore one small, connected region of the graph at a time, improving locality on graphs with good community structure. We then present HATS, a hardware-accelerated traversal scheduler that adds just 0.4% area and 0.2% power over general-purpose cores.

We evaluate BDFS and HATS on several algorithms using large real-world graphs. On a simulated 16-core system, BDFS reduces main memory accesses by up to 2.4× and by 30% on average. However, BDFS is too expensive in software and degrades performance by 21% on average. HATS eliminates these overheads, allowing BDFS to improve performance by 83% on average (up to 3.1×) over a locality-oblivious software implementation and by 31% on average (up to 2.1×) over specialized prefetchers.

***Index Terms*—graph analytics, multicore, caches, locality, scheduling, prefetching.**

I. 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-bound. First, these algorithms have low compute-to-communication ratio, as they execute very few instructions (usually few 10s) for each vertex or 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.

The conventional wisdom has been that graph algorithms have essentially random accesses [26, 31]. This misconception partially stems from limited evaluations that use synthetic, randomly generated graphs. However, a more detailed analysis reveals that many real-world graphs have abundant structure. Specifically, they have strong community structure corresponding to communities that exist in some meaningful sense in the real world [29]. Many real-world graphs are also scale-free, i.e., they have skewed degree distributions where a small subset of vertices are much more popular, and hence accessed more frequently, than others [6]. Graph algorithms thus offer significant potential locality [7], though it is irregular and difficult to predict.

This locality can be exploited by controlling the *traversal schedule*, i.e., the order in which vertices and edges of the graph are processed. Current software graph processing frameworks cannot exploit this insight at runtime because online traversal scheduling is simply too expensive. When only a few instructions are executed per edge, even trivial traversal scheduling adds prohibitive overheads. Instead, software frameworks process vertices in the order they are laid out in memory [48, 52], a strategy we call *vertex-ordered scheduling*. Vertex-ordered scheduling is sensible on systems with general-purpose cores, but it forgoes significant locality. To recover this locality, current frameworks can use offline preprocessing that changes the graph layout to improve the locality of subsequent vertex-ordered traversals [22, 55, 59, 62, 64]. Unfortunately, preprocessing is itself very expensive, and thus only makes sense on graphs that change infrequently, ruling out many important applications [33, 39].

The key idea of this paper is that *hardware acceleration enables more sophisticated, online traversal scheduling*, allowing systems to improve locality without expensive preprocessing. We propose HATS, which introduces a simple, specialized scheduling unit near each core that runs ahead and chooses which edges to traverse. Prior graph accelerators for FPGAs [15, 42, 43] and ASICs [2, 22, 40, 44] include specialized scheduling logic, but they all implement the simple, locality-oblivious vertex-ordered scheduling. HATS is the first design that exploits hardware acceleration to improve traversal scheduling itself.

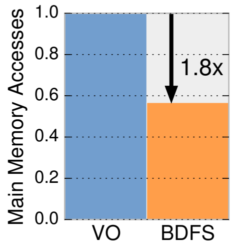


Fig. 1: BDFS reduces memory accesses by 1.8× for PageRank Delta on `uk-2002`.

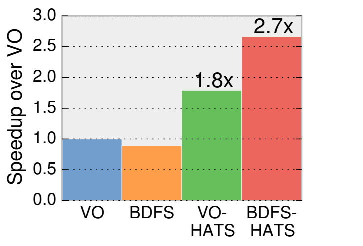


Fig. 2: VO-HATS and BDFS-HATS improve performance by 1.8× and 2.7× for PageRank Delta on `uk-2002`.

Specifically, we propose *bounded depth-first scheduling (BDFS)*. In BDFS, each core explores the graph in a depth-first fashion up to a given maximum depth. This restricts each core to explore a small, well-connected region of the graph at a time, improving temporal locality on graphs with good community structure. Prior work [1, 13] has observed that DFS is a good technique to exploit locality and has exploited it in offline graph preprocessing [5, 59, 61]. BDFS is the first to exploit DFS for *online* locality-aware scheduling. HATS implements BDFS with simple hardware similar to previous indirect prefetchers [58]. Unlike these prefetchers, which only hide latency, BDFS changes the traversal to improve locality and thus reduces both latency *and bandwidth*.

Fig. 1 illustrates the benefits of BDFS for the PageRank Delta algorithm [35] on the `uk-2002` web graph [16]. BDFS reduces memory accesses by 1.8× over the vertex-ordered schedule (VO). Prior prefetchers and graph accelerators do not reduce memory accesses, since they use the same vertex-ordered schedule as software frameworks.

Fig. 2 shows the execution time of PageRank Delta on `uk-2002`. *In software, BDFS does not improve performance because its overheads outweigh its locality benefits*. But hardware acceleration reverses this situation. HATS improves VO's performance by 1.8× (VO-HATS) due to accurate prefetching that hides memory latency. But prefetching saturates memory bandwidth, so *improving performance further requires reducing memory accesses*. BDFS achieves this and BDFS-HATS outperforms VO-HATS by 1.5× and VO by 2.7×.

We have prototyped BDFS-HATS in RTL and evaluated it using detailed microarchitectural simulation. We consider two system configurations: one where HATS engines are implemented in hardware and another where they use on-chip reconfigurable logic (similar to the Xilinx Zynq SoC, but with high-performance cores). BDFS-HATS is easy to implement and requires just 0.14 mm^2 and 72 mW at 65 nm (or 3.2 K FPGA LUTs). This translates to 0.4% area and 0.2% power overhead over a general-purpose core. We evaluate HATS on five important graph algorithms, processing real-world graphs whose working sets are much larger than the on-chip cache capacity. On a 16-core system, BDFS-HATS reduces main memory accesses by up to 2.4× and by 30% on average, and improves performance by up to 3.1× and by 83% on average. HATS thus gives a practical way to improve the locality of graph processing.

II. Background and Motivation

A. Current graph processing frameworks

Software graph processing frameworks [21, 41, 48, 52, 63] provide a simple interface that lets application programmers specify algorithm-specific logic to perform operations on graph vertices and edges. The runtime is then responsible for scheduling and performing these operations. The runtime tracks which vertices are active in each iteration and performs algorithm-specific operations on them until there are no more active vertices or a termination condition (e.g., number of iterations) is reached. We assume a Bulk Synchronous Parallel (BSP) [54] model, where updates to algorithm-specific data are made visible only at the end of each iteration.

Many graph algorithms are *unordered* and the runtime has complete freedom on how to schedule the processing of active edges in each iteration. Such scheduling does not affect the correctness of the algorithm, but has a large impact on locality. Before analyzing the locality tradeoffs of scheduling, we first describe these frameworks in more detail.

Graph format: Most graph processing frameworks use the compressed sparse row (CSR) format, or its variations, for its simplicity and space efficiency [21, 41, 48, 52, 63]. As Fig. 3 shows, CSR uses two arrays, `offset` and `neighbor`, to store the graph structure. For each vertex id, the `offset` array stores where its neighbors begin in the `neighbor` array. Hence, each vertex v has edges to each `neighbor`$[i]$ for $i =$ `offset`$[v] \rightarrow$ `offset`$[v+1]$. The `neighbor` array stores the vertex id of each neighbor, so it has as many entries as edges in the graph. For weighted graphs, the `neighbor` array also stores the weight of each edge. Algorithm-specific data is stored in a separate `vertex_data` array. For example, in PageRank, `vertex_data` stores the score of each vertex.

Graph algorithms can perform *pull-* or *push-based* traversals. In pull-based traversals, the CSR format encodes the *incoming* edges of each vertex, as Fig. 3 shows, and each processed vertex (the destination vertex) pulls updates from its in-neighbors (sources). In push-based traversals, the CSR format encodes the *outgoing* edges of each vertex, and each processed vertex (source) pushes updates to its out-neighbors (destinations).

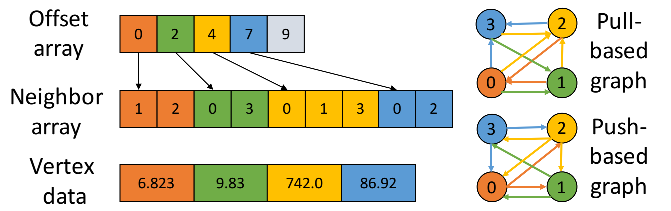


Fig. 3: Compressed sparse row (CSR) format. The offset array stores, for each vertex, the starting location of its neighbors' vertex ids in the neighbor array. The vertex data array stores algorithm-specific data.

Vertex-ordered scheduling: State-of-the-art graph processing frameworks [41, 48, 52, 63] follow a *vertex-ordered schedule* (VO), a simple technique that achieves spatial locality in accesses to edges but suffers from poor temporal and spatial locality on accesses to neighbor vertices.

The vertex-ordered schedule simply processes the active vertices in order of vertex id, and processes all the edges of each vertex consecutively, as specified by the graph layout. Processing an edge usually involves accessing the `vertex_data` of both the current and neighbor vertices. Listing 1 shows pseudocode for a single iteration of PageRank following the vertex-ordered schedule. We show the pull version, where destination vertices pull updates from their in-neighbors.

```
def PageRank(Graph G):
  for dst in range(G.numVertices):
    for src in G.neighbors(dst):
      G.vertex_data[dst].newScore +=
              G.vertex_data[src].oldScore /
              G.vertex_data[src].degree
```

Listing 1: PageRank using the vertex-ordered schedule.

When the in-memory graph layout (i.e., `offset` and `neighbor` arrays) does not correlate with the graph's community structure, the vertex-ordered schedule suffers from poor temporal locality. Consider the example graph shown in Fig. 4. The graph has two well-connected regions that are weakly connected to each other. To maximize temporal locality, all the vertices and edges in one region should be processed before moving to the next. But if the vertices of the two regions are interleaved in the graph layout as in Fig. 4, the vertex-ordered schedule alternates between regions, yielding poor temporal locality.

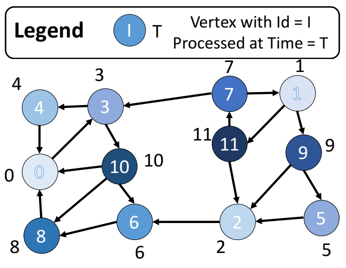


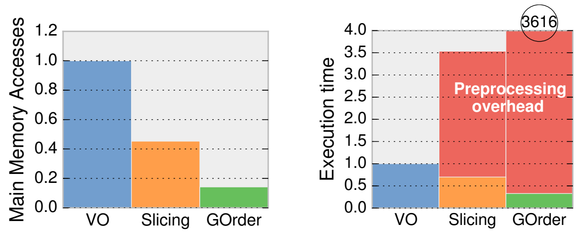
Fig. 4: The vertex-ordered schedule ignores graph structure and alternates between the two communities.

Preprocessing improves locality but is expensive: Prior work has proposed several *graph preprocessing techniques* to improve locality [22, 43, 52, 55, 59, 61, 62, 64]. These techniques change the order in which vertices are stored so that closely connected communities are stored near each other in memory. This improves the temporal and spatial locality of the vertex-ordered schedule: it improves temporal locality by placing a vertex's neighbors close together, so accesses from those neighbors to the vertex happen nearby in time; and it improves spatial locality by placing related vertices in the same cache line.

For example, for the graph in Fig. 4, preprocessing would analyze the graph structure, identify the two regions and modify the layout to place the vertices in the first region before those in the second. When the vertex-ordered schedule is used with the modified layout, it closely follows the community structure, fully processing the first region before moving to the second.

Although preprocessing improves locality, it is very expensive. Rewriting the graph requires several passes over the full edge list. As a result, preprocessing often takes longer than the graph algorithm itself, making it impractical for many important use cases [33, 39]. Preprocessing costs can be amortized if the same graph is reused many times, but in many applications the graph changes over time or is produced by another algorithm, and is used once or at most a few times [34].

Fig. 5 illustrates this for one iteration of the PageRank algorithm on the `uk-2002` graph [16]. We compare *(1)* The vertex-ordered (VO) schedule *(2)* Slicing [22], a relatively cheap preprocessing technique that ignores graph structure, and *(3)* GOrder [55], an expensive preprocessing technique that heavily exploits graph structure. Because GOrder takes too long to simulate, we measure its preprocessing overhead on an Intel Xeon E5-2658 v3 (Haswell) processor running at 2.2 GHz with a 30 MB LLC.



(a) Main memory accesses. (b) Execution time.

Fig. 5: Memory accesses and execution time for one PageRank iteration on `uk-2002` with various preprocessing schemes.

Although both Slicing and GOrder reduce memory accesses significantly and improve PageRank's runtime over the vertex-ordered schedule, they incur significant preprocessing time. When including preprocessing time, these techniques are beneficial only when the algorithm converges in more than 10 and 5440 iterations, respectively.

Several preprocessing techniques exploit the locality benefits of depth-first search (DFS) and breadth-first search (BFS) traversals [5, 14, 59, 61]. Children-DFS [5, 55] partitions the graph by using a variant of DFS that seeks to group the neighbors of each vertex. PathGraph [59] partitions the graph by performing local breadth-first traversals while limiting the partition sizes and relabels the vertices in each partition in DFS order. FBSGraph [61] is a distributed framework that uses path-centric partitioning and scheduling to improve convergence rate of asynchronous graph algorithms. It leverages the necessary graph partitioning pass and only improves temporal locality of `vertex_data` since it does not change the graph layout.

Unlike these techniques, BDFS improves temporal locality without preprocessing. However, because BDFS does not change the graph's layout, it does not improve spatial locality.

Online heuristics to improve locality: Motivated by the high costs of preprocessing, prior work has explored alternative, cheaper runtime techniques to improve locality. Milk [26] and Propagation Blocking [8] translate irregular indirect memory references into batches of efficient sequential DRAM accesses. These techniques are a spatial locality optimization, and only benefit algorithms with small vertex objects. By contrast, BDFS exploits the graph's community structure to improve *temporal* locality and benefits algorithms with large or small vertex objects. While these techniques are effective with unstructured (i.e., random) graphs, they forgo significant temporal locality for graphs with good community structure.

B. Prior hardware techniques to accelerate graph processing

Indirect prefetching: Conventional stream or strided prefetchers do not capture the indirect memory access patterns of graph algorithms. IMP [58] is a hardware prefetcher that dynamically identifies and prefetches indirect memory access patterns, without requiring any application-specific information. Similarly, Ainsworth and Jones [3] propose a specialized prefetcher that uses information about an application's data structures to prefetch indirect memory accesses.

These prefetchers all assume a vertex-ordered schedule and improve performance by hiding memory access latency. They easily saturate memory bandwidth and become bandwidth-bound. By contrast, BDFS *changes the traversal schedule* to reduce bandwidth demand, allowing it to outperform perfect prefetching of a vertex-ordered schedule. HATS fetches graph data ahead of the core, but this is more similar to how graph accelerators non-speculatively fetch data than to indirect prefetchers like IMP, which predict the access pattern outside the core to issue speculative prefetches.

Graph accelerators: Recent work has proposed specialized graph-processing accelerators for both FPGAs [15, 42, 43] and ASICs [2, 22, 40, 44, 50, 60]. While these accelerators introduce specialized scheduling logic, they implement the same vertex-ordered schedule used by software algorithms and likewise rely on expensive preprocessing to improve locality [22, 43, 50, 60]. The premise of our paper is that this misses an opportunity: specialization enables online locality-aware schedules that achieve most of the benefits of preprocessing without its overheads.

Beyond scheduling, these accelerators use both compute and memory system specialization to achieve large performance and energy efficiency gains. Our paper complements this prior work by using locality-aware scheduling to make better use of limited on-chip cache capacity. Although we describe our techniques in the context of a general-purpose system, we expect it to be more beneficial on accelerators that are even more bottlenecked on memory accesses.

Decoupled access-execute: HATS takes inspiration from decoupled access-execute (DAE) architectures [49], where an access core performs all memory operations and an execute core performs all compute operations. Access and execute cores communicate through queues, allowing the access core to run ahead.

In some aspects, HATS is similar to an access core: it is decoupled from the main core through a queue, and runs ahead of it, exposing abundant memory-level parallelism. However, HATS is specialized to graph traversals, making it much cheaper and faster than a programmable access core. And unlike DAE, the main core still performs memory accesses instead of communicating them to HATS. This lets HATS focus on handling the traversal of the graph. Also, unlike in conventional DAE, communication between HATS and the main core is *one-sided*, letting HATS run far ahead of the core and avoiding the performance bottlenecks of DAE, where two-way communication often caused loss of decoupling [53].

III. Improving Locality with BDFS

Our goal is to achieve most of the benefits of preprocessing while avoiding its overheads. We improve temporal locality by scheduling edges *at runtime* to match the graph's community structure, without modifying the graph layout. This section describes our basic technique, BDFS, and the next describes our hardware implementation of this technique, HATS.

BDFS traverses the graph by performing a series of bounded depth-first searches, each of which visits a region of connected vertices. Bounded depth-first search is used in several contexts, such as iterative deepening [27] and search and optimization techniques [9, 51]. Moreover, as described in Sec. II-A, several preprocessing algorithms leverage DFS to improve locality [5, 59, 61]. However, to the best of our knowledge, we are the first to use BDFS for *online* locality-aware scheduling of graph traversals.

We first describe a sequential implementation of BDFS, analyze its locality, and then discuss its parallel implementation.

A. BDFS algorithm

Listing 2 shows the pseudocode for PageRank using a recursive implementation of BDFS. BDFS uses an `active` bitvector to track the vertices that are not yet processed. Listing 2 shows a version of PageRank where all the vertices are active in each iteration [45], so the bitvector is initialized to all ones.[1] BDFS starts processing at the first vertex (id 0). Thereafter, it chooses the next vertex to process from the neighbors of the current vertex, ignoring inactive vertices. This exploration proceeds in a depth-first fashion, always staying within `maxDepth` levels away from the root vertex. Once the exploration from a root vertex is finished, BDFS scans the `active` bitvector to find the next unvisited vertex. This repeats until all vertices are visited.

[1]There are more efficient versions that do not process all vertices on each iteration. We later evaluate PageRank Delta, which performs this optimization.

```
def PageRank(Graph G):
  iterator = BDFS(G)
  while iterator.hasNext():
    (src, dst) = iterator.next()
    G.vertex_data[dst].newScore +=
            G.vertex_data[src].oldScore /
            G.vertex_data[src].degree

def BDFS::next():
  active = BitVector(G.numVertices)
  active.setAll()
  for root in range(G.numVertices):
    if active[root]:
      active[root] = False
      BDFS::explore(root, 0)

def BDFS::explore(int dst, int curDepth):
  for src in G.neighbors(dst):
    yield (src, dst)
    if curDepth < maxDepth:
      if active[src]:
        active[src] = False
        BDFS::explore(src, curDepth+1)
```

Listing 2: PageRank implementation using BDFS. `yield()` **returns a value to the caller, but resumes at that point when the callee is next invoked.**

Fig. 6 shows the order in which BDFS processes vertices in the example graph using `maxDepth` of 10. (This is a pull-based traversal so BDFS traverses *incoming* edges, e.g., from vertex 0 to 4.) Unlike the vertex-ordered schedule, BDFS tends to process close-knit regions together. BDFS improves temporal locality of accesses to `vertex_data` for two reasons. First, neighbors of a given vertex are more likely to share neighbors, so processing them together naturally exploits the community structure of real-world graphs. Second, since processing an edge involves accessing the `vertex_data` of both the currently processed vertex and its neighbor, processing one of the neighbor vertices next results in at least one access to already cached data.

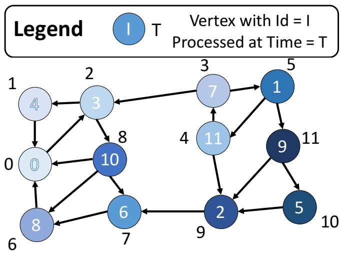


Fig. 6: BDFS improves temporal locality by processing neighbors together and vertices within a community close in time.

Scheduling overheads: The main scheduling structures in BDFS are a LIFO stack and the `active` bitvector. BDFS requires only a small stack, which causes near-zero main memory accesses. Although the bitvector gets irregular accesses, it is much smaller than `vertex_data`. For example, in PageRank the bitvector is 128× smaller than `vertex_data`, which stores 16 B per vertex.

The real overheads of BDFS are not extra memory accesses, but the scheduling logic in Listing 2 to find the next vertex. Although BDFS has linear-time complexity ($O(\#Edges + \#Vertices)$), since most graph algorithms execute few instructions per edge, it is relatively expensive in software: BDFS not only executes 2–3× more instructions than VO, but these extra instructions have data-dependent branches that limit instruction-level parallelism. These overheads outweigh the locality improvements of BDFS, motivating HATS.

B. Analysis of access patterns

Fig. 7 compares the memory access patterns of the vertex-ordered (VO) schedule (Listing 1) and BDFS (Listing 2). We show accesses to the `neighbor` and `vertex_data` arrays.

VO processes vertices and their edges sequentially, which results in good spatial locality on the `offset` and `neighbor`

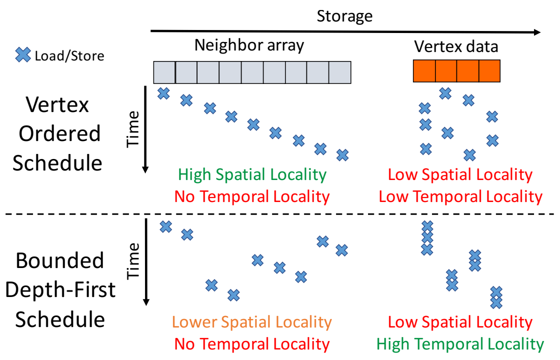


Fig. 7: Memory access patterns with the vertex-ordered schedule (top) and BDFS (bottom).

arrays, and in `vertex_data` accesses *for the currently processed vertex*. While the `neighbor` array is often the largest data structure in the graph, each cache line has many elements (typically 16), which amortizes their fetches well.

However, VO suffers from poor temporal and spatial locality on `vertex_data` accesses *for neighbor vertices*. These accesses dominate misses. Fig. 8 illustrates this by showing the breakdown of main memory accesses to different data structures in PageRank: 86% of main memory accesses are to neighbor `vertex_data`.

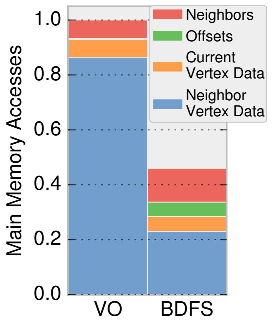


Fig. 8: Breakdown of memory accesses to different data structures for PageRank on the `uk-2002` graph.

Fig. 7 shows that BDFS improves temporal locality in neighbor `vertex_data` accesses by processing communities together. However, it reduces spatial locality in `offset` and `neighbor` array accesses. In BDFS, the first access to a vertex's slice of the `neighbor` array often misses. Fortunately, accesses to the remaining neighbors enjoy the same spatial locality as VO. Fig. 8 shows this is a good tradeoff: neighbor `vertex_data` misses are almost 5× lower, and while `offset` and `neighbor` misses increase, BDFS reduces memory accesses by 2.2×.

C. BDFS does not require tuning the maximum depth

Alternative search strategies: An alternative to BDFS is bounded breadth-first scheduling (BBFS). BDFS outperforms BBFS and is also cheaper to implement. DFS has better locality than BFS [1, 5, 13], and DFS works well with a small stack while BFS requires a large FIFO queue (up to the entire graph).

Fig. 9 illustrates this for PageRank. It shows the memory accesses of BDFS and BBFS as the *fringe* (BDFS stack or BBFS queue) grows. Memory accesses are normalized to the vertex-ordered schedule. BDFS outperforms BBFS at all fringe sizes and achieves near-peak performance with a 10-element fringe, whereas BBFS needs about a 100-element fringe to be effective. All the graphs we evaluate show similar trends.

BDFS is insensitive to stack depth: Fig. 9 shows that BDFS's performance is flat after a stack depth of 5–10. Smaller fringes cause more misses because they traverse smaller communities

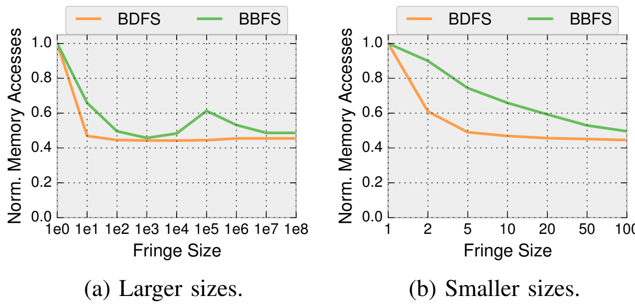


Fig. 9: Memory accesses of PageRank on the `uk-2002` graph with BDFS and bounded BFS (BBFS) at different fringe sizes. BDFS reduces memory accesses with much less storage than BBFS.

that use only a fraction of the cache. For example, with an average degree of 4 neighbors/vertex, a depth of 4 traverses only about $4^4 = 256$ vertices, whereas a depth of 10 traverses about $4^{10} = 1\,\mathrm{M}$ vertices.

However, the converse is not true: *deeper stacks do not add misses* even if they result in huge traversals with many more vertices than the cache can hold. This stems from DFS's divide-and-conquer nature [18]. Suppose that, for a given graph, a stack of depth D yields the largest communities that fit on cache. Suppose we instead use depth $D+1$. If the root node has N neighbors, this is equivalent to performing N depth-D traversals in sequence, each of which fits in cache. By induction, BDFS does not overwhelm the cache with a deeper stack.

This observation yields two nice properties. First, *BDFS needs no tuning*. Rather than analyzing the graph and figuring out the right stack depth, we simply use a fixed depth in hardware that is large enough to yield large traversals even with small degrees. Second, BDFS should yield good locality at different cache levels, regardless of their size.

D. Parallel BDFS

We parallelize BDFS by evenly dividing the `active` bitvector across threads. Each thread then begins independent BDFS traversals through its chunk of the vertices, as in Listing 2. The only change is that operations on the `active` bitvector are done atomically (e.g., test-and-clear) to avoid repeating work. Finally, we use work-stealing [11] to avoid load imbalance: when a thread finishes its chunk it tries to steal half of another thread's remaining vertices.

We tried a number of more sophisticated parallelization strategies that seek to keep all threads exploring within the same community of the graph. We found that, on most graphs, these added synchronization overheads without providing much benefit over our simple work-stealing approach.

IV. HATS: Hardware-Accelerated Traversal Scheduling

BDFS effectively reduces cache misses, but when implemented in software, its overheads negate the benefits of its higher locality. To address this problem, we present *hardware-accelerated traversal scheduling (HATS)*. HATS is a simple, specialized engine that performs traversal scheduling. HATS enables sophisticated scheduling strategies like BDFS.

System architecture: Fig. 10 shows the system architecture we use in this paper. Each core is augmented with a HATS engine, which it configures to perform the traversal (e.g., passing in the addresses of CSR structures). Each HATS engine runs ahead of its core and communicates edges to the core through a FIFO buffer. Our design effectively offloads the traversal scheduling portion of the graph algorithm to the HATS engines, and uses cores exclusively for edge and vertex processing. For example, in Listing 2, the core executes the per-edge operations inside the `PageRank()` function, while the HATS engine executes everything else.

We propose and evaluate HATS on general-purpose processors, where HATS is implemented as either fixed-function

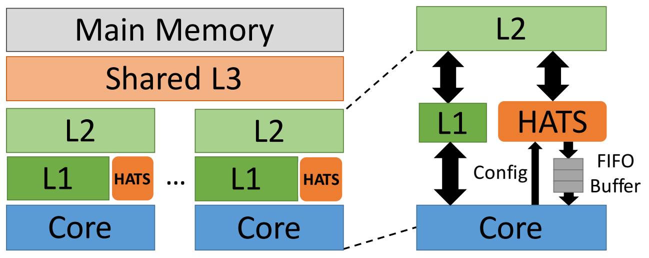


Fig. 10: System architecture. Each core has a HATS engine that traverses the graph and sends edges to process to the core.

hardware or using on-chip reconfigurable logic. We focus on general-purpose processors for two reasons. First, traversal scheduling is needed by all graph algorithms, so it is natural to specialize this common part and leave algorithm-specific edge and vertex processing to programmable cores. Second, specialized traversal schedulers impose negligible system-wide overheads, similar in cost to prior indirect prefetchers. And unlike prefetchers, HATS reduces both memory latency *and bandwidth* (Sec. II-B). HATS thus adds a small dose of specialization to get a large performance boost for an important application domain, without sacrificing the programmability and low entry cost of general-purpose processors.

That said, HATS can be applied to other system architectures, e.g., by replacing the general-purpose cores with an algorithm-specific accelerator.

Generality: HATS supports both push- and pull-based traversals, and all-active and non-all-active algorithms. This lets HATS accelerate the vast majority of graph processing algorithms—the full spectrum of what state-of-the-art frameworks like Ligra support.

HATS assumes a CSR graph format, which is by far the most commonly used one [21, 41, 48, 63]. With small additions, HATS could support other CSR variants (e.g., DCSR [12]). Moreover, the reconfigurable logic implementation of HATS would allow supporting other graph formats with no overheads.

In the remainder of this section, we explain the HATS interface and operation in detail, which is common to all HATS variants. We then describe HATS implementations of VO and BDFS traversals, and compare the area and power overheads of the ASIC and FPGA implementations of HATS.

A. HATS interface and operation

HATS only accelerates traversal scheduling and leaves all other responsibilities to cores, including initialization, edge and vertex processing, and load balancing.

Operation: Regardless of the traversal scheduling strategy implemented by HATS (VO or BDFS), each traversal (e.g., one iteration of PageRank) proceeds in the following steps:

1. Initialization: Software first initializes all the required data structures, including all graph data and, if needed, the `active` bitvector, which specifies the set of vertices to visit. The need for this bitvector depends on the graph algorithm and traversal schedule: VO-HATS (Sec. IV-B) uses an `active` bitvector only for algorithms where not all vertices are active each iteration (e.g., BFS), while BDFS-HATS (Sec. IV-C) always uses an `active` bitvector to avoid processing vertices multiple times.

2. HATS configuration: Each thread then configures its own HATS unit by conveying the following information:

1) The base addresses and sizes of graph data structures: `offset`, `neighbor`, and `vertex_data` arrays, and `active` bitvector.
2) The type of traversal (push or pull).
3) The chunk of vertices that the HATS is responsible for (start and end vertex ids).

This configuration data is written using memory-mapped registers (e.g., like a DMA engine is configured). After the core writes this configuration, HATS starts the traversal.

3. Processing: During traversals, HATS reads the `offset` and `neighbor` arrays, as well as the `active` bitvector, if used. For BDFS, HATS also performs updates to the `active` bitvector. Finally, it prefetches `vertex_data`.

As HATS finds unvisited active vertices, it fills its FIFO buffer with edges (source and destination vertex ids) for the core to process. The core uses a `fetch_edge` instruction to fetch edges from the buffer (this is the only new instruction). `fetch_edge` returns the source and destination ids in registers. Software takes two extra instructions to translate these ids to `vertex_data` addresses. If HATS has finished traversing its assigned chunk of vertices, `fetch_edge` returns $(-1, -1)$. If the FIFO is empty, `fetch_edge` stalls the core. If it fills up, HATS's traversal stalls.

HATS is transparent to applications: We expose the above functionality to the software graph processing framework through a simple low-level API consisting of two methods: `hats_configure(...)` performs the configuration step and `hats_fetch_edge()` translates to a `fetch_edge` instruction. Graph algorithms need not deal with this API: we code all our algorithms to a highly optimized, Ligra-like graph processing framework (Sec. V-A). Only the framework needs to be modified to use HATS—application code is unchanged.

Parallelism and load-balancing: Parallel operation is similar to the software BDFS implementation (Sec. III-D): vertices are divided into as many chunks as threads, and each HATS engine is responsible for scanning a separate chunk. We perform load-balancing using work-stealing: if a HATS engine finishes its chunk early, its thread interrupts a randomly chosen thread, which donates half of the remaining chunk in its HATS engine. We use the same termination algorithm as Cilk [19].

Handling preemption: Because HATS does work on behalf of the thread, some of its state is architecturally visible and must be considered on preemption events. If the OS deschedules a thread, it quiesces the HATS engine and saves this architectural state (which includes the remainder of the chunk and base addresses of all data structures). When the thread is rescheduled, its core's HATS is configured using this state. Note that this is needed only when the thread is descheduled, not when taking exceptions or system calls (similar to how FPU state is not saved when entering into the kernel). It is thus a rare event.

Virtual memory: Finally, HATS operates on virtual addresses. Like prior indirect prefetchers, HATS leverages the core's address-translation machinery [3, 58]. Unlike indirect prefetchers, HATS does not monitor the core's cache accesses. Since we place HATS at the core's L2, we use the L2 TLB.

HATS may cause a page fault, which is handled as in prior indirect prefetchers: the core is interrupted and the OS page fault handler invoked. The HATS engine stalls until the page fault handler completes. Once the page fault handler finishes, core and HATS engine resume normal execution.

B. VO-HATS implementation

We now describe the design of the HATS engine when using the vertex-ordered schedule. We describe its operation assuming a push-based traversal. Sec. IV-D discusses the changes needed for pull-based traversals.

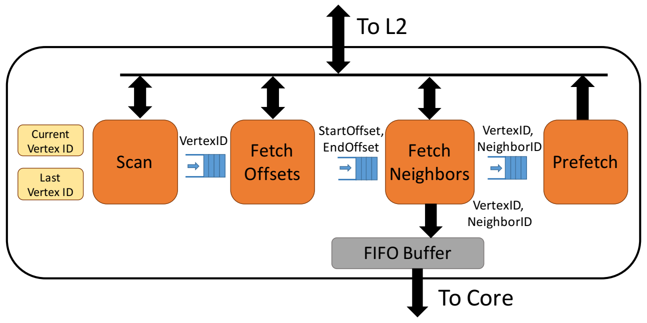


Fig. 11: Microarchitecture of VO-HATS.

Our VO-HATS design uses a simple pipelined implementation, illustrated in Fig. 11. Each pipeline stage corresponds to a particular step in the fetching of graph data:

1) *Scan* holds the current and last vertex ids of the HATS chunk. For an all-active algorithm, Scan simply outputs the current vertex id each cycle, and increments it. If the algorithm is not all-active, Scan loads the `active` bitvector line by line and outputs the ids of active vertices.
2) *Fetch offsets* takes a vertex id as input and outputs its start and end offsets, which it loads from the `offsets` array.
3) *Fetch neighbors* takes the start and end offsets of a single vertex as input and outputs its neighbor ids, which it loads from the `neighbors` array. The vertex and its neighbor ids are then queued in the FIFO buffer.
4) *Prefetch* issues prefetches for the vertex and its neighbors' `vertex_data`.

To allow enough memory-level parallelism, these stages are decoupled using small FIFOs. In practice, we find that allowing the *Scan* and *Fetch neighbors* stages to each request up to two cache lines in parallel suffices to keep the FIFO full.

C. BDFS-HATS implementation

Our implementation of BDFS-HATS shares many common elements with VO-HATS, but has additional logic to perform data-dependent traversals. Fig. 12 shows its design. We first explain its basic operation, and then the optimizations required to achieve good performance.

Basic operation: The main component of the BDFS scheduler is a fixed-depth stack. Each stack level stores the following information about a single vertex: its vertex id, current and end offsets, and a cache line worth of neighbor ids. These

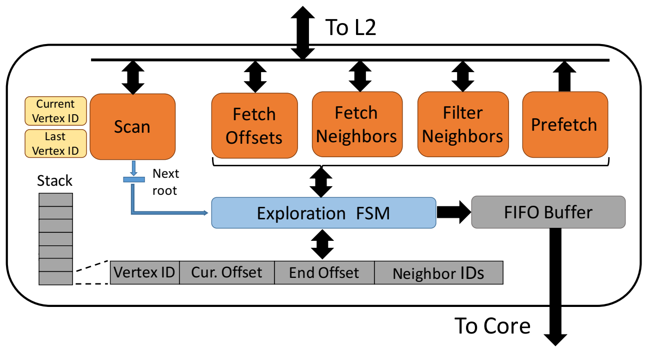


Fig. 12: Microarchitecture of BDFS-HATS.

fields are populated as the vertex is processed. The stack is provisioned for the maximum depth of BDFS exploration (10 levels in our implementation). As discussed in Sec. III-C, it is not necessary to tune BDFS's depth—we always use the maximum depth.

Unlike VO, BDFS always uses an `active` bitvector, even for all-active algorithms. HATS reads this bitvector to restrict its exploration to active vertices, and updates it as it traverses the graph, clearing the vertices it decides to explore.

The traversal begins with an empty stack. As in VO, the *Scan* stage traverses the `active` bitvector and produces the next active vertex id. This vertex is immediately marked as inactive. Then the vertex serves as the root of a bounded-depth first exploration: the vertex id is stored in the first level of the stack. Its offsets are fetched, and, once known, they are used to fetch the first cache line of neighbor vertex ids. These neighbors are checked in the `active` bitvector, and those that are active are marked inactive and stored, along with the vertex's offsets, in its stack entry.

The traversal continues in a depth-first fashion: the first neighbor of the topmost element is used to populate the next level of the stack as above, and so on until the stack is filled.

As the depth-first traversal proceeds, newly fetched neighbor ids are used to produce edges, which are queued to the FIFO buffer. Once the stack fills up, the neighbor ids of the last level are fetched and used to produce edges, but are not traversed. When all the neighbors at a given level have been traversed, the level is cleared and the exploration continues at the next neighbor of the previous level. When all of the root's neighbors have been explored, the current region has been fully explored, and the *Scan* stage provides the next root vertex.

BDFS-HATS uses a finite state machine (FSM) to implement this procedure, shown in Fig. 12. On each cycle, the FSM decides on the next piece of data to fetch based on the current state of the stack.

Exploiting intra-traversal parallelism: Unlike VO, BDFS traversals experience more data dependences and thus more serialization. Additional optimizations are needed to exploit the parallelism *within* a single BDFS traversal in order to obtain enough memory-level parallelism and saturate each core.

First, we move the `active` bitvector check-and-clear operations off the critical path and perform them in parallel. These checks constitute a substantial fraction of the work in BDFS.

TABLE I: Area and power of VO-HATS and BDFS-HATS implementations: ASIC (65nm) and FPGA (Zynq-7045).

HATS Design	ASIC Area (mm^2)	ASIC Area %core	ASIC Power (mW)	ASIC Power %TDP	FPGA Area $(LUTs)$	FPGA Area %FPGA
VO	0.07	0.19%	37	0.11%	1725	0.79%
BDFS	0.14	0.38%	72	0.22%	3203	1.47%

Instead of checking whether a vertex should be visited eagerly, we add all vertices to the neighbor list, issue pending bitvector checks if there is nothing else to do, and mark them as active or inactive as the responses arrive.

Second, all levels in the stack expand the first two active neighbors in parallel, instead of only expanding the first one. Each stack level has an additional entry, and when the topmost element is populated, its first and second active neighbors are used to populate the next level. This way, when the level's current vertex is completely explored, the data for the next vertex is already available (and as soon as we switch to it, the data for the following vertex starts being fetched). This greatly reduces the critical path at the cost of some additional storage.

With these optimizations, we find that BDFS-HATS achieves enough throughput to avoid stalling the core.

D. Extending HATS for pull-based traversals

We have so far described push-based traversal variants of HATS. The above designs can be easily extended to perform pull-based traversals. The key difference is when the `active` bitvector checks happen. In a push-based traversal, the `active` bitvector is checked to filter vertices before exploring their neighbors. For example, VO-HATS does this in the *Scan* stage. By contrast, a pull-based traversal fetches the neighbors of all the vertices in the graph, then uses the bitvector to filter inactive neighbors. Thus, adapting our VO-HATS design simply requires performing the bitvector checks after the *Fetch neighbors* stage instead of in the *Scan* stage. The BDFS design requires similar changes. Our HATS prototypes support both push- and pull-based traversals.

E. Hardware costs

ASIC implementation: We have written Verilog RTL for both VO-HATS and BDFS-HATS engines and synthesized them using a commercial 65 nm process. Both designs meet a 1.1 GHz target frequency. Table I shows their area and power consumption under typical operating conditions. Area and power are negligible when compared to those of a core in the Intel Core 2 E6750, also manufactured in 65 nm [17]. Table I shows that BDFS-HATS takes about 0.4% of core area and 0.2% of core TDP. VO-HATS is even cheaper.

Prior indirect prefetchers do not allow for a direct area and power comparison, but we can use their internal storage requirements as a proxy. VO-HATS requires 2.5 Kbits of storage for its internal FIFO buffers and BDFS-HATS requires 6.4 Kbits for 10 stack levels. In addition, both designs use a 1 Kbit output FIFO buffer. In comparison, IMP [58] requires 5.5 Kbits of storage, so our HATS designs have about the same cost.

FPGA implementation: We also synthesized the HATS designs on an FPGA platform. VO-HATS and BDFS-HATS

TABLE II: Configuration of the simulated system.

Cores	16 cores, x86-64 ISA, 2.2 GHz, Haswell-like OOO [47]
L1 caches	32 KB, per-core, 8-way set-associative, split D/I, 3-cycle latency
L2 cache	128 KB, private per-core, 8-way set-associative, 6-cycle latency
L3 cache	32 MB, shared, 16-way hashed set-associative, inclusive, 24-cycle bank latency, LRU replacement
Global NoC	4×4 mesh, 128-bit flits and links, X-Y routing, 1-cycle pipelined routers, 1-cycle links
Coherence	MESI, 64 B lines, in-cache directory, no silent drops
Memory	4 controllers, FR-FCFS, DDR4 1600 (12.8 GB/s per controller)

require 1725 and 3203 LUTs respectively, as shown in Table I. This is less than 2% of the total LUT count on a modest Xilinx Zynq-7045 SoC [56] (state-of-the-art FPGAs have 10× more LUTs [57]). Both designs meet a 220MHz target frequency, 5× slower than the ASIC implementation.

To ensure that the HATS engine can match the core's throughput at this frequency, we need more parallelism within the engine. However, we do not need to replicate the entire HATS pipeline to achieve this. We find that `active` bitvector checks in the *Filter neighbors* stage become the bottleneck when operating at a lower frequency. Thus, we only replicate the bitvector check logic and perform multiple bitvector operations in parallel (4 in our case). Sec. V-C shows that with this support, even a 220MHz design can keep the core busy.

V. Evaluation

A. Methodology

We now present our evaluation methodology, including the simulated system, graph algorithms, and datasets we use.

Simulation infrastructure: We perform microarchitectural, execution-driven simulation using zsim [47]. We have implemented detailed cycle-driven models of our proposed VO and BDFS HATS designs.

We simulate a 16-core system with parameters given in Table II. The system uses out-of-order cores modeled after and validated against Intel Haswell cores. Each core has private L1 and L2 caches, and all cores share a banked 32 MB last-level cache. The system has four memory controllers, like Haswell-EP systems [23]. We use McPAT [30] to derive the energy numbers of chip components at 22 nm, and Micron DDR3L datasheets [37] to compute main memory energy.

Algorithms: We use five graph algorithms from the widely used Ligra [48] framework, as shown in Table III. These include both all-active and non-all-active algorithms. All algorithms use objects that are much smaller than a cache line (64 B).

PageRank computes the relative importance of vertices in a graph, and was originally used to rank webpages [45]. *PageRank Delta* is a variant of PageRank in which vertices are active in an iteration only if they have accumulated enough change in their PageRank score [35]. *Connected Components* divides a graph's vertices into disjoint subsets (or components) such that there is no path between vertices belonging to different subsets [13]. *Radii Estimation* estimates the radius of each vertex by performing multiple parallel BFS's from a small sample of vertices [32]. *Maximal Independent Set* finds a maximal subset of vertices such that no vertices in the subset are neighbors [10].

TABLE III: Graph algorithms.

Algorithm	Vertex Size	All-Active?
PageRank (**PR**)	16 B	Yes
PageRank Delta (**PRD**)	16 B	No
Conn. Components (**CC**)	8 B	No
Radii Estimation (**RE**)	24 B	No
Max. Indep. Set (**MIS**)	8 B	No

We obtain the source code for these algorithms from Ligra [48]. We adapt the scheduling code to use the HATS programming model, without modifying the per-algorithm code. We also incorporate several optimizations in the scheduling code like careful loop unrolling that yield significant speedups: our implementations outperform Ligra by up to 2.5×.

Our approach lets us start with an optimized software baseline, which is important since it affects the qualitative tradeoffs. In particular, we find that well-optimized implementations are more memory-bound and saturate bandwidth more effectively.

Datasets: We use several large real-world web and social graphs detailed in Table IV. These graphs are diverse (harmonic diameter: 5–38; average degree: 9–38; clustering coefficient: 0.06–0.55).

TABLE IV: Graph datasets.

Graph	Vertices (M)	Edges (M)	Description
uk	19	298	uk-2002 [16]
arb	22	640	arabic-2005 [16]
twi	41	1468	Twitter followers [28]
sk	51	1949	sk-2005 [16]
web	118	1020	webbase-2001 [16]

With the objects sizes listed in Table III, the `vertex_data` footprint is much larger than the last-level cache. We represent graphs in memory in compressed sparse row (CSR) format.

Graph algorithms are generally executed for several iterations until a convergence condition is reached. To avoid long simulation times, we use *iteration sampling*: we perform detailed simulation only for every 4th iteration and fast-forward through the other iterations (after skipping initialization). This yields accurate results since the execution characteristics of all algorithms change slowly over consecutive iterations. Even with iteration sampling, we perform detailed simulation for *over 100 billion instructions* for the largest graph.

B. ASIC HATS Evaluation

Main memory accesses: Fig. 13 shows the main memory accesses of VO and BDFS for *single-threaded* PageRank. Each bar shows the breakdown of accesses to different data structures as in Fig. 6. This includes misses due to demand accesses and prefetches. BDFS's benefits stem from reducing neighbor `vertex_data` misses, as explained in Sec. III-B. Fig. 13 shows that these benefits hold across most graphs: BDFS reduces main memory accesses significantly, by up to 2.6× and by 60% on average. `Other` indicates accesses to BDFS's data structures, mainly the `active` bitvector. These are negligible except when the bitvector does not fit in cache (for `web`).

BDFS reduces misses on all graphs except `twi`, due to `twi`'s weak community structure. On `twi`, BDFS does not improve temporal locality of `vertex_data` accesses, and adds `offset`

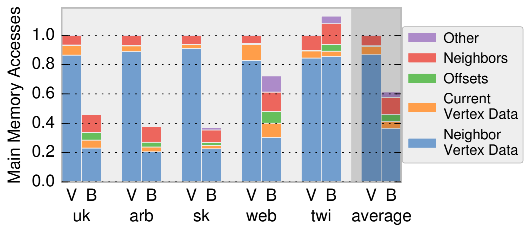


Fig. 13: Breakdown of main memory accesses by data structure for VO and BDFS on *single-threaded* PageRank.

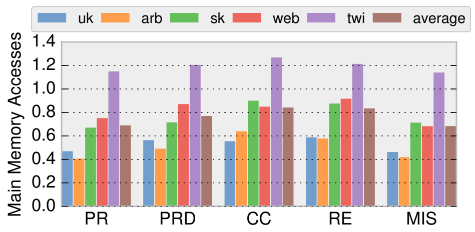


Fig. 14: Main memory accesses for BDFS over all algorithms at *16 threads*, normalized to VO's accesses.

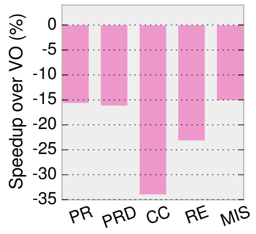
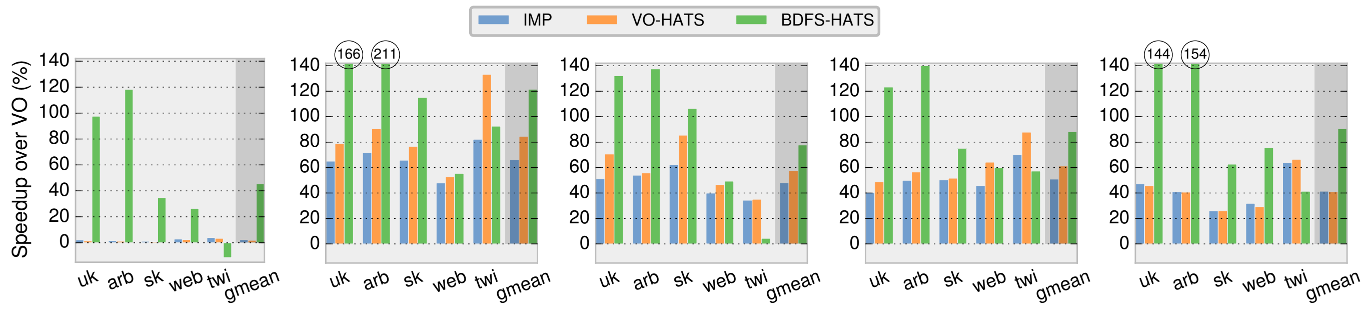


Fig. 15: Slowdown of *software* BDFS over VO at 16 threads.



(a) PR (b) PRD (c) CC (d) RE (e) MIS

Fig. 16: Speedup over software VO at 16 threads. VO-HATS and BDFS-HATS significantly improve performance over software VO and hardware prefetchers (IMP).

and `neighbor` misses. Preprocessing techniques [4] also show lower benefits for `twi`. Excluding `twi`, BDFS reduces memory accesses by 2× on average for PageRank.

`twi`'s weak community structure is an outlier. For example, `twi` has a clustering coefficient of 0.06, whereas most real-world graphs are above 0.2 [29]. Therefore, we expect BDFS to be beneficial in the common case. In Sec. V-D we present Adaptive-HATS, which can detect when graphs have weak community structure and switch to a VO schedule.

Fig. 14 shows the main memory accesses of BDFS at 16 threads for all five algorithms. BDFS reduces memory accesses significantly for all algorithms: by 44%, 29%, 18%, 19%, and 46% on average for `PR`, `PRD`, `CC`, `RE`, and `MIS` respectively. Some non-all-active algorithms like `PRD`, `CC`, and `RE` get slightly lower reductions. In these algorithms, only a subset of vertices are active in some iterations and as a result the active `vertex_data` is much more likely to fit in cache.

There is a slight increase in BDFS's memory accesses from 1 to 16 threads (compare Fig. 13 and PageRank in Fig. 14). In the single-thread experiments the whole 32 MB LLC is available to a single traversal, whereas in the 16-thread experiments the LLC is shared among 16 concurrent traversals, causing interference. The increase is small except for the `sk` graph, which is quite sensitive to per-thread cache capacity.

Performance: Fig. 15 shows the average slowdown of software BDFS over VO. In software, BDFS is slower than VO for all algorithms. This happens because, despite its large reductions in memory accesses, BDFS adds bookkeeping overheads when implemented in software. Since graph algorithms execute only a few instructions per edge, these overheads are relatively large.

Fig. 16 shows the speedup of three schemes over software VO: the IMP indirect memory prefetcher [58], and hardware-accelerated VO (VO-HATS) and BDFS (BDFS-HATS). To ensure IMP issues accurate prefetches, we configure it with explicit information about the graph structures as in [3].

IMP improves performance for the four non-all-active algorithms that are memory-latency bound (`PRD`, `CC`, `RE`, and `MIS`). When IMP does not saturate bandwidth (`PRD`, `CC`, and `RE`), VO-HATS achieves further gains by offloading traversal scheduling work to HATS. When IMP already saturates bandwidth (`PR`, `MIS`), VO-HATS does not improve performance further. Overall, VO-HATS improves performance over VO by up to 2.3× and by 85%, 58%, 61%, and 41% on average for `PRD`, `CC`, `RE`, and `MIS` respectively.

However, neither IMP nor VO-HATS reduce memory traffic, so their performance gains are limited by memory bandwidth. This is most noticeable for `PR` (Fig. 16a): software VO already saturates memory bandwidth, so VO-HATS and IMP barely improve performance. By contrast, BDFS's reduced memory accesses translate to improved performance for BDFS-HATS on `PR`, with up to 2.2× speedup over VO on the `arb` graph. On average, BDFS-HATS improves the performance of `PR` by 46% over VO and by 43% over VO-HATS.

BDFS-HATS achieves similar but slightly lower gains over VO-HATS for the non-all-active algorithms. BDFS-HATS improves average performance over VO-HATS by 20%, 13%, 17%, and 35% for `PRD`, `CC`, `RE`, and `MIS` respectively and over VO by 2.2×, 78%, 88%, and 91%.

Energy: Fig. 17 shows the energy breakdown for various schemes. For software-only VO, most of the energy comes from core and main memory, and the fraction of energy from cores depends on how compute-bound the algorithm is. For highly memory-bound applications like PageRank, main memory contributes 46% of total energy.

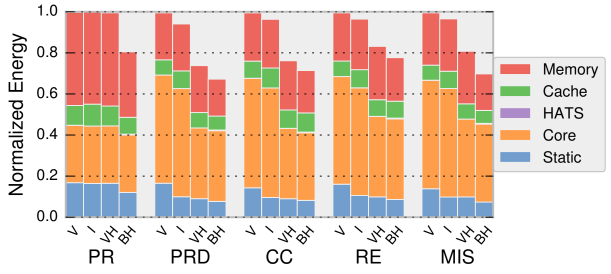


Fig. 17: Energy breakdown normalized to VO. (V = VO, I = IMP, VH = VO-HATS, BH = BDFS-HATS.)

HATS reduces core energy because it offloads the traversal scheduling to specialized hardware, reducing instruction counts on general-purpose cores. In particular, non-all-active algorithms spend a significant fraction of instructions in activeness checking even with simple VO. HATS completely eliminates these instructions and, on average, reduces core energy by 35%, 36%, 25%, and 28% for `PRD`, `CC`, `RE`, and `MIS` respectively.

BDFS's reduction in main memory accesses causes proportional reductions in main memory energy. Overall, BDFS-HATS reduces energy by 19%, 33%, 28%, 22%, and 30% on average over VO for the five algorithms. The overall energy reductions would be higher on graph processing accelerators, which reduce core energy by over 10× [22, 44], making memory energy the main bottleneck. IMP barely reduces energy since it neither reduces instruction count nor memory accesses.

C. *HATS on an on-chip reconfigurable fabric*

Results so far assume an ASIC implementation of HATS. Fig. 18 shows results for our VO-HATS and BDFS-HATS reconfigurable logic implementations. We model an on-chip FPGA fabric that can issue accesses to the L2 cache, similar to the Xilinx Zynq SoC [56] but using high-performance cores. Unlike Zynq, where the FPGA fabric is shared by all cores, we assume a per-core FPGA fabric near the private L2s. We later explore the effect of placing HATS at different points in the memory hierarchy, which accounts for less-integrated FPGA fabrics like HARP [24, 46].

The key difference with the ASIC implementation is the slower clock frequency, 220 MHz. Fig. 18 shows that when HATS has enough parallelism through replication of some parts of the pipeline, as explained in Sec. IV-E, even this slow clock is sufficient to keep the core busy. There is only a small drop in performance (1%) for both HATS versions. Without these changes (i.e., using the ASIC design on the FPGA), VO-HATS and BDFS-HATS are 15% and 34% slower on average.

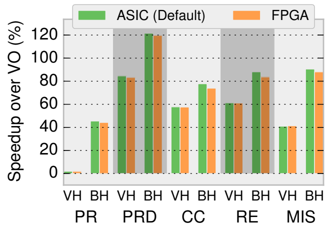


Fig. 18: HATS performance on an FPGA fabric.

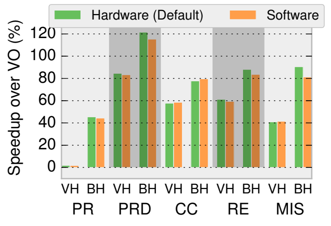
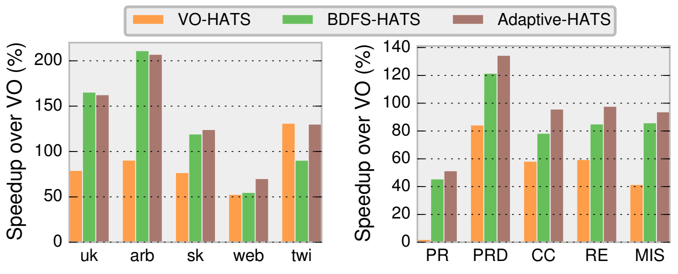


Fig. 19: FIFO buffer type.

We also modeled a variant where HATS and the core communicate through a FIFO buffer in shared memory. This avoids the need for a dedicated FIFO channel between the HATS engine and the core, which some reconfigurable fabrics may not offer. It also avoids changing the ISA (no `fetch_edge` instruction). Although buffer management operations increase core instructions by up to 10% (on `PR`), since the workloads are memory-bandwidth bound, there is negligible impact on performance, as Fig. 19 shows: VO-HATS is insensitive and BDFS-HATS shows at most 5% performance loss (on `MIS`).

D. *Adaptive-HATS*

We now explore an adaptive version of HATS that switches between VO and BDFS dynamically. Adaptive-HATS is beneficial for two reasons. First, when the graph does not have good community structure, BDFS increases memory accesses over VO and lowers performance. This can be observed for the `twi` graph across all algorithms in Fig. 16. Second, even for graphs with good community structure, the later phases of BDFS exploration usually process low-locality work. Using the simpler VO schedule in such phases improves performance due to lower scheduling overheads.



(a) PRD for each graph. (b) Gmean across graphs.

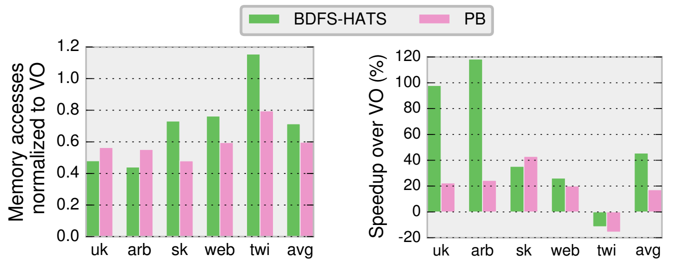
Fig. 20: Adaptive-HATS outperforms BDFS-HATS by avoiding BDFS-HATS's performance pathologies.

Adaptive-HATS requires small extensions to BDFS-HATS: switching between VO and BDFS only requires changing the maximum depth of exploration (1 for VO and 10 for BDFS). Every 50 M cycles, all HATS units switch to the alternative mode of exploration for 5 M cycles, and use the best-performing mode for the next 45 M cycles. Fig. 20a compares the performance of VO-HATS, BDFS-HATS and Adaptive-HATS on `PRD` for each graph. `web` and `twi` benefit the most from Adaptive-HATS on `PRD`; we observe similar benefits for other algorithms. Fig. 20b shows gmean performance across graphs. Adaptive-HATS outperforms BDFS-HATS by 4%, 6%, 10%, 7%, and 4% for `PR`, `PRD`, `CC`, `RE`, and `MIS`.

E. *BDFS-HATS versus other locality optimizations*

We now compare BDFS-HATS with online and offline techniques to improve locality.

Propagation Blocking [8] (PB) is an online heuristic to improve the spatial locality of all-active algorithms like PageRank. PB first accesses the graph sequentially to gather the updates to neighbors. It partitions these updates into bins, with each bin holding updates for a cache-fitting slice of vertices.



(a) Memory accesses. (b) Performance.

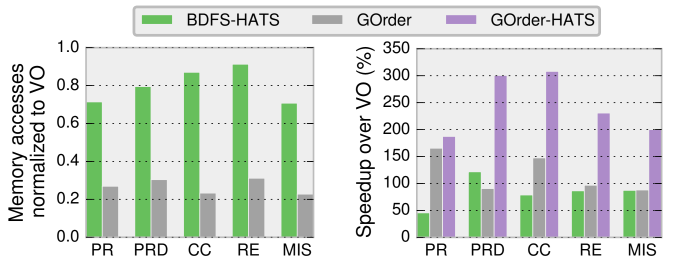
Fig. 21: Propagation Blocking reduces memory traffic significantly but shows limited performance gains.

Updates are stored in main memory. PB then reads the updates from memory bin by bin, and finally applies them. Both phases generate sequential accesses to main memory.

We used all optimizations from the original implementation, which we obtained from PB's authors [8]. We modified our simulator to model non-temporal stores, which are crucial to reduce PB's memory traffic. Moreover, we compare to Deterministic PB, which generates the per-update neighbor vertex ids only once and reuses them across iterations. We found that a bin size of 1 MB works best for our system.

Fig. 21a compares the memory accesses of BDFS-HATS and PB, normalized to VO. On average, PB achieves slightly lower memory accesses than BDFS-HATS, and works well even for unstructured graphs like `twi`. However, PB is a software technique that adds non-trivial compute to achieve these memory access reductions. Hence, as shown in Fig. 21b, the performance gains of PB are limited: while PB achieves up to 43% speedup for `sk`, it hurts performance for `twi`. On average, PB is 17% faster then VO, whereas BDFS-HATS is 46% faster. Moreover, PB has several limitations. PB can be extended for non-all-active algorithms [26], but the per-update neighbor vertex ids cannot be reused across iterations. And PB works only on algorithms where updates are commutative.

Fig. 22a compares the memory accesses of BDFS-HATS with GOrder [55] preprocessing, a very expensive heuristic (see Fig. 5b) that heavily exploits graph structure. GOrder achieves much lower memory accesses than BDFS-HATS and these memory traffic reductions translate to improved performance as shown in Fig. 22b. GOrder-HATS, which combines GOrder with VO-HATS, further improves performance significantly for non-all-active algorithms (`PRD`, `CC`, `RE`, `MIS`).



(a) Memory accesses. (b) Performance.

Fig. 22: BDFS-HATS versus GOrder preprocessing.

F. Sensitivity studies

Impact of prefetching: HATS engines accurately prefetch `vertex_data` into the L2 to accelerate edge processing by the cores. Fig. 23 shows that this prefetching is effective by comparing VO and BDFS HATS variants with and without `vertex_data` prefetching. Prefetching accounts for about a third of the speedup achieved by BDFS-HATS over VO. (Note that these prefetches are irregular, so a conventional prefetcher would not be able to perform them.)

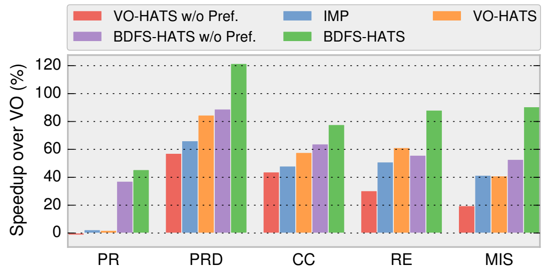


Fig. 23: Impact of prefetching on performance.

We find that HATS prefetches are timely. First, the limited size of the HATS buffer (64 entries) constrains how far ahead the HATS runs. Thus, prefetched data takes a small fraction of the L2 (up to 4 KB), avoiding too-early prefetches. Second, the HATS buffer is large enough to avoid late prefetches. We observe that a small fraction (5-10%) of prefetches are late (i.e., partially overlapped with the demand access). Even then, these late prefetches cover 90% of access latency on average.

HATS location: Fig. 24 shows how the location of HATS changes the benefits of BDFS-HATS over VO. Performance changes only slightly when moving HATS from the L2 to the L1. However, placing HATS near the LLC (e.g., on a shared FPGA fabric) causes a noticeable drop in performance for non-all-active algorithms. With this configuration, HATS can only prefetch `vertex_data` into the LLC. Thus, cores experience few tens of cycles of latency when accessing `vertex_data`.

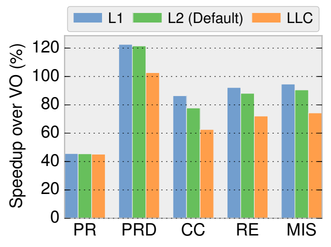


Fig. 24: Sensitivity of BDFS-HATS to on-chip location.

Memory bandwidth: Fig. 25 shows the speedup of VO-HATS and BDFS-HATS over VO as the number of memory controllers grows from two to six (i.e., as peak main memory bandwidth

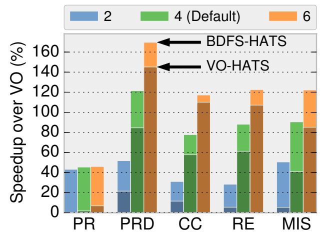


Fig. 25: Speedup of VO-HATS (shaded) and BDFS-HATS over VO with 2–6 memory ctlrs.

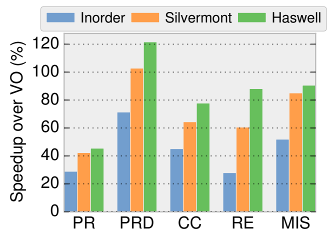


Fig. 26: Speedup of BDFS-HATS with different cores, over VO with Haswell cores.

grows from about 26 to 77 GB/s). While both VO-HATS and BDFS-HATS are more beneficial when the system has higher bandwidth, the improvement of BDFS-HATS over VO-HATS increases when the system has lower bandwidth. At two memory controllers, BDFS-HATS outperforms VO-HATS by 43%, 25%, 18%, 22%, and 43%. At six memory controllers, the speedups reduce to 37%, 10%, 3%, 8%, and 20%.

General-purpose core type: Fig. 26 shows the speedup of BDFS-HATS different core types. All speedups are normalized to VO with Haswell-like cores. While compute-bound non-all-active algorithms (`PRD, CC, RE`) are more sensitive, BDFS-HATS retains a large fraction of its benefits with lean OOO (Silvermont-like) cores since the system is memory bandwidth-bound. Moreover, HATS with energy-efficient in-order cores is faster than software VO with power-hungry OOO cores.

Cache size: Fig. 27 shows the performance of VO-HATS and BDFS-HATS at various cache sizes. All speedups are relative to software VO at a fixed cache size of 32 MB, making speedups across cache sizes comparable. For `PR` and `MIS`, BDFS-HATS with just a 16 MB cache outperforms both VO-HATS and VO with a 32 MB cache. For `PRD` and `RE`, BDFS-HATS at 16 MB outperforms VO with a 32 MB cache. It closely matches VO-HATS with a 32 MB cache.

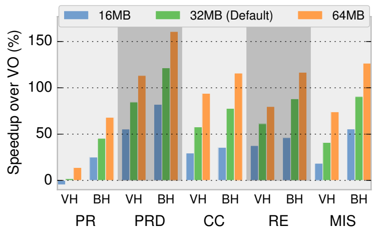


Fig. 27: Sensitivity to LLC size. All speedups are relative to software VO at 32 MB.

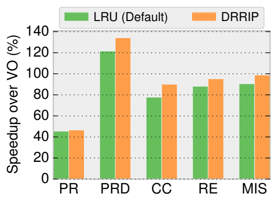


Fig. 28: BDFS-HATS with different LLC replacement policies.

LLC replacement policy: Finally, Fig. 28 shows how the LLC replacement policy affects the benefits of BDFS. We evaluate BDFS-HATS with LRU and DRRIP [25], a high-performance replacement policy. BDFS-HATS achieves slightly higher gains when using DRRIP. This happens because DRRIP is scan- and thrash-resistant, so it reduces the cache pollution caused by access patterns with no temporal locality. This leaves more cache capacity for data with temporal locality, which BDFS exploits. These results show that BDFS-HATS and high-performance replacement policies are complementary. BDFS-HATS would also benefit from specializing the replacement policy for different graph data structures [38].

VI. Additional Related Work

A. *Memory system specialization for graph processing*

Recent graph processing accelerators have proposed various ways to tune the memory hierarchy to the needs of graph algorithms, e.g., by using separate scratchpads to hold vertex and edge data and matching their word sizes to object sizes [44], or by adding a large on-chip eDRAM scratchpad that can hold larger graphs than is possible with SRAM [22].

Prior work has also proposed near-data processing (NDP) designs [2, 20, 40, 50, 60] that execute most graph-processing operations in logic close to main memory, reducing the cost of memory accesses. NDP's high memory bandwidth makes it attractive for processing large *unstructured* graphs without any locality, but as we have seen, graphs often have community structure and can use caches effectively. Moreover, NDP systems are still under development, so it is important to optimize systems that use conventional off-chip main memory.

HATS complements this prior work by using locality-aware scheduling to make better use of limited on-chip storage.

B. *Preprocessing and locality optimizations*

As discussed in Sec. II, preprocessing algorithms such as RCM [14], GOrder [55], and Rabbit Order [4] improve the locality of VO, but they are very expensive. These techniques reorder graph vertices to assign close-by ids to related vertices. They turn the graph's adjacency matrix into a narrow band matrix, with most nonzeros clustered around its diagonal. However, this reordering is orders of magnitude more expensive than the runtime of a single traversal [33, 39]. Similarly, edge-centric scheduling with Hilbert Order [36] outperforms VO by balancing the locality of source and destination vertices but requires an expensive sort of all edges.

VII. Conclusion

Graph processing algorithms are increasingly bottlenecked by main memory accesses. We have shown how runtime scheduling strategies that exploit the community structure of real-world graphs can significantly improve locality. We propose bounded depth-first scheduling (BDFS), a simple yet highly effective scheduling technique to improve locality for graphs with good community structure, and HATS, a hardware-accelerated traversal scheduler. BDFS-HATS requires inexpensive hardware and reduces memory accesses significantly. On a simulated 16-core system, BDFS reduces main memory accesses by up to 2.4× and BDFS-HATS improves performance by up to 3.1× over a locality-oblivious software implementation and by up to 2.1× over specialized prefetchers.

Acknowledgments

We thank Nosayba El-Sayed, Joel Emer, Yee Ling Gan, Mark Jeffrey, Hyun Ryong Lee, Suvinay Subramanian, Po-An Tsai, Victor Ying, Guowei Zhang, and the anonymous reviewers for their feedback. We thank Scott Beamer for sharing his implementation of Propagation Blocking. This work was supported in part by NSF grant CAREER-1452994, DARPA SDH under contract HR0011-18-3-0007, and a grant from the Qatar Computing Research Institute. Nathan Beckmann was supported by a Google Faculty Research Award.

References

[1] U. A. Acar, A. Charguéraud, and M. Rainey, "A work-efficient algorithm for parallel unordered depth-first search," in *Proc. SC15*, 2015.
[2] 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.
[3] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proc. ICS'16*, 2016.

[4] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, “Rabbit order: Just-in-time parallel reordering for fast graph analysis,” in *Proc. IPDPS*, 2016.

[5] J. Banerjee, W. Kim, S.-J. Kim, and J. F. Garza, “Clustering a DAG for CAD Databases,” *IEEE TSE*, 1988.

[6] A.-L. Barabási and R. Albert, “Emergence of scaling in random networks,” *Science*, vol. 286, no. 5439, 1999.

[7] S. Beamer, K. Asanovic, and D. Patterson, “Locality exists in graph processing: Workload characterization on an Ivy Bridge server,” in *Proc. IISWC*, 2015.

[8] S. Beamer, K. Asanovic, and D. Patterson, “Reducing Pagerank communication via Propagation Blocking,” in *Proc. IPDPS*, 2017.

[9] J. C. Beck and L. Perron, “Discrepancy-bounded depth first search,” in *Proc. CPAIOR-2*, 2000.

[10] G. E. Blelloch, J. T. Fineman, and J. Shun, “Greedy sequential maximal independent set and matching are parallel on average,” in *Proc. SPAA*, 2012.

[11] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *JACM*, 1999.

[12] A. Buluc and J. R. Gilbert, “On the representation and multiplication of hypersparse matrices,” in *Proc. IPDPS*, 2008.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. MIT press, 2009.

[14] E. Cuthill and J. McKee, “Reducing the bandwidth of sparse symmetric matrices,” in *Proc. ACM*, 1969.

[15] 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.

[16] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM TOMS*, vol. 38, no. 1, 2011.

[17] J. Doweck, “Inside Intel Core microarchitecture,” in *IEEE Hot Chips Symposium*, 2006.

[18] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, “Cache-oblivious algorithms,” in *Proc. FOCS-40*, 1999.

[19] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the Cilk-5 multithreaded language,” in *Proc. PLDI*, 1998.

[20] M. Gao, G. Ayers, and C. Kozyrakis, “Practical near-data processing for in-memory analytics frameworks,” in *Proc. PACT-24*, 2015.

[21] J. Gonzalez, R. Xin, A. Dave, D. Crankshaw, M. Franklin, and I. Stoica, “GraphX: Graph processing in a distributed dataflow framework,” in *Proc. OSDI-11*, 2014.

[22] 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.

[23] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon *et al.*, “Haswell: The fourth-generation intel core processor,” *IEEE Micro*, vol. 34, no. 2, 2014.

[24] Intel, “Intel-Altera Heterogeneous Architecture Research Platform Program,” https://cpufpga.files.wordpress.com/2016/04/harp_isca_2016_final.pdf, 2016.

[25] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, “High performance cache replacement using re-reference interval prediction (RRIP),” in *Proc. ISCA-37*, 2010.

[26] V. Kiriansky, Y. Zhang, and S. Amarasinghe, “Optimizing indirect memory references with milk,” in *Proc. PACT-25*, 2016.

[27] R. E. Korf, “Depth-first iterative-deepening: An optimal admissible tree search,” in *JAI*, 1985.

[28] H. Kwak, C. Lee, H. Park, and S. Moon, “What is Twitter, a social network or a news media?” in *Proc. WWW-19*, 2010.

[29] 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-17*, 2008.

[30] 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.

[31] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, “Challenges in parallel graph processing,” *PPL*, vol. 17, no. 01, 2007.

[32] C. Magnien, M. Latapy, and M. Habib, “Fast computation of empirically tight bounds for the diameter of massive graphs,” *JEA*, vol. 13, 2009.

[33] J. Malicevic, B. J. E. Lepers, and W. Zwaenepoel, “Everything you always wanted to know about multicore graph processing but were afraid to ask,” in *USENIX ATC*, 2017.

[34] A. McGregor, “Graph stream algorithms: A survey,” *ACM SIGMOD Record*, vol. 43, no. 1, 2014.

[35] F. McSherry, “A uniform approach to accelerated PageRank computation,” in *Proc. WWW-14*, 2005.

[36] F. McSherry, M. Isard, and D. G. Murray, “Scalability! But at what COST?” in *Proc. HotOS-15*, 2015.

[37] Micron, “1.35V DDR3L power calculator (4Gb x16 chips),” 2013.

[38] A. Mukkara, N. Beckmann, and D. Sanchez, “Whirlpool: Improving dynamic cache management with static data classification,” in *Proc. ASPLOS-XXI*, 2016.

[39] A. Mukkara, N. Beckmann, and D. Sanchez, “Cache-Guided Scheduling: Exploiting caches to maximize locality in graph processing,” in *AGP’17*, 2017.

[40] 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.

[41] D. Nguyen, A. Lenharth, and K. Pingali, “A lightweight infrastructure for graph analytics,” in *Proc. SOSP-24*, 2013.

[42] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, “GraphGen: An FPGA framework for vertex-centric graph computation,” in *Proc. FCCM-22*, 2014.

[43] T. Oguntebi and K. Olukotun, “GraphOps: A dataflow library for graph analytics acceleration,” in *Proc. FPGA’16*, 2016.

[44] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, “Energy efficient architecture for graph analytics accelerators,” in *Proc. ISCA-43*, 2016.

[45] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the web.” Stanford InfoLab, Tech. Rep., 1999.

[46] P.K. Gupta, “Accelerating datacenter workloads,” FPL’16 Keynote, http://www.fpl2016.org/slides/Gupta--AcceleratingDatacenterWorkloads.pdf, 2016.

[47] D. Sanchez and C. Kozyrakis, “ZSim: Fast and accurate microarchitectural simulation of thousand-core systems,” in *Proc. ISCA-40*, 2013.

[48] J. Shun and G. E. Blelloch, “Ligra: A lightweight graph processing framework for shared memory,” in *Proc. PPoPP*, 2013.

[49] J. E. Smith, “Decoupled access/execute computer architectures,” in *Proc. ISCA-9*, 1982.

[50] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, “GraphR: Accelerating graph processing using ReRAM,” in *Proc. HPCA-24*, 2018.

[51] M. Stickel and W. Tyson, “An analysis of consecutively bounded depth-first search with applications in automated deduction,” in *IJCAI-9*, 1985.

[52] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, “GraphMat: High performance graph analytics made productive,” *PVLDB*, 2015.

[53] N. Topham and K. McDougall, “Performance of the decoupled ACRI-1 architecture: The perfect club,” in *Proc. HPCN*, 1995.

[54] L. G. Valiant, “A bridging model for parallel computation,” *Comm. ACM*, vol. 33, no. 8, 1990.

[55] H. Wei, J. X. Yu, C. Lu, and X. Lin, “Speedup graph processing by graph ordering,” in *Proc. SIGMOD’16*, 2016.

[56] Xilinx, “ZC706 evaluation board for the Zynq-7000 XC7Z045 all programmable SoC user guide,” https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf, 2017.

[57] Xilinx, “Ultrascale+ FPGAs product tables and product selection guide,” https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf, 2018.

[58] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, “IMP: Indirect Memory Prefetcher,” in *Proc. MICRO-48*, 2015.

[59] P. Yuan, C. Xie, L. Liu, and H. Jin, “PathGraph: A path centric graph processing system,” *IEEE TPDS*, 2016.

[60] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, “GraphP: Reducing communication for PIM-based graph processing with efficient data partition,” in *Proc. HPCA-24*, 2018.

[61] Y. Zhang, X. Liao, H. Jin, L. Gu, and B. B. Zhou, “FBSGraph: Accelerating asynchronous graph processing via forward and backward sweeping,” *IEEE TKDE*, 2018.

[62] Y. Zhang, V. Kiriansky, C. Mendis, M. Zaharia, and S. Amarasinghe, “Making caches work for graph analytics,” *IEEE BigData*, 2017.

[63] X. Zhu, W. Chen, W. Zheng, and X. Ma, “Gemini: A computation-centric distributed graph processing system,” in *Proc. OSDI-12*, 2016.

[64] X. Zhu, W. Han, and W. Chen, “GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning,” in *USENIX ATC*, 2015.