SAGA-Bench: Software and Hardware Characterization of StreAming Graph Analytics Workloads

Abanti Basak*, Jilan Lin*, Ryan Lorica*, Xinfeng Xie*, Zeshan Chishti†, Alaa Alameldeen†, Yuan Xie*
*University of California, Santa Barbara** *Intel Labs†*
{abasak,jilan,xinfeng,yuanxie}@ucsb.edu, {zeshan.a.chishti,alaa.r.alameldeen}@intel.com

***Abstract*—Many application scenarios such as social network analysis and real-time financial fraud detection involve performing batched updates and analytics on a time-evolving or streaming graph. Despite their importance, streaming graph analytics workloads have not been systematically studied at either the software or the architecture levels. This paper fills this gap through three contributions.**

First, we develop and open-source SAGA-Bench, a benchmark for StreAming Graph Analytics, which puts together different data structures and compute models on the same platform for a fair and systematic characterization.

Second, we perform software-level characterization using SAGA-Bench. Our profiling reveals that the best data structure for a streaming graph depends on the per-batch degree distribution of the graph. We also observe that the incremental compute model provides performance benefits especially for larger graphs. Finally, we show that the graph update phase contributes at least 40% of the streaming graph processing latency in many cases.

Third, we perform workload characterization at the architecture level. Our study reveals that the graph update phase exhibits lower utilization of architecture resources than the compute phase. Furthermore, the hardware resource utilization of the update phase strongly depends on the underlying structure of the batches of the graph. Finally, between compute and update phases, the former exhibits a higher L3 cache hit ratio, whereas the latter shows a higher L2 cache hit ratio.

I. Introduction

Streaming graph processing involves performing batched updates and analytics on graphs that are evolving over time. This scenario is critical in many applications such as social network analysis [1]–[3], real-time financial fraud detection [4], anomaly detection [5], and recommendation systems [6]–[8]. These application scenarios require effective handling of the streaming graph data by providing low-latency real-time support for both: 1) update (ingestion of new edges) and 2) compute (timely analytics on freshly ingested graph data stream). The data structures and compute models underpinning streaming graph systems are still actively being researched. Although many systems have been proposed, these workloads have not been studied *systematically* at the software level. Moreover, they remain unexplored at the architecture level.

At the software level, a lack of systematicness arises from the heterogeneity of previously proposed streaming graph systems [1], [3], [9]–[16]. In addition to the core software components (data structures and compute models), each system is accompanied with additional optimization features (e.g., data compression, specially designed APIs, specialized memory allocation schemes). Moreover, measurement methods often vary across these systems. Hence, it is difficult to perform a fair and systematic comparison of the basic data structures and compute models across these systems since the observed performance differences may arise from a variety of features.

At the architecture level, streaming graphs remain unexplored. Prior research [17]–[35] focuses on *static* graphs which assumes that a graph is built once and never changes while algorithms are run on it. Although Dai *et al.* [35] address dynamic graphs, the discussion is limited to an additional optimization for a newly proposed design for static graph analytics and does not consider a detailed workload characterization. We recognize two reasons for which streaming graphs have failed to receive attention at the architecture level. First, data structures and compute models are still being developed, indicating the lack of software-level maturity. Second, the lack of an open-source benchmark containing the core data structures and compute models limits microarchitecture exploration. Existing open-source implementations for streaming graphs [3], [7], [9], [13], [16] are *holistic systems*, each with a specialized complex package of system-specific optimizations. A more useful resource for microarchitecture exploration is an open-source *benchmark* with the essential software techniques (data structures and compute models) to understand the core complexities of the workloads without system-specific optimizations. However, such a benchmark for streaming graphs is currently missing in the architecture community.

Motivated by the lack of systematic software and hardware studies, this paper presents three contributions:

Contribution 1: Development of SAGA-Bench (Section III). SAGA-Bench is an open-source C++ benchmark for StreAming Graph Analytics containing a collection of data structures and compute models on the same platform for a fair and systematic study. SAGA-Bench does not seek to be yet another novel and competitive state-of-the-art streaming graph system. Instead, it is a systematic performance analysis platform for software and hardware studies of the essential data structures and compute models proposed across various existing systems. For software studies, the core data structures

978-1-7281-4798-7/20/$31.00 ©2020 IEEE
DOI 10.1109/ISPASS48437.2020.00012

and compute models (without system-specific optimizations) are integrated into SAGA-Bench and evaluated using the same measurement methodology (thus alleviating the problem of difficult-to-interpret comparisons across heterogeneous systems). At the architecture level, SAGA-Bench provides an open-source benchmark for streaming graph workloads. Since streaming graphs are still being actively researched at the software level, the goal of SAGA-Bench is to remain in active development over time through progressive integration of future novel data structures and compute models. *To the best of our knowledge, our work is the first to develop a resource for streaming graphs which simultaneously provides 1) a common platform for performance analysis studies of software techniques and 2) a benchmark for architecture studies* (code available at **https://github.com/abasak24/SAGA-Bench**).

Contribution 2: Software-level Workload Characterization (Section V). We further use SAGA-Bench to perform software-level profiling to provide insights on the best data structure and compute model. This analysis is important because 1) data structures and compute models are still topics of active research and 2) we seek to demystify the performance trade-offs of different data structures and compute models systematically on the same platform, as opposed to prior difficult-to-interpret cross-system comparisons. In addition, this software-level analysis helps identify the best software for further architecture characterization (see Contribution 3). Our key findings from software-level profiling are as follows:

- The best data structure for a streaming graph depends on the per-batch degree distribution of the graph. Short-tailed graphs perform the best on adjacency list (occasionally Stinger [9]), whereas hash-based data structure is the most scalable for heavy-tailed graphs.
- The incremental compute model offers performance benefits especially for larger graphs. Although an intuitive finding, we provide detailed supporting data to quantitatively confirm this observation.
- The graph update phase contributes at least 40% of the streaming graph processing latency for many workloads.

Beyond prior work, 1) we provide novel insights on the comparative performance trade-offs of various data structures on input datasets of different structural properties and 2) we explicitly highlight the performance limitation of the graph update phase in terms of the latency breakdown.

Contribution 3: Architecture-level Workload Characterization (Section VI). We use the best data structure and compute model from software-level profiling to perform architecture-level characterization of both update and compute phases. Our key observations and insights are as follows:

- The graph update phase exhibits lower utilization of hardware resources than the graph compute phase, indicating lower thread-level parallelism (TLP) of the update phase.
- The hardware resource utilization of the update phase strongly depends on the underlying structure of the batches of the graph. The update of heavy-tailed graphs benefits negligibly from larger core counts, memory bandwidth, and inter-socket bandwidth. In contrast, the update of short-tailed graphs shows higher utilization of these architecture resources. We further provide insights that the lower TLP of the update phase arises from 1) thread contentions in short-tailed graphs and 2) workload imbalance in heavy-tailed graphs.
- Compared to the update phase, the compute phase exhibits higher L3 cache hit ratio. In contrast, the update phase exhibits a higher L2 cache hit ratio than the compute phase. This occurs due to 1) a data reuse relationship and 2) a difference in working set sizes between the update and compute phases.

To the best of our knowledge, our work is the first to perform a comparative study between update and compute phases and to provide novel insights on the architecture-level features of the graph update phase. Previous architecture-level research on graph processing [17]–[35] focuses on static graphs and does not consider a detailed study of the graph update phase.

II. Background and Motivation

We describe how streaming graph analytics differs from static graph analytics in terms of execution flow, coverage in previous work, and unique challenges.

A. Streaming vs Static Graph Analytics

As shown in Fig. 1, the input to a streaming graph analytics system is a stream of incoming edges. Once a **batch** of edges enters the system, two action phases described below are executed, which provide newly computed results:

- ***Update phase***: the incoming edges in a given batch are ingested into the graph data structure.
- ***Compute phase***: an algorithm such as PageRank is performed on the freshly updated data structure.

The primary optimization target in streaming graph analytics is timely response, i.e., low latency between the input edge batch and the newly computed results. Hence, in the rest of the paper, we use *batch processing latency* as the performance metric for streaming graphs.

$$\text{batch processing latency}_{batch\ i} = \text{update latency}_{batch\ i} + \text{compute latency}_{batch\ i} \quad (1)$$

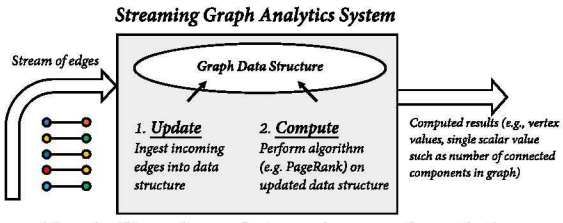


Fig. 1: Overview of streaming graph analytics

Fig. 2 shows the difference in execution flow between static and streaming graph analytics. In the former, an entire input file is read to build a graph usually in the Compressed Sparse Row (CSR) format [36]. It is then assumed that the graph topology never changes as different algorithms are run on it. Streaming graph analytics, on the other hand, has to

handle dynamism by performing repeated update and compute operations on continuous batches of incoming edges[1].

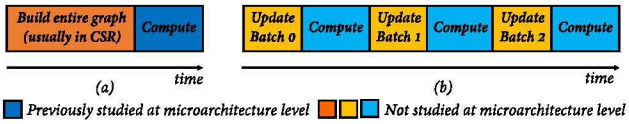


Fig. 2: Execution flow of (a) static and (b) streaming graphs

B. Coverage of Previous Studies

At the software level, there exists an abundance of both standalone systems [48]–[53] and systematic studies [54]–[57] for static graph processing. However, for streaming graphs, although many novel systems have been proposed [1], [3], [9]–[16], there is a lack of systematic and comparative study of the techniques proposed across various systems. We fill this gap through a systematic performance analysis of different data structures and compute models.

Fig. 2 highlights that only the compute phase in static graph analytics has been previously studied [17]–[35] at the architecture level (the graph building phase has not been considered in detail). In contrast, the entire execution flow of streaming graph analytics still remains unexplored due to immature software and lack of a benchmark. This paper fills this gap by creating SAGA-Bench and performing workload characterization at the architecture level.

C. Challenges of Streaming Graph Analytics

Streaming graph analytics possesses unique challenges because its optimization goal is different from that of static graph analytics. In static graphs, the optimization target is the compute phase. The graph building phase is considered to be a fixed one-time overhead that can be amortized by performing repeated computations. In contrast, the optimization goal in streaming graphs is real-timeliness (Equation 1). Hence, the *graph update phase lies on the critical path for streaming graphs and cannot be considered as a one-time overhead.* This is the primary factor which hinders smooth portability of static graphs' software-hardware solutions to streaming graphs.

Inefficiency of borrowing software solutions from static graph analytics: Borrowing array-based CSR and preprocessing techniques [58] beneficial for the compute phase would substantially hurt the update latency. Similarly, borrowing conventional algorithms [36], [59] would lead to redundant computations because two successive compute phases may have large overlap in vertices and edges.

Inefficiency of borrowing architecture solutions from static graph analytics: Previous architecture optimizations for static graph analytics ignore the update or graph building phase. This is inefficient for streaming graphs because update lies on the critical path and is interleaved with compute. For the compute phase, previous architecture optimizations in static graphs assume the conventional CSR data layout and algorithms. Streaming graphs, however, rely on a set of different data structures and compute models. Without extensive hardware characterization of these novel underlying software components, it is unclear whether an architecture optimization targeted at the compute for static graphs would work equally well in the streaming scenario.

III. SAGA-Bench Description

SAGA-Bench is implemented in C++ and contains a collection of 4 data structures (Section III-A), 2 compute models (Section III-B), and 6 vertex-centric algorithms (Section III-C) implemented in both the compute models[2].

A. Data Structures for Graph Topology

SAGA-Bench contains four vertex-centric data structures which support multithreaded edge update, as described below[3], for storing the graph *topology*[4]. As described in prior work [9], [10], we implement each edge update only after a search operation so that edges are ingested uniquely.

*1) **Adjacency List (shared style multithreading) (AS):*** AS is implemented as an array of vectors where each vector contains the neighbors belonging to a particular node. Multiple threads update a batch of edges into AS (implemented in the code with OpenMP). A thread responsible for an edge update 1) locks the vector corresponding to the source node, 2) scans the vector to search for the target edge, and 3) inserts the edge if the search is negative. Since edge update involves locking the entire vector corresponding to a source node, there is no parallelism in intra-node edge update. However, parallelism is possible in updating edges for *different* nodes.

*2) **Adjacency List (chunked style multithreading) (AC):*** As shown in Fig. 3, AC is an adjacency list partitioned into multiple chunks, each chunk storing neighbors for a subset of source vertices. Each chunk is a single-threaded data structure and no locks are required for updating the edges inside it (the rest of the intra-chunk operation is the same as in AS). Update multithreading is achieved with multiple chunks.

*3) **Stinger:*** Stinger [9] is a shared-memory data structure where multithreading is achieved with OpenMP. As shown in Fig. 4, it contains two components. First, an array stores information on the source node ID and its degree. Second, each node entry in the array points to a linked list of edgeblocks which contains the edge information for the corresponding node. Each edgeblock accommodates a fixed number of edges (16 in our implementation). Stinger differs from AS in two

[1]The current version of SAGA-Bench 1) maintains the latest snapshot of an evolving graph similar to [1], [9], [14] and 2) supports the model where update and compute are interleaved (Fig. 2b) similar to [9], [11], [12], [15], [37]–[43]. A few existing systems maintain multiple over-time snapshots [2], [44]–[47]. Two very recently proposed systems [13], [16] utilize data structures capable of parallelizing update and compute. The multi-snapshot model and the novel data structures will be included in the future version of SAGA-Bench.

[2]All implementations are done from scratch. Even when an open-source implementation is available (e.g., Stinger [9]), it is modified to conform to the APIs of SAGA-Bench. Closed-source software techniques are implemented by closely following their descriptions in the corresponding published papers.

[3]The description assumes storing out-neighbors. For directed graphs, there is a second copy of the data structure for storing in-neighbors.

[4]Vertex property values are maintained in a separate array in all cases.

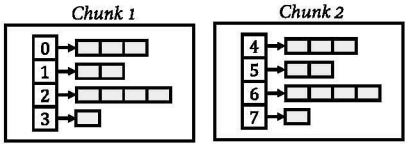


Fig. 3: Chunked adjacency list (AC)

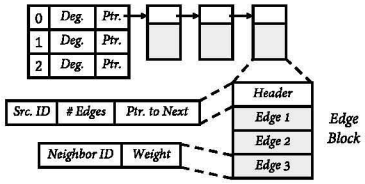


Fig. 4: Stinger

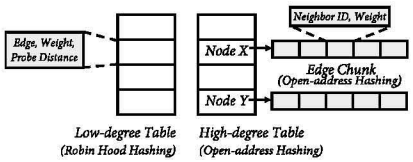


Fig. 5: Degree-aware Hashing (DAH)

aspects. First, unlike AS, Stinger can enjoy intra-node parallelism. Due to fragmented edgeblocks, Stinger can perform multiple edge updates for a single node by acquiring fine-grained locks within its linked list of edgeblocks. Second, unlike AS, Stinger requires two scans for an edge insertion as a trade-off for its fine-grained locks. The first scan through the linked list searches for the target edge. If not found, another scan through the same linked list is required to find an empty space for inserting the edge.

*4) **Degree-aware Hashing (DAH)**:* As shown in Fig. 5, DAH [10] contains two hash tables, one for low-degree vertices and another for high-degree vertices. Multithreading is achieved with multiple chunks of DAH, where each chunk is single-threaded and lockless (similar to AC). Although DAH allows amortized constant-time edge update through hashing, it incurs an overhead from the following meta-operations due to its degree-awareness: 1) querying the degree of each table before deciding where to place the new edge and 2) periodic flushing of edge information from the low-degree table to the high-degree table.

*5) **Choice of data structures**:* To enable systematic and insightful studies, the four data structures of SAGA-Bench have been chosen to include variations in the following factors.
Edge update mechanism: To study the effect of the update technique on update latency, SAGA-Bench contains a variety of update mechanisms: 1) hash-based update (DAH), 2) update in memory-contiguous vectors (AS, AC), and 3) update in coarse-grained linked lists (Stinger).
Intra-node parallelism: Stinger supports intra-node parallelism in edge update, whereas others do not possess this flexibility (AS, AC, DAH). This allows us to study the benefit of this extra degree of parallelism on the update latency.
Multithreading technique: SAGA-Bench contains data structures with two update multithreading techniques: 1) shared-memory style (AS, Stinger) and 2) chunked style (AC, DAH). We implemented adjacency list in both the techniques in order to understand, for a given data structure, the benefit of one over the other for datasets of different properties.
Traversal mechanism: Graph data layout and compute latency are strongly tied because a basic operation of vertex-centric computation is neighbor traversal for vertices. Data structures in SAGA-Bench support a variety of traversal mechanisms in order to study their effects on the compute latency.

B. Compute Models

SAGA-Bench supports two compute models:
Recomputation from scratch (FS). Every update phase is considered to produce a brand-new version of the entire graph.

Algorithm 1 Incremental PageRank

Require: Streaming graph $\mathcal{G}\{V, E\}$ which contains $|V|$ vertices and $|E|$ edges as of the latest update phase; PageRank scores $\{PR(v_j)\}$ from previous batch; array of affected vertices *affected*.

1: Initialize: two queues Q_{curr}, Q_{next}; *visited* bitvector of size $|V|$; triggering threshold $\epsilon = 10^{-7}$.
2: **for** v_i in V **do**
3: **if** v_i is a new vertex **then**
4: $PR(v_i) = 1/|V|$
5:
6: *# pragma omp parallel for*
7: **for** i in range($|V|$) **do**
8: **if** *affected*$[i] == true$ **then**
9: $old\_score = PR(v_i)$
10: Re-calculate $PR(v_i)$
11: **if** $|old\_score - PR(v_i)| > \epsilon$ **then**
12: **for** v_j in v_i's out-neighbors **do**
13: **if** $visited[j] == false$ **then**
14: **if** $CAS(visited[j], false, true)$ **then**
15: $Q_{next}.push\_back(v_j)$
16:
17: $Q_{curr} = Q_{next}$
18: $Q_{next}.clear()$
19: **while** Q_{curr} *is not empty* **do**
20: $visited \leftarrow \{false\}$
21: *# pragma omp parallel for*
22: **for** v_j in Q_{curr} **do**
23: Re-do lines 9-15.
24: $Q_{curr} = Q_{next}$
25: $Q_{next}.clear()$

All vertex values are reset to the initial values and an entirely new computation is started on this fresh graph, i.e., the current computation is oblivious of the computation performed in the previous batch. This compute model is implemented using conventional algorithms for static graphs (borrowed from GAP [36]) during each successive compute phase. We implement Max Computation and Single Source Widest Paths (Section III-C) since they are not implemented in GAP.
Incremental computation (INC). This compute model considers the fact that there may be sharing of vertices and edges between two successive compute phases. Hence, the amount of work may be saved by 1) reusing the outcome of the computations performed in the previous batch and by 2) performing computation on only the portion of the graph

TABLE I: Vertex functions for algorithms

Alg	Vertex function
BFS	$v.depth \leftarrow \min_{e \in InEdges(v)}(e.source.depth + 1)$ [11]
CC	$v.value \leftarrow \min(v.value, \min_{e \in Edges(v)} e.other.value)$ [60]
MC	$v.value \leftarrow \max(v.value, \max_{e \in InEdges(v)}(e.source.value))$ [11]
PR	$v.rank \leftarrow 0.15 + 0.85 * \sum_{e \in InEdges(v)} e.source.rank$ [60]
SSSP	$v.path \leftarrow \min_{e \in InEdges(v)}(e.source.path + e.weight)$ [11]
SSWP	$v.path \leftarrow \max_{e \in InEdges(v)}(min(e.source.path, e.weight))$ [11]

TABLE II: Evaluated datasets. BatchCount computed with batch size of 500K (see Section IV-B).

Dataset	vertices	edges	batchCount
Livejournal (LJ)	4,847,571	68,993,773	138
Orkut	3,072,441	117,185,083	235
RMAT	32,118,308	500,000,000	1000
wiki-topcats (Wiki)	1,791,489	28,511,807	58
wiki-talk (Talk)	2,394,385	5,021,410	11

affected (directly and indirectly) by the latest update phase. We implement incremental algorithms in SAGA-Bench using two techniques introduced in previous work (pseudocode in Algorithm 1):

- *Processing amortization* [11], [60]: Work is saved by starting the computation from the vertex values right before the latest update, i.e., the vertex values produced by the compute phase on the previous batch (lines 2-4). These intermediate values have been shown to be closer to the final results, allowing faster convergence to the final results in cases of many algorithms.
- *Selective triggering* [1], [15]: Computation starts from a subset of vertices affected by the latest update (line 8), and large enough changes (decided by a triggering condition in line 11) are progressively propagated iteration-by-iteration to neighboring vertices. These iterations continue until no more vertices are triggered (lines 19-25). The goal is to cut computation costs by operating on only a fraction of the graph affected (directly and indirectly) by the latest update instead of on the entire graph.

C. Algorithms

As summarized in Table I, SAGA-Bench contains six vertex-centric algorithms implemented in both FS and INC compute models: 1) Breadth First Search (BFS), 2) Connected Components (CC), 3) Max Computation (MC), 4) PageRank (PR), 5) Single Source Shortest Path (SSSP), and 6) Single Source Widest Path (SSWP).

D. API and extensibility

The API of SAGA-Bench is general enough to accommodate future software techniques. The API includes functions that define batched updates, graph traversal, and algorithm execution (specific functions: *update()*, *in_neigh()*, *out_neigh()*, and *performAlg()*). A new data structure, compute model, or graph algorithm can be added in the future by implementing these API functions.

IV. Experimental Setup

In this section, we describe the experimental platform, methodology, and datasets.

A. Platform

Characterization is performed on a dual-socket Intel Xeon Gold 6142 (Skylake) server with 16 physical cores per socket and 2-way simultaneous multithreading per physical core (total of 64 hardware execution threads in the system). The server contains 32KB private L1 data and instruction caches per physical core, 1MB private L2 cache per physical core, 22MB shared last-level cache (LLC) per socket, and 768GB DRAM with maximum per-socket memory bandwidth of 128GB/s. Three QuickPath Interconnect (QPI) links provide 136.2GB/s of inter-socket communication (68.1GB/s in each direction).

B. Methodology

SAGA-Bench is compiled with gcc-7.3.1. All experiments (except for studies on core scaling in Section VI) are performed with 64 threads, the maximum number of hardware execution threads. To make our analysis reproducible, we turn off the Turbo Boost feature for all experiments. In addition, we pin software threads to hardware threads to exclude performance variations due to OS thread scheduling.

Graph datasets (Section IV-C) are first randomly shuffled to break any ordering in the input files. This is done to ensure the realistic scenario that streaming edges are not likely to come in any pre-defined order. The shuffled input file is then read in batches of 500K edges (similar batch size value has been considered in [9], [12]–[14]).

All the experiments are repeated three times and each experiment provides batchCount (see Table II) values. To analyze the over-time effect of changing graph size and sparsity in streaming graphs, we divide the total number of batches in a given experiment into three equal stages. Experimental results contain three representative data points P1, P2, and P3, which are the averages for early, middle, and final stages, respectively. The average for a given stage (P1, P2, or P3) is computed by taking into account 1) the corresponding one-third of batchCount values and 2) the fact that the experiment has been repeated three times. For example, for BFS run in the incremental compute model on the Orkut dataset on the AS data structure, the batch processing latency at P1 is the average of 1/3 $\times$ Orkut's batchCount $\times$ 3 latency values produced from the three repeated experiments. All the averages are computed with 95% confidence intervals. Despite three runs, our confidence intervals are tight because each run produces batchCount values which are taken into account (as described above) for the calculation of the average.

Architecture-level profiling of memory, caches, and inter-socket bandwidth (Section VI) is performed with Intel Processor Counter Monitor (PCM) [61].

TABLE III: Best combination of data structure and compute model and the corresponding absolute batch processing latency (in seconds). Conclusion for each entry is derived by comparing 4 data structures × 2 compute models=8 averages with 95% confidence intervals. [x/y=x is the best average but x and y are competitive].

Alg Datset	P1 (early stage)	P2 (middle stage)	P3 (final stage)
BFS LJ	INC/FS+AS 0.1705	INC+AS 0.1502	INC+AS 0.1407
BFS Orkut	INC+AS 0.1521	INC+AS 0.1445	INC+AS 0.2003
BFS RMAT	INC+AS 0.2220	INC+AS 0.2029	INC+AS 0.2190
BFS Wiki	INC/FS+Stinger 0.2587	INC/FS+DAH 0.4063	INC+DAH 0.3757
BFS Talk	INC/FS+DAH/Stinger 0.3406	INC/FS+DAH 0.3330	INC/FS+DAH 0.3225
CC LJ	INC+AS 0.1818	INC+AS 0.1513	INC+AS 0.1374
CC Orkut	INC+AS 0.1486	INC+AS 0.1614	INC+AS 0.1932
CC RMAT	INC+AS 0.2453	INC+AS 0.2517	INC+AS 0.2757
CC Wiki	INC+Stinger 0.2731	INC+DAH 0.4082	INC+DAH 0.3728
CC Talk	INC+DAH/Stinger 0.3525	INC+DAH 0.3438	INC+DAH 0.3315
MC LJ	FS/INC+AS 0.3109	INC/FS+AS 0.3552	INC/FS+AS 0.4097
MC Orkut	FS/INC+AS/Stinger 0.3204	INC+AS 0.4094	INC/FS+AS 0.5208
MC RMAT	INC+AS/Stinger 0.9772	INC+AS/Stinger 1.9038	INC/FS+AS/Stinger 2.5754
MC Wiki	FS/INC+Stinger 0.3435	FS/INC+DAH/Stinger 0.6448	INC/FS+DAH 0.7657
MC Talk	FS/INC+DAH/Stinger 0.3806	INC/FS+DAH 0.3856	INC/FS+DAH 0.3901
PR LJ	INC+Stinger 0.3864	INC+Stinger 0.4397	INC+Stinger 0.4536
PR Orkut	INC+Stinger 0.3091	INC+AS/Stinger 0.3234	INC+AS 0.3578
PR RMAT	INC+Stinger 0.4347	INC+Stinger 0.4319	INC+Stinger 0.4582
PR Wiki	INC+Stinger 0.4311	INC+Stinger 0.6478	INC+DAH 0.7669
PR Talk	INC/FS+Stinger/DAH 0.4969	INC/FS+DAH 0.6649	INC/FS+DAH 0.6175
SSSP LJ	FS+AS/Stinger 0.2664	FS+Stinger/AS 0.2971	FS/INC+Stinger/AS 0.3384
SSSP Orkut	FS+Stinger 0.2785	INC+AS/Stinger 0.3761	INC+AS 0.4254
SSSP RMAT	INC+Stinger/AS 0.4919	INC+AS/Stinger 0.6074	INC+AS/Stinger 0.5069
SSSP Wiki	INC/FS+Stinger 0.3345	FS+DAH/Stinger 0.5756	FS+DAH 0.5718
SSSP Talk	FS/INC+DAH/Stinger 0.3478	FS+DAH 0.3471	FS/INC+DAH 0.3735
SSWP LJ	INC+AS/Stinger 0.2408	INC+AS 0.2078	INC+AS 0.2045
SSWP Orkut	INC+Stinger/AS 0.2064	INC+AS/Stinger 0.2896	INC+AS 0.3309
SSWP RMAT	INC+Stinger 0.2770	INC+AS 0.3070	INC+AS 0.3212
SSWP Wiki	FS/INC+Stinger 0.2863	FS+DAH/Stinger 0.5603	FS+DAH 0.5935
SSWP Talk	INC/FS+DAH/Stinger 0.3531	FS/INC+DAH 0.3841	INC/FS+DAH 0.3524

C. Datasets

The datasets in Table II are taken from SNAP [62], with the exception of synthetic RMAT [63] for which we used parameters a=0.55, b=0.15, c=0.15, d=0.25. Livejournal and Orkut are online social networks, Wiki-topcats is Wikipedia hyperlink graph, and Wiki-talk is Wikipedia communication network. All the datasets are directed except for Orkut.

V. Software-Level Profiling

We perform a systematic characterization of the data structures and the compute models on the same platform to measure their impact on update, compute, and batch-processing latencies for a range of algorithms and datasets.

A. Best Combination of Data Structure and Compute Model

Table III shows, for a given algorithm and dataset, the combination of data structure and compute model which provides the lowest batch processing latency. Moreover, the table shows the best combination over time in the early, middle, and final stages. The observed trends are summarized below.

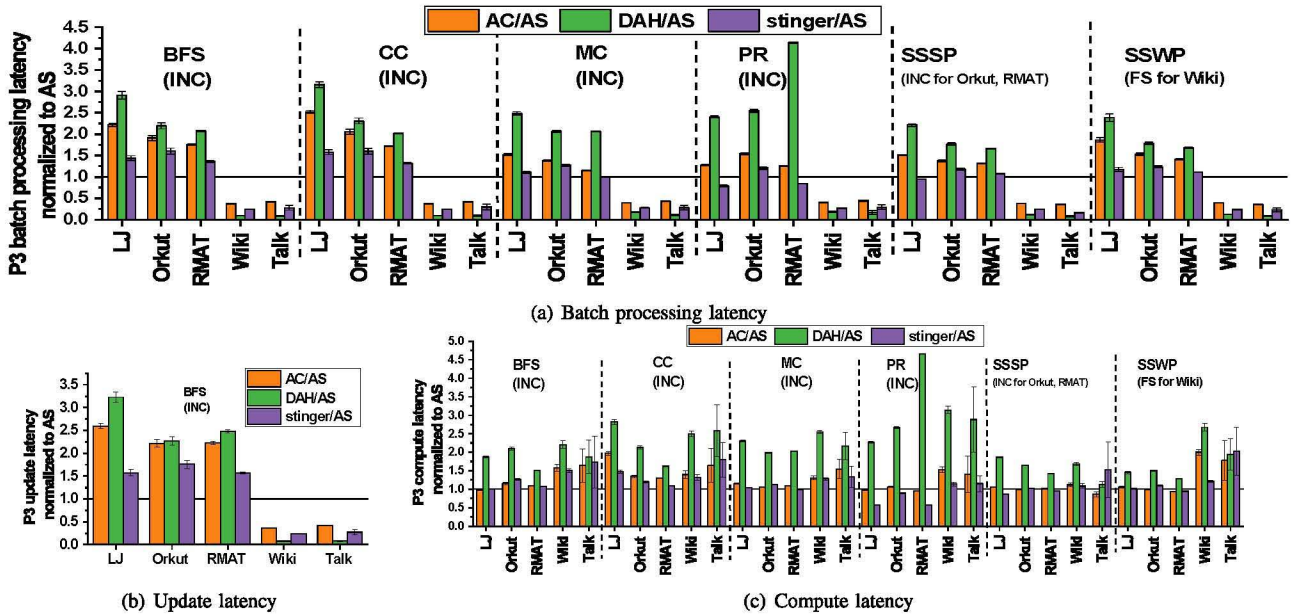
Best data structure. The best data structure depends on the dataset. AS and Stinger are the most competitive data structures over P1, P2, and P3 for LJ, Orkut, and RMAT. For Wiki and Talk, on the other hand, DAH consistently shows good scalability over time, i.e., DAH is the best data structure at P3. Wiki and Talk also exhibit strong over-time variation in the best data structure. Although Stinger starts out being the best or competitive to DAH in P1, DAH finally takes over by the time P3 is reached.

Best compute model. The incremental compute model (INC) is predominantly optimal. However, the recomputation from scratch model (FS) is competitive in a few cases: 1) Wiki and Talk which are small datasets; 2) MC algorithm; and 3) SSSP algorithm except for SSSP on the large RMAT dataset.

B. Impact of Data Structure

Primary Observation: The best data structure for a streaming graph depends on the per-batch degree distribution of the graph. Short-tailed graphs perform the best on AS (occasionally Stinger), whereas DAH is the most scalable data structure for heavy-tailed graphs.

Fig. 6(a) shows, for each algorithm (at the best compute model) and dataset, the total batch processing latency of each data structure normalized to AS at P3. The most striking trend is the flipped relative performance benefits of AS and DAH for different datasets. For LJ, Orkut, and RMAT, AS (occasionally Stinger) provides the lowest batch processing latency and DAH provides the highest latency (1.66×-4.14× higher than AS). For Wiki and Talk, on the other hand, AS is the lowest-performing data structure with 5.6×-12.8× higher latency than DAH, the best-performing data structure. Fig. 6(b) and 6(c) further confirm that this difference is caused by the update phase. Although the relative benefits of AS and DAH are consistent for all datasets in the compute phase, the update phase shows flipped behavior for Wiki and Talk. To understand the graph structural property which affects this behavior, we define



(a) Batch processing latency

(b) Update latency

(c) Compute latency

Fig. 6: (a) Total (b) Update and (c) Compute latencies at P3 of AC, DAH, and Stinger (normalized to AS) at the best compute model (P3 column of Table III). The compute model is kept to be the best to isolate the impact of only the data structure. We show only BFS in (b) because the same trend is observed for other algorithms (update is independent of the running algorithm).

short (heavy)-tailed graphs. Short (heavy)-tailed graphs are graphs with batches containing low (high) *maximum degree*, indicating a short (heavy) tail in the degree distribution of the batch[5]. As shown in Table IV, in contrast to the three other datasets, Wiki and Talk are heavy-tailed graphs with much higher *per-batch maximum degree*, i.e., a heavier tail in each edge batch's degree distribution. Therefore, in contrast to the other datasets, Wiki and Talk have to undergo a much higher maximum per-node edge updates in each batch. AS suffers from coarse-grained locks (the entire vector for a source node is locked) and a lack of intra-node parallelism (Section III-A1). These cause substantial lock contention overhead and update serialization in case of heavy-tailed graphs with a high count of edge updates for the high-degree node. On the other hand, DAH is lockless due to chunked multithreading and offers a fast hash-based update mechanism, which becomes highly beneficial for heavy-tailed graphs. In contrast, for shorter-tailed graphs like LJ, Orkut, and RMAT, DAH becomes lower performing (Fig. 6(b)) because the overhead due to its meta-operations (Section III-A4) overpowers any other benefits. Hence, AS takes over as the higher-performing data structure since the number of edge updates for the high-degree node is low enough to not cause substantial lock contention.

In addition to the above primary observation, we provide insights on the relative strengths of different data structures

[5]Our definition is with respect to a batch because it directly impacts the streaming graph processing latency (Equation 1). However, in our setup, the degree distribution of the entire dataset is generally reflected in a typical batch (batch size = 500K) due to random shuffling of the datasets (Section IV-B). Table IV shows that Wiki and Talk are heavy-tailed across the entire dataset as well as in a typical batch. All our datasets and their corresponding edge batches show power-law degree distribution, and the maximum degree indicates the heaviness of the tail of the degree distribution.

TABLE IV: Max in/out degree for each dataset

Dataset	Entire Dataset		One Batch (Batch size = 500K)	
	Max In-degree	Max Out-degree	Max In-degree	Max Out-degree
LJ	13906	20293	106	147
Orkut	33313	33313	144	144
RMAT	8016	7997	10	10
Wiki	**238040**	3907	**4174**	70
Talk	3311	**100022**	330	**9957**

for both update and compute phases:

Update latency for short-tailed LJ, Orkut, and RMAT. Fig. 6(b) provides evidence of the following relative ordering of the four data structures for update latency (from highest to lowest): DAH > AC > Stinger > AS[6]. Stinger exhibits 1.57×-1.76× higher update latency than AS because it requires two passes to insert edges for a particular node. In addition, each pass involves occasional pointer-chasing, whereas AS contains per-node edge information in a contiguous vector. Compared to AS, AC exhibits 2.2×-2.6× higher latency and DAH exhibits 2.3×-3.2× higher latency. Between AC and DAH, the latter incurs higher latency due to meta-operations such as degree-querying and inter-hash-table flushing during edge update.

Update latency for heavy-tailed Wiki and Talk. Fig. 6(b) shows the following ordering of the four data structures for update latency (from highest to lowest): AS > AC > Stinger > DAH. Averaged over Wiki and Talk, AS shows 12.6×, 3.9×, 2.6× higher update latency compared to DAH, Stinger, and AC, respectively. The benefit of DAH over AS for Wiki and Talk has been discussed above. Stinger and AC perform much better than AS as well. This is because Stinger offers fine-

[6]Although confidence intervals of DAH/AS and AC/AS overlap for Orkut, we report DAH > AC because this relation holds strictly for 2 out of 3 cases.

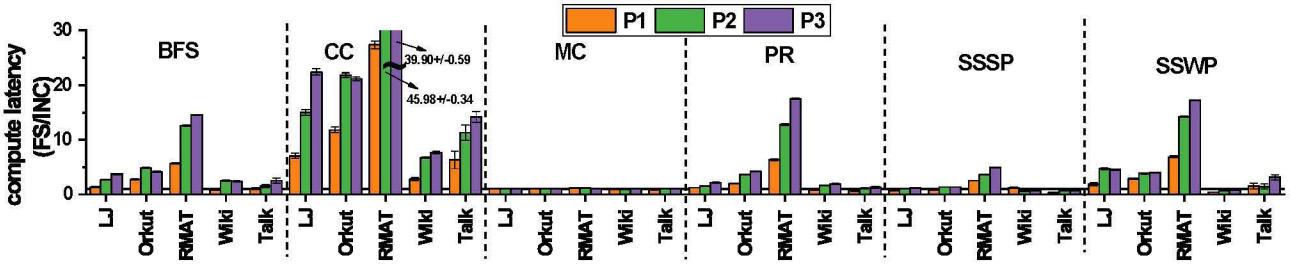


Fig. 7: Compute latency of FS (normalized to INC) at the best data structure over three stages for all the algorithms. Experiments are performed at the best data structure to isolate the performance impact of only the compute model.

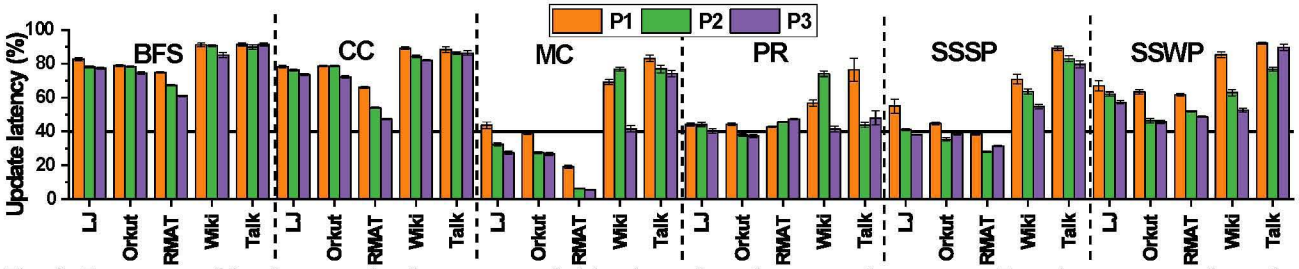


Fig. 8: Percentage of batch processing latency occupied by the update phase over three stages. Experiments are performed at the combination of the best data structure and the best compute model to get the latency breakdown under the best conditions.

grained locks and intra-node parallelism, which can parallelize edge updates for the high-degree node in heavy-tailed graphs. The benefit of AC comes from its chunked and lockless feature. Hence, unlike AS, it does not suffer from lock contention overheads in the case of heavy-tailed graphs. Thus, the choice of multithreading technique is important for the update phase. For adjacency list, heavy-tailed graphs exhibit lower update latency on the lockless chunked-style AC, whereas short-tailed graphs perform better on the shared-style AS.

Impact of data structures and their traversal mechanisms on compute latency. As shown in Fig. 6(c), DAH shows higher compute latency (up to 4.7×) compared to AS in all cases. DAH has an expensive neighbor traversal due to degree-query meta-operations to locate the right hash table for edge retrieval. It performs particularly poorly in PR because we normalize the rank of an incoming neighbor by its out-degree, requiring another degree-query in addition to the one involved in neighbor traversal. AC and Stinger are competitive to AS in multiple cases because all three data structures are based on adjacency list with similar traversal mechanisms. However, in some cases, both show up to 2× higher latency than AS. For example, in Stinger this occurs due to occasional pointer chasing during edge traversal.

C. Impact of Compute Model

Observation: Larger graphs benefit more from the incremental compute model.

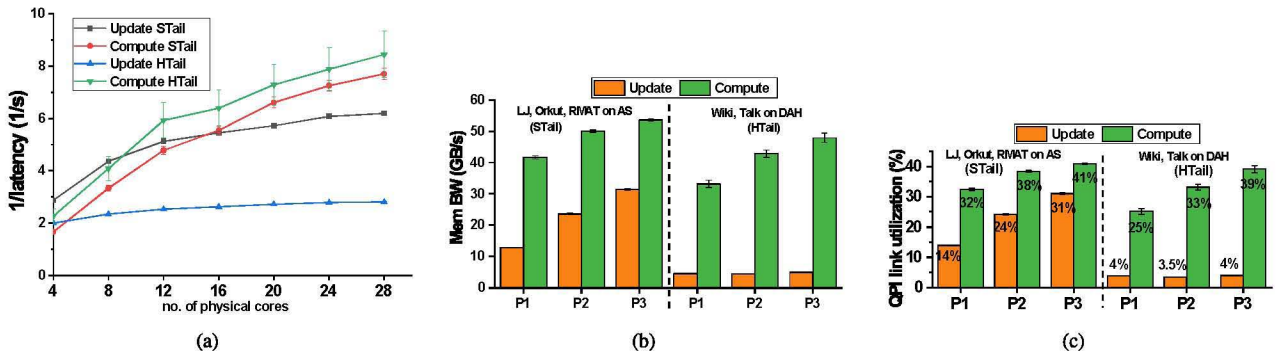
As shown in Fig. 7, for a given algorithm (BFS, CC, PR, SSSP, and SSWP), at any given stage (P1, P2, or P3), RMAT (the largest graph) is the largest beneficiary of INC, whereas Wiki and Talk (the smallest graphs) are the smallest beneficiaries. For RMAT at P3, INC improves compute performance by 15×, 40×, 18×, 5×, and 17× in BFS, CC, PR, SSSP, and SSWP, respectively. In comparison, for Wiki at P3, INC shows only 2.4×, 7.7×, 1.9×, 0.6×, and 0.8× improvements for BFS, CC, PR, SSSP, and SSWP, respectively. In addition, the benefit of INC is higher for later stages P2 and P3 where the graph becomes larger. For example, for BFS on RMAT, INC improves the compute performance by 6×, 13×, and 15× at P1, P2, and P3, respectively. Hence, the incremental compute model offers performance advantages in the compute phase for larger graphs, i.e., a larger dataset at a given stage (RMAT versus Wiki at P3) or the same dataset becoming larger over time (RMAT at P1 versus P2 and P3). For larger graphs, INC saves substantial amount of computations by operating on only a small fraction of the graph[7].

D. Latency Breakdown

Observation: The graph update operation is an important performance limiter in streaming graphs. The update phase contributes at least 40% of the batch processing latency for many workloads.

Fig. 8 shows that the update phase is expensive in many cases such as BFS, CC, and SSWP across all the three stages P1, P2, and P3. For small datasets such as Wiki and Talk, the amount of computation during the compute phase is small and the bottleneck shifts to the update phase. However, the large contribution of the update phase is not limited to only small datasets. Larger datasets LJ, Orkut, and RMAT also show near or more than 40% latency contribution of the update phase in most cases. This provides quantitative evidence that the update phase is as important as the compute phase in the case of streaming graph analytics.

[7]MC is an exception which shows small benefit over INC because FS and INC implementations in MC are similar. In SSSP, FS is competitive to INC (except for the large RMAT dataset) because the delta-stepping FS implementation [36] is highly optimized.



(a) (b) (c)

Fig. 9: (a) STail/HTail update/compute performance scalability to physical core count (cores are distributed equally among 2 sockets at any given core count. Hardware execution threads=2 × number of physical cores), (b) memory bandwidth utilization, and (c) QPI link utilization.

VI. Architecture Level Profiling

We quantitatively study the impact of different architecture resources on the performance of both update and compute phases. Architecture-level characterization is performed with the predominantly best data structure and compute model identified in software-level study (Section V). We use the incremental compute model (INC) for all the algorithms and categorize the results into two groups:

- **STail**: Average across short-tailed graphs LJ, Orkut, and RMAT on AS across six algorithms.
- **HTail**: Average across heavy-tailed graphs Wiki and Talk on DAH across six algorithms.

A. Update Phase vs Compute Phase

Observation and insight: Compared to the compute phase, the update phase exhibits lower utilization of hardware resources, such as higher core counts and memory and inter-socket bandwidths. This trend indicates lower thread-level parallelism (TLP) of the update phase. This observation opens opportunities for inter-phase optimizations in streaming graphs where, unlike in static graph analytics, update and compute phases are interleaved (e.g., the slack in resource utilization in one phase could be leveraged to optimize the other phase). To support our observation, we highlight the following results:

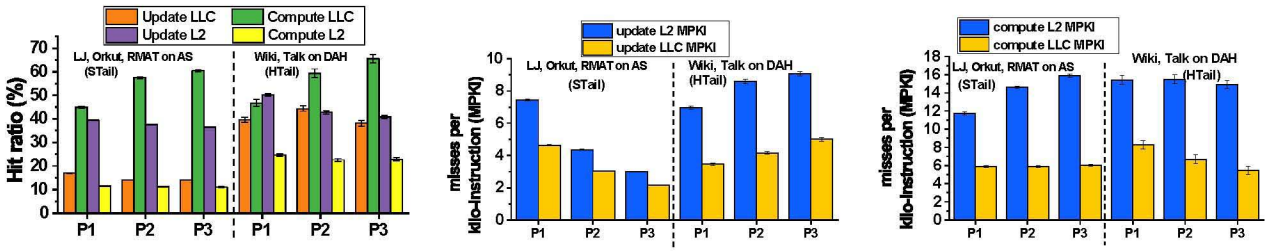
Performance Scalability to Core Counts. In contrast to the compute phase, the update phase shows lower performance scalability to larger core counts. Fig. 9(a) shows that the performance scalability curves of the update phase flatten at earlier core counts than that of the compute phase. At each 4-hop increment in core count (4-8, 8-12, etc.), the update phase undergoes a lower incremental performance improvement than the compute phase. Taking the example of STail, the incremental performance improvement for the update phase is 52% (from 4 to 8 cores) and 17% (from 8 to 12 cores), beyond which the incremental improvement diminishes substantially (6%, 5%, 6%, and 2% for each successive 4-hop increment in core count). In contrast, the STail compute phase shows 100%, 43%, 16%, 19%, 9.7%, and 6% incremental performance improvements for each successive 4-hop increment from 4 to 28 core count.

Memory and Inter-Socket Bandwidth Utilization. The update phase utilizes lower memory and inter-socket bandwidths than the compute phase. As shown in Fig. 9(b), the update memory bandwidth utilizations in STail are 13GB/s, 24GB/s, and 32GB/s at P1, P2, and P3, respectively. In contrast, the corresponding compute phase utilizes 43GB/s, 51GB/s, and 54GB/s, respectively. Fig. 9(c) shows similar difference between the update and compute phases for QPI link utilization. STail update utilizes 14%, 24%, and 31% inter-socket bandwidth at P1, P2, and P3, respectively. In contrast, STail compute utilizes 32%, 38%, and 41% of the available QPI bandwidth at P1, P2, and P3.

These experiments provide evidence that the update phase possesses lower TLP than the compute phase. Even the best data structure for a given category of datasets (AS for LJ, Orkut, RMAT and DAH for Wiki, Talk) suffers from low parallelism in the update phase. Consequently, the update phase is unable to 1) leverage a large number of cores to improve performance and 2) generate a large number of local and remote memory requests to consume sufficiently large memory and inter-socket bandwidths. The next section further elucidates the reasons behind the poor TLP in the update phase.

B. Graph Structure and Update Phase

Observation and insight: The hardware resource utilization of the update phase strongly depends on the underlying structure of the batches of the graph. The update of heavy-tailed graphs on the best data structure (DAH) benefits negligibly from larger core counts, memory bandwidth, and inter-socket bandwidth. In contrast, the update of short-tailed graphs on the corresponding best data structure (AS) shows higher utilization of these architecture resources. This observation, together with the previous one in Section VI-A, supports that the low TLP of the update phase arises from 1) thread contention in short-tailed graphs on AS and 2) workload imbalance in heavy-tailed graphs on DAH. This observation indicates that the parallelism bottleneck of the



(a) (b) (c)

Fig. 10: (a) Private L2 and shared LLC hit ratios. Private L2 and shared LLC MPKI for (b) update and (c) compute phases.

update phase can be addressed with better work distribution technique among threads either to reduce thread contentions or workload imbalance, depending on the specific data structure. Fig. 9(a) shows that, in contrast to STail update, HTail update performance scales worse with large core counts. HTail update shows 17% incremental performance improvement for 4 to 8 cores, beyond which the incremental performance improvement drops below 10%. STail update, on the other hand, shows 52% and 17% incremental performance improvements up to 12 cores. In addition, Fig. 9(b) and 9(c) show that, in contrast to STail update, HTail update exhibits particularly poor utilization of both memory bandwidth (about 5GB/s across P1, P2, and P3) and QPI bandwidth (about 4% across P1, P2, and P3). HTail update on DAH (the best data structure for Wiki and Talk) suffers from low TLP due to workload imbalance, i.e., a large amount of edge updates for a very high-degree node as discussed in SectionV-B. The thread corresponding to the DAH chunk which accommodates the high-degree node is doing most of the work in the update phase. We note that DAH chunks are single-threaded (SectionIII-A4), eliminating the possibility of low TLP due to thread contentions. As to STail update, although it exhibits higher TLP than HTail update, it still shows lower TLP than the compute phase (SectionVI-A). In this case, low TLP arises from the thread contentions in AS where multiple threads share the edge data of the same source node. Workload imbalance is not an extremely serious issue for STail because the datasets are not as heavy-tailed or highly imbalanced as HTail (SectionV-B).

C. On-Chip Caches

Observation and insight: Compared to the update phase, the compute phase exhibits higher L3 cache hit ratios. In contrast, the update phase exhibits higher L2 cache hit ratios than the compute phase. This occurs due to 1) a data reuse relationship and 2) a difference in working set sizes between the two phases.

Fig. 10(a) shows that, for the shared LLC, the compute phase shows a higher hit ratio than the update phase (comparison between "Update LLC" and "Compute LLC"). This is because 1) the compute phase can reuse the edge data freshly brought into LLC by the update phase and 2) the compute phase has a larger working set size because of accesses to vertex property values in addition to the edge data and can therefore leverage the large shared LLC capacity. Moreover, the LLC hit ratio for the compute phase increases over time from P1 to P3 as the graph becomes less sparse and more connected, leading to the possibility of higher reuse.

On the other hand, for the private L2 cache, the update phase shows a higher hit ratio than the compute phase (comparison between "Update L2" and "Compute L2") because of the smaller working set size of the former. The update operation affects only a part of the edge data whose reuse can be captured by the L2 cache. However, the L2 cache provides low benefit for the compute phase because of its large working set size consisting of edge data and vertex property values whose reuse cannot be captured by the L2 cache (such an observation matches prior work for the compute phase in static graph analytics [34]).

Besides aggregate hit ratios, we measure misses per kilo instructions (MPKI) in Fig. 10(b) and 10(c) to further confirm our findings. The L2 cache does a better job at servicing memory requests in the update phase than in the compute phase. This is confirmed by the lower update L2 MPKI (3-9) in Fig. 10(b) compared to the compute L2 MPKI (12-16) in Fig. 10(c). The LLC is effective for the compute phase and is capable of reducing the MPKI from 15 (average) to 6 (average) between the L2 and LLC levels (Fig. 10(c)).

VII. Conclusion

This paper develops SAGA-Bench for streaming graph analytics and characterizes these workloads at the software and the architecture levels. The software-level study shows that 1) the best data structure depends on the per-batch degree distribution of the graph; 2) larger graphs benefit more from the incremental compute model; and 3) the update phase occupies more than 40% of the latency in many cases. The architecture-level study reveals that the update phase shows lower utilization of architecture resources due to lower TLP arising from thread contentions or workload imbalance. Finally, between update and compute phases, the former shows a higher L2 cache hit ratio, whereas the latter benefits more from the LLC.

VIII. Acknowledgments

We thank Shuangchen Li, Eric Hein, Oluwole Jaiyeoba, Kevin Skadron, and the anonymous reviewers for their valuable feedback. This work was supported in part by NSF 1816833, 1533933, 1719160, 1730309, and CRISP, one of the six centers in JUMP, a SRC program sponsored by DARPA.

REFERENCES

[1] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 85–98, ACM, 2012.

[2] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: a graph engine for temporal graph analysis," in *Proceedings of the Ninth European Conference on Computer Systems*, p. 1, ACM, 2014.

[3] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455, ACM, 2013.

[4] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1876–1888, 2018.

[5] D. Eswaran, C. Faloutsos, S. Guha, and N. Mishra, "Spotlight: Detecting anomalies in streaming graphs," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1378–1386, ACM, 2018.

[6] C. Eksombatchai, P. Jindal, J. Z. Liu, Y. Liu, R. Sharma, C. Sugnet, M. Ulrich, and J. Leskovec, "Pixie: A system for recommending 3+ billion items to 200+ million users in real-time," in *Proceedings of the 2018 World Wide Web Conference*, WWW '18, (Republic and Canton of Geneva, Switzerland), pp. 1775–1784, International World Wide Web Conferences Steering Committee, 2018.

[7] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin, "Graphjet: real-time content recommendations at twitter," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1281–1292, 2016.

[8] A. Grewal, J. Jiang, G. Lam, T. Jung, L. Vuddemarri, Q. Li, A. Landge, and J. Lin, "Recservice: Distributed real-time graph processing at twitter," in *Proceedings of the 10th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'18, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2018.

[9] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*, pp. 1–5, IEEE, 2012.

[10] K. Iwabuchi, S. Sallinen, R. Pearce, B. Van Essen, M. Gokhale, and S. Matsuoka, "Towards a distributed large-scale dynamic graph data store," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 892–901, IEEE, 2016.

[11] F. Sheng, Q. Cao, H. Cai, J. Yao, and C. Xie, "Grapu: Accelerate streaming graph analysis through preprocessing buffered updates," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, (New York, NY, USA), pp. 301–312, ACM, 2018.

[12] K. Vora, R. Gupta, and G. Xu, "Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 237–251, 2017.

[13] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, (New York, NY, USA), pp. 918–934, ACM, 2019.

[14] W. Jaiyeoba and K. Skadron, "Graphtinker: A high performance data structure for dynamic graph processing," in *2019 IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 2019.

[15] Z. Cai, D. Logothetis, and G. Siganos, "Facilitating real-time graph mining," in *Proceedings of the fourth international workshop on Cloud data management*, pp. 1–8, ACM, 2012.

[16] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," in *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pp. 249–263, 2019.

[17] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *2015 IEEE International Symposium on Workload Characterization*, pp. 56–65, IEEE, 2015.

[18] S. Eyerman, W. Heirman, K. D. Bois, J. B. Fryman, and I. Hur, "Many-core graph workload analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, (Piscataway, NJ, USA), pp. 22:1–22:11, IEEE Press, 2018.

[19] 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 *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 49–58, IEEE, 2016.

[20] L. Jiang, L. Chen, and J. Qiu, "Performance characterization of multi-threaded graph processing applications on many-integrated-core architecture," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 199–208, IEEE, 2018.

[21] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[22] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–14, IEEE, 2018.

[23] A. Mukkara, N. Beckmann, and D. Sanchez, "Phi: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates," in *2019 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52),*, IEEE, 2019.

[24] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *2019 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52),*, IEEE, 2019.

[25] M. Y. Yan, X. Hu, S. C. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. J. Feng, P. Gu, L. Deng, X. C. Ye, Z. M. Zhang, D. R. Fan, and Y. Xie, "Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach," in *2019 52th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, Oct. 2019.

[26] 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 *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 544–557, Feb 2018.

[27] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *2017 IEEE International symposium on high performance computer architecture (HPCA)*, pp. 457–468, IEEE, 2017.

[28] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 105–117, 2016.

[29] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 166–177, IEEE, 2016.

[30] J. Zhou, S. Liu, Q. Guo, X. Zhou, T. Zhi, D. Liu, C. Wang, X. Zhou, Y. Chen, and T. Chen, "Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 731–734, IEEE, 2017.

[31] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, (New York, NY, USA), pp. 39:1–39:11, ACM, 2016.

[32] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 531–543, IEEE, 2018.

[33] S. G. Singapura, A. Srivastava, R. Kannan, and V. K. Prasanna, "Oscar: Optimizing scratchpad reuse for graph processing," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, 2017.

[34] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 373–386, Feb 2019.

[35] G. Dai, T. Huang, Y. Wang, H. Yang, and J. Wawrzynek, "Hyve: Hybrid vertex-edge memory hierarchy for energy-efficient graph processing," *IEEE Transactions on Computers*, vol. 68, pp. 1131–1146, Aug 2019.

[36] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[37] F. Busato, O. Green, N. Bombieri, and D. A. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–7, Sep. 2018.

[38] G. Feng, X. Meng, and K. Ammar, "Distinger: A distributed graph data structure for massive dynamic graph processing," in *2015 IEEE International Conference on Big Data (Big Data)*, pp. 1814–1822, IEEE.

[39] O. Green and D. A. Bader, "custinger: Supporting dynamic graph algorithms for gpus," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2016.

[40] D. Sengupta and S. L. Song, "Evograph: On-the-fly efficient mining of evolving graphs on gpu," in *International Supercomputing Conference*, pp. 97–119, Springer, 2017.

[41] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan, "Graphin: An online high performance incremental graph processing framework," in *European Conference on Parallel Processing*, pp. 319–333, Springer, 2016.

[42] M. Sha, Y. Li, B. He, and K.-L. Tan, "Accelerating dynamic graph analytics on gpus," *Proceedings of the VLDB Endowment*, vol. 11, no. 1, pp. 107–120, 2017.

[43] X. Shi, B. Cui, Y. Shao, and Y. Tong, "Tornado: A system for real-time iterative analysis over evolving data," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 417–430, ACM, 2016.

[44] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *Proceedings of the 31st IEEE International Conference on Data Engineering*, IEEE, 2015.

[45] X. Ju, D. Williams, and H. Jamjoom, "Version traveler: Fast and memory-efficient version switching in graph processing systems.," in *USENIX Annual Technical Conference.*, 2016.

[46] M. Then, T. Kersten, S. Günnemann, A. Kemper, and T. Neumann, "Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs," *Proceedings of the VLDB Endowment*, vol. 10, no. 8, pp. 877–888, 2017.

[47] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, p. 5, ACM, 2016.

[48] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM Sigplan Notices*, vol. 48, pp. 135–146, ACM, 2013.

[49] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pp. 17–30, 2012.

[50] 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," *Proc. VLDB Endow.*, vol. 8, pp. 1214–1225, July 2015.

[51] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[52] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, (Broomfield, CO), pp. 599–613, USENIX Association, Oct. 2014.

[53] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 472–488, ACM, 2013.

[54] Y. Gao, W. Zhou, J. Han, D. Meng, Z. Zhang, and Z. Xu, "An evaluation and analysis of graph processing frameworks on five key issues," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, (New York, NY, USA), pp. 11:1–11:8, ACM, 2015.

[55] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pp. 631–643, 2017.

[56] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 979–990, ACM, 2014.

[57] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, "How well do graph-processing platforms perform? an empirical performance evaluation and analysis," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 395–404, IEEE, 2014.

[58] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, (New York, NY, USA), pp. 1813–1828, ACM, 2016.

[59] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: understanding graph computing in the context of industrial solutions," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, IEEE, 2015.

[60] K. Vora, R. Gupta, and G. Xu, "Synergistic analysis of evolving graphs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, p. 32, 2016.

[61] I. Corporation, "Intel processor counter monitor." https://github.com/opcm/pcm, 2017.

[62] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.

[63] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, pp. 442–446, SIAM, 2004.