University of California
Santa Barbara

Benchmarking, Performance Analysis, and Domain-Specific Architectures for Graph Processing Applications

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Abanti Basak

Committee in charge:

Professor Yuan Xie, Co-Chair
Professor Yufei Ding, Co-Chair
Professor Chandra Krintz
Professor Behrooz Parhami

June 2021

The Dissertation of Abanti Basak is approved.

Professor Chandra Krintz

Professor Behrooz Parhami

Professor Yufei Ding, Committee Co-Chair

Professor Yuan Xie, Committee Co-Chair

May 2021

Benchmarking, Performance Analysis, and Domain-Specific Architectures for Graph Processing Applications

by

Abanti Basak

Acknowledgements

I am grateful for the support I received during my Ph.D. journey. I thank my advisors Professor Yuan Xie and Professor Yufei Ding for their precious guidance. In addition, I have had the good fortune of working with great industry mentors: Dr. Alaa Alameldeen, Dr. Zeshan Chishti, and Dr. Wei Wu during my internships at Intel Labs; and Dr. Li Zhao during my internship at Alibaba Group. They have all helped me grow as a researcher. I am also grateful for the immense help I received from my labmates and the postdoctoral researchers at UCSB SEAL Lab, especially Dr. Shuangchen Li and Dr. Xing Hu for guiding me through my first paper. Most importantly, I thank my parents, my sister Aroni, and my husband Subhro for being constant sources of support, encouragement, and inspiration during my time as a Ph.D. student.

Curriculum Vitæ

Abanti Basak

Education

2021	Ph.D. in Electrical and Computer Engineering (Expected), University of California, Santa Barbara.
2016	B.S.E. in Electrical Engineering, Princeton University.

Publications

[C1]. **Abanti Basak**, Zheng Qu, Jilan Lin, Alaa Alameldeen, Zeshan Chishti, Yufei Ding, Yuan Xie. "Improving Streaming Graph Processing Performance using Input Knowledge." *Under review*, 2021.

[C2]. Jilan Lin, **Abanti Basak**, Shuangchen Li, Dimin Niu, Hongzhong Zheng, Yufei Ding, Yuan Xie. "KiloGraph: Towards A High-Flexible Graph Processing Architecture." *Under review*, 2021.

[C3]. Xinfeng Xie, Zheng Liang, Peng Gu, **Abanti Basak**, Lei Deng, Ling Liang, Xing Hu, Yuan Xie. "SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator." *27th IEEE International Symposium on High-Performance Computer Architecture (HPCA-27)*, 2021.

[C4]. **Abanti Basak**, Jilan Lin, Ryan Lorica, Xinfeng Xie, Zeshan Chishti, Alaa Alameldeen, Yuan Xie. "SAGA-Bench: Software and Hardware Characterization of Streaming Graph Analytics Workloads." *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[C5]. Mingyu Yan, Xing Hu, Shuangchen Li, **Abanti Basak**, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, Yuan Xie. "Alleviating Irregularity in Graph Analytics Acceleration: a Hardware/Software Co-Design Approach." *52nd IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019.

[C6]. **Abanti Basak**, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Xiaowei Jiang, Li Zhao, and Yuan Xie. "Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads." *25th IEEE International Symposium on High-Performance Computer Architecture (HPCA-25)*, 2019.

[C7]. **Abanti Basak**, Xing Hu, Shuangchen Li, Sang Min Oh, and Yuan Xie. "Exploring Core and Cache Hierarchy Bottlenecks in Graph Processing Workloads." *IEEE Computer Architecture Letters (CAL)*, 2018.

[C8]. Xing Hu, Matheus Ogleari, Jishen Zhao, Shuangchen Li, **Abanti Basak**, and Yuan Xie. "Persistence Parallelism Optimization: A Holistic Approach from Memory Bus to RDMA Network." *51st IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*, 2018.

Abstract

Benchmarking, Performance Analysis, and Domain-Specific Architectures for Graph Processing Applications

by

Abanti Basak

Both static and streaming graph processing are central in data analytics scenarios such as recommendation systems, financial fraud detection, and social network analysis. The rich space of graph applications poses several challenges for the computer architecture community. First, standard static graph algorithm performance is sub-optimal on today's general-purpose architectures such as CPUs due to inefficiencies in the memory subsystem. It is currently increasingly difficult to rely on relative compute/memory technology scaling for continued performance improvement for a given optimized static graph algorithm on a general-purpose CPU. Second, while a large body of research in the computer architecture community focuses on static graph workloads, streaming graphs remain completely unexplored. The primary practical barriers for computer architecture researchers toward studying streaming graphs are immature software, a lack of systematic software analysis, and an absence of open-source benchmarks. This dissertation seeks to solve these challenges for both static and streaming graph workloads through benchmarking, performance analysis, and CPU-centric domain-specific architectures using software/hardware co-design.

For static graph workloads, this thesis highlights novel performance bottleneck insights such as 1) the factors limiting memory-level parallelism, 2) the heterogeneous reuse distances of different application data types, and 3) the difference in the performance sensitivities of the different levels of the cache hierarchy. Guided by the workload

characterization, a domain-specific prefetcher called DROPLET is proposed to solve the memory access bottleneck. DROPLET is a physically decoupled but functionally cooperative prefetcher co-located at the L2 cache and at the memory controller. Moreover, DROPLET is data-aware because it prefetches different graph data types differently according to their intrinsic reuse distances. DROPLET achieves 19%-102% performance improvement over a no-prefetch baseline and 14%-74% performance improvement over a Variable Length Delta Prefetcher (VLDP). DROPLET also performs 4%-12.5% better than a monolithic L1 prefetcher similar to the state-of-the-art prefetcher for graphs.

For streaming graph workloads, this thesis develops a performance analysis framework called SAGA-Bench and performs workload characterization at both the software and the architecture levels. The findings include 1) the performance limitation of the graph update phase, 2) the input-dependent software performance trade-offs in graph updates, and 3) the difference in architecture resource utilization (core counts, memory bandwidth, and cache hierarchy) between the graph update and the graph compute phases. In addition, the thesis proposes the SPRING approach to demonstrate that input knowledge-driven software and hardware co-design is critical to optimize the performance of streaming graph processing. Evaluated across 260 workloads, our input-aware techniques provide on average 4.55× and 2.6× improvement in graph update performance for different input types. The graph compute performance is improved by 1.26× (up to 2.7×).

Contents

Chapter 1

Introduction

1.1 Motivation and Contributions

Due to the explosion of data in today's world, both static and streaming graph processing are widely used to solve big data problems in multiple domains such as social networks, web searches, recommender systems, fraud detection, financial money flows, and transportation. Static graph processing consists of performing analytics on statically known whole input graphs, whereas streaming graph processing handles time-evolving graphs. The high potential of graph processing is due to its rich, expressive, and widely applicable data representation consisting of a set of entities (vertices) connected to each other by relational links (edges). Numerous vendors such as Oracle [1], Amazon (AWS) [2], and Microsoft [3] provide graph processing engines for enterprises. Moreover, companies such as Google [4], Facebook [5], and Twitter [6, 7] have built customized graph processing frameworks to drive their products. Graph technology is also predicted to be the driver for many emerging data-driven markets, such as an expected $7 trillion worth market of self-driving cars by 2050 [8].

However, the rich space of big-data graph applications poses multiple challenges for

the computer architecture community. First, standard static graph algorithm performance is sub-optimal on today's general-purpose architectures such as CPUs because of critical mismatches between the trends in the application and the computer architecture landscapes. As input graphs become larger and exceed the on-chip cache sizes, static graph algorithms become memory-bound. In contrast, memory technology does not scale as fast as compute technology. Consequently, the performance of single-machine in-memory static graph analytics is bounded by the inefficiencies in the memory subsystem, making the cores stall as they wait for data to be fetched from the DRAM. As shown in Fig. 1.1 for one of the benchmarks used in our evaluation, 45% of the cycles are DRAM-bound stall cycles, whereas the core is fully utilized without stalling in only 15% of the cycles. It is currently increasingly difficult to rely on relative compute/memory technology scaling for continued performance improvement for a given optimized static graph algorithm on a general-purpose CPU. Therefore, it is an opportune time to specifically study these application characteristics to find the performance bottlenecks and address them using domain-specific computer architecture. As a result, recent work in computer architecture has focused on performance analysis [9–13] and CPU-integrated domain specialization [14, 15] for static graph workloads. However, prior performance analysis [9–13] is not explicitly aware of the application data types. In addition, prior workload characterization is real hardware based and therefore misses the opportunity to conduct a detailed sensitivity analysis of different architectural parameters such as the instruction window size and the cache parameters. With the insights from these missing studies, it is possible to substantially improve the performance of domain-specific solutions for static graphs.

Second, while a large body of research in the computer architecture community focuses on static graph workloads, streaming graphs remain completely unexplored. Domain knowledge, contributed by some influential workload characterization and benchmark-

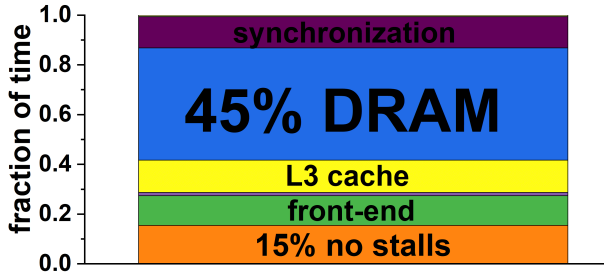


Figure 1.1: Cycle stack of PageRank on static orkut dataset [16]

ing efforts [9, 11, 17], has been key to driving research on domain-specific architectures for static graph processing. However, the domain knowledge is insufficient in its existing state because it ignores the time-evolving nature of graphs. Instead, the focus is restricted to static graphs which never change while algorithms are run. In reality, however, most graphs are fast-changing in today's big data era where data evolves rapidly. This realistic scenario is captured by streaming graph processing, i.e., performing batched updates and analytics on time-evolving graphs. Streaming graphs are critical in graph convolutional networks [18], social network analysis [19], real-time financial fraud detection [20], anomaly detection [21], and recommendation systems [22]. By neglecting the dynamic nature of graphs, the research community is missing the opportunity to leverage a broader domain knowledge to design more adequate architecture solutions for practical and realistic graph processing. We recognize two reasons for this negligence:

- *Immature software and lack of systematic software performance analysis*: The data structures and compute models underpinning streaming graph systems are still actively being researched and have not been studied *systematically*. A lack of systematicness arises from the heterogeneity of the proposed systems. 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. Consequently, it is challenging for computer architects to rely on a standard set of underlying software components. This is in significant contrast to static graph processing where the core data structures and computation models are better standardized.

- *Absence of open-source benchmarks*: The lack of an open-source benchmark makes it challenging to study streaming graph workloads at the architecture level. Existing open-source implementations are *holistic systems*, each with a specialized complex package of system-specific optimizations. A more useful resource for architecture exploration is an open-source benchmark with the essential software techniques (data structures and compute models) to understand the core characteristics of the workloads without system-specific optimizations. However, such a benchmark for streaming graphs is currently missing in the architecture community. This is in significant contrast to static graph processing where workload characterization and performance analysis are facilitated by system-independent reference implementations targeted at architecture research [11, 17].

This thesis seeks to solve the above challenges for both static and streaming graph workloads. For each category of graph workloads, the contributions of the dissertation span two areas: 1) benchmarking and performance analysis, and 2) CPU-coupled domain-specific architectures. The contributions are described below and summarized in Table 1.1.

Contributions in benchmarking and performance analysis (Chapters 3 and 5): For static graph workloads, this thesis performs a data-aware performance analysis of the GAP benchmark suite [17], focusing on the memory-level parallelism and the cache

hierarchy. We extend or fill the gaps in prior characterization work [9–13] in two aspects. First, we perform a data-aware profiling which provides clearer guidelines on the management of specific data types for performance optimization. Second, with the flexibility of a simulated platform, we vary the instruction window and cache configuration design parameters to explicitly explore their performance sensitivity. Beyond prior performance analysis work, our key findings include (Chapter 3):

- Load-load dependency chains that involve specific application data types, rather than the instruction window size limitation, make up the key bottleneck in achieving a high memory-level parallelism.
- Different graph data types exhibit heterogeneous reuse distances. The architectural consequences are (1) the private L2 cache shows negligible impact on improving system performance, (2) the shared L3 cache shows higher performance sensitivity, and (3) the graph property data type benefits the most from a larger shared L3 cache.

For streaming graph workloads, this thesis highlights that existing graph-targeted architectures rely on the restricted domain knowledge of static graphs and ignore the critical dynamic nature of graphs. We therefore broaden the domain knowledge for the computer architecture community through the following contributions (Chapter 5):

- An open-source performance analysis platform called SAGA-Bench: SAGA-Bench is targeted at software and hardware studies of the essential data structures and compute models proposed across various existing streaming graph systems. For software performance analysis, we enable systematicness by using comparable implementations of the core software components (without system-specific optimizations) and identical measurement methodology, thus alleviating the problem of difficult-to-interpret comparisons across heterogeneous stand-alone systems. For hardware studies, SAGA-Bench provides a benchmark to study the architecture bottlenecks for these workloads.

- Software-level workload characterization: We further use SAGA-Bench to systematically analyze the software performance, which leads to three key findings. First, the graph update operation is an important performance limiter contributing at least 40% of the batch processing latency for many workloads. Second, the best (i.e., lowest batch processing latency) data structure for a streaming graph depends on the degree distribution of the input batches of the graph. Third, the performance of different compute models depends on the input graph size. The incremental compute model offers performance benefits especially for larger graphs.

- Architecture-level workload characterization: We perform a comparative study between the graph update and the graph compute phases. First, we find that, compared to the compute phase, the update phase exhibits a lower utilization of hardware resources, such as higher core counts and memory and inter-socket bandwidths. We further provide insights to explain the resource utilization of the graph update phase in terms of the underlying structure/topology of the input graph batches. Finally, we observe that the L2 cache services more memory requests in the update phase than in the compute phase, whereas the LLC is effective for the compute phase.

Contributions in CPU-coupled domain-specific architectures (Chapters 4 and 6): To optimize the performance of static and streaming graph applications, we design lightweight domain-specific architectures using software/hardware co-design tightly integrated with commodity CPUs. These lightweight solutions involve lower design costs than fully customized accelerator chips. In addition, low-overhead tight integration with commodity processors makes these solutions more amenable to widespread adoption.

For static graph workloads, we use the guidelines from our characterization to design DROPLET, a Data-awaRe decOuPLed prEfeTcher for graphs (Chapter 4). DROPLET is a physically decoupled but functionally cooperative prefetcher co-located at the L2 cache

and at the memory controller. We adopt a decoupled design to overcome the serialization due to the dependency between different graph data types. Moreover, DROPLET is data-aware because it prefetches different graph data types differently according to their intrinsic reuse distances. DROPLET achieves 19%-102% performance improvement over a no-prefetch baseline, 9%-74% performance improvement over a conventional stream prefetcher, 14%-74% performance improvement over a Variable Length Delta Prefetcher (VLDP) [23], and 19%-115% performance improvement over a delta correlation prefetcher implemented as a global history buffer (GHB) [24]. DROPLET performs 4%-12.5% better than a monolithic L1 prefetcher similar to the state-of-the-art prefetcher for graphs [15].

For streaming graph workloads, our proposed SPRING approach demonstrates that input knowledge-driven software and hardware co-design is critical to optimize the performance (Chapter 6). To improve graph update efficiency, we first characterize the performance trade-offs of an input-oblivious software technique called batch reordering [25,26]. Guided by our findings, we propose input-aware batch reordering to adaptively reorder input batches based on their degree distributions. To complement adaptive batch reordering, we propose updating graphs dynamically, based on their input characteristics, either in software (via update search coalescing) or in hardware (via acceleration support). To improve graph computation efficiency, we present input-aware work aggregation which adaptively modulates the computation granularity based on inter-batch locality characteristics. Evaluated across 260 workloads, our input-aware techniques provide on average 4.55× and 2.6× improvement in graph update performance for different input types (on top of eliminating the performance degradation from input-oblivious batch reordering). The graph compute performance is improved by 1.26× (up to 2.7×).

Table 1.1 summarizes the contributions of this thesis described in detail in the previous paragraphs. The rest of the dissertation is organized as follows. In Chapter 2, we provide 1) the necessary background on our target application areas (static and streaming

Table 1.1: Summary of contributions

	Static Graph Workloads	**Streaming Graph Workloads**
Benchmarking and Performance Analysis	Performance analysis of GAP benchmark suite [17] (Chapter 3)	SAGA-Bench and associated workload characterization (Chapter 5)
CPU-coupled domain-specific architectures	DROPLET (Chapter 4)	SPRING (Chapter 6)

graphs) and 2) explain the choice of the CPU platform for graph processing applications. As shown in Table 1.1, Chapters 3 and 4 analyze and optimize static graph processing. In Chapter 3, we perform a data-aware performance analysis of the GAP benchmark suite [17], focusing on the memory-level parallelism and the cache hierarchy. Based on these profiling observations, we propose, in Chapter 4, our domain-specific prefetcher called DROPLET to solve the memory access bottleneck. As shown in Table 1.1, the second part of the dissertation focuses on streaming graph applications, which is a more general form of graph processing involving the time-evolving nature of graphs. In Chapter 5, we develop a performance analysis framework called SAGA-Bench and perform workload characterization at both the software and the architecture levels. Guided by the performance analysis, we propose, in Chapter 6, our SPRING approach, i.e., input-dependent software/hardware co-design to improve the performance of streaming graph systems. Finally, we summarize our contributions and future research directions in Chapter 7.

1.2 Future Influence and Impact

The insights and solutions developed by this thesis have significant potential for long-term impact. In addition to being a high-performance domain-specific prefetching solution for static graphs, DROPLET demonstrates working design principles that could influence future prefetcher designs. First, DROPLET provides evidence for the benefits

of software-assisted hardware prefetching (i.e., with data type hints from the software, a hardware prefetcher is capable of achieving high accuracy and performance). DROPLET design also provides a reference methodology for designing prefetchers for irregular or sparse applications. Finally, DROPLET demonstrates the benefits of physically decoupling a prefetcher across the memory hierarchy according to the application needs.

Our contributions in streaming graph workloads help broaden the conventional and restrictive domain knowledge of static graphs underpinning today's proposed domain-specific architectures. Re-thinking existing designs to support streaming graphs is a non-trivial task. The performance metric is batch processing latency which is different from whole-graph computation performance optimized by existing designs. Streaming graphs involve a different set of underlying data structures and compute models with their unique hardware implications. The primary practical barriers for computer architecture researchers are immature software, a lack of systematic software analysis, and an absence of open-source benchmarks. Consequently, research has naturally inclined to static graphs where the software is more mature, and open-source benchmarks are readily available for researchers to get started with. Our work (SAGA-Bench and SPRING) alleviates the barriers to studying streaming graphs and opens the door to the development of more practical, general, and realistic domain-specific architectures aware of the dynamic nature of graphs. Moreover, SAGA-Bench by itself (as a tool) has the potential to be of significant practical value because, to the best of our knowledge, it is the first 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. Equipped with the flexibility to support both future streaming graph codes and traditional static graph analytics, SAGA-Bench provides a one-stop shop for architecture researchers to perform systematic studies of a large spectrum of graph workloads. Our framework improves research productivity during the stage of novel workload discovery

and characterization. A common platform 1) relieves the difficulty of navigating through heterogeneous stand-alone systems for software performance comparison, and 2) provides a means to easily implement the core software of the workload to quickly characterize it on the hardware to understand the architecture bottlenecks.

Chapter 2

Background

In this chapter, we first discuss the key characteristics of static and streaming graph workloads. Next, we provide some background on different hardware platforms for graph processing applications to understand why CPU is the platform of choice in this dissertation.

2.1 Static and Streaming Graph Workloads

Static graph processing constitutes performing analytics on statically known input graphs, whereas streaming graph processing handles time-evolving graphs. There are significant differences between the underlying data structures, the execution flow, and the optimization goal between the two categories of graph workloads.

Data Structures: One of the most widely used graph data structures for static graph analytics is the Compressed Sparse Row (CSR) representation because of its efficient memory space usage. As shown in Fig. 2.1, the CSR format consists of three main components: the offset pointers, the neighbor IDs, and the vertex data. Each entry in the offset pointer array belongs to a unique vertex V and points to the start of the list

of V's neighbors in the neighbor ID array. In the case of weighted graphs, each entry in the neighbor ID array also includes the weight of the corresponding edge. The vertex data array stores the property of each vertex and is indexed by the vertex ID (the vertex data size is fixed). In the rest of the dissertation, we use the following terminology: 1) **Structure data** (the neighbor ID array), 2) **Property data** (the vertex data array), and 3) **Intermediate data** (any other data). Property data is indirectly indexed using information from the structure data. Using the example in Fig. 2.1, to find the property of the neighbors of vertex 6, the structure data is first accessed to obtain the neighbor IDs (59 and 78), which in turn are used to index the property data.

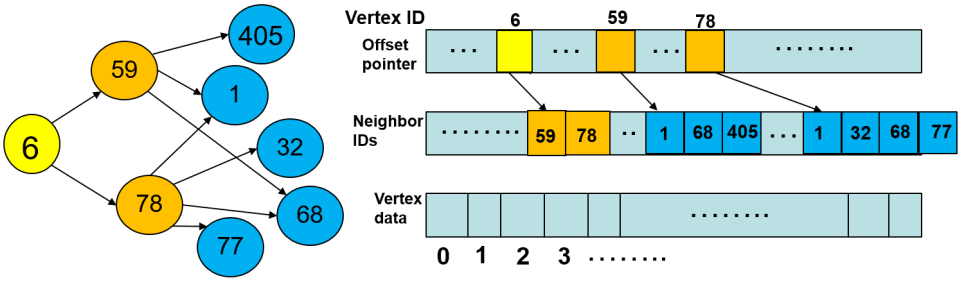


Figure 2.1: CSR data layout for graphs

While CSR is a standard data structure for static graphs, it is not an appropriate choice for streaming graphs. This is because the latter involves updating the graph topology (therefore changing the data structure during the runtime) and compact arrays like CSR is not efficient for insertion/deletion. Data structures for streaming graphs are still being actively researched. Chapter 5 describes multiple data structure propositions for streaming graphs and implements them in our proposed SAGA-Bench performance analysis framework for a systematic performance analysis.

Execution Flow and Optimization Goal: Fig. 2.2 shows the difference in the execution flow between static and streaming graph analytics. In the former, an entire input file is read to build a graph usually in the CSR format. 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. 2.3 further clarifies update and compute operations. 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: 1) ***Update phase***, i.e., the incoming edges in a given batch are ingested into the graph data structure (the updates may involve addition/deletion of edges, weight changes, and addition/deletion of nodes); 2) ***Compute phase***, i.e., an algorithm such as PageRank is performed on the freshly updated data structure. Following previous work, the batch sizes considered in this dissertation range from 100 to 500K (Chapters 5 and 6) . In our evaluation (Chapters 5 and 6), the number of input batches is the ratio of the total number of edges in the graph and the batch size. When an input dataset possesses timestamp information, the input file specifies the order in which the edges appear in the graph and batches are formed accordingly. When timestamps are not available, the datasets are randomly shuffled to break any ordering in the input files (they are often ordered in increasing source vertex ID, which is not the likely scenario of edge appearance for real-world streaming graphs). The shuffled input file is then read in batches of the given batch size.

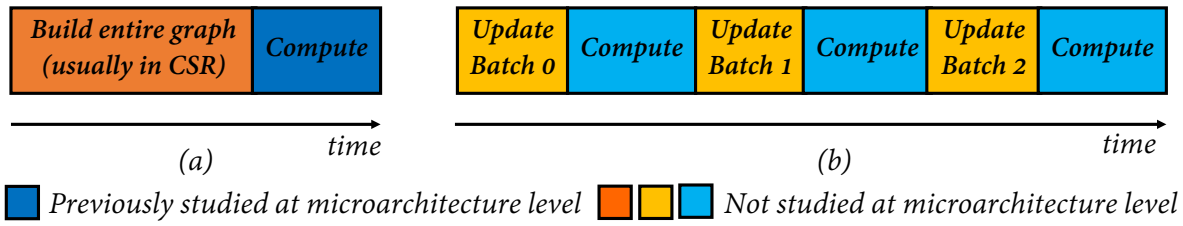


Figure 2.2: Execution flow of (a) static and (b) streaming graphs

[1]The current version of SAGA-Bench 1) maintains the latest snapshot of an evolving graph similar to [19, 27, 28] and 2) supports the model where update and compute are interleaved (Fig. 2.2b) similar to [27, 29–38]. A few existing systems maintain multiple over-time snapshots [39–43]. Two very recently proposed systems [25, 26] 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.

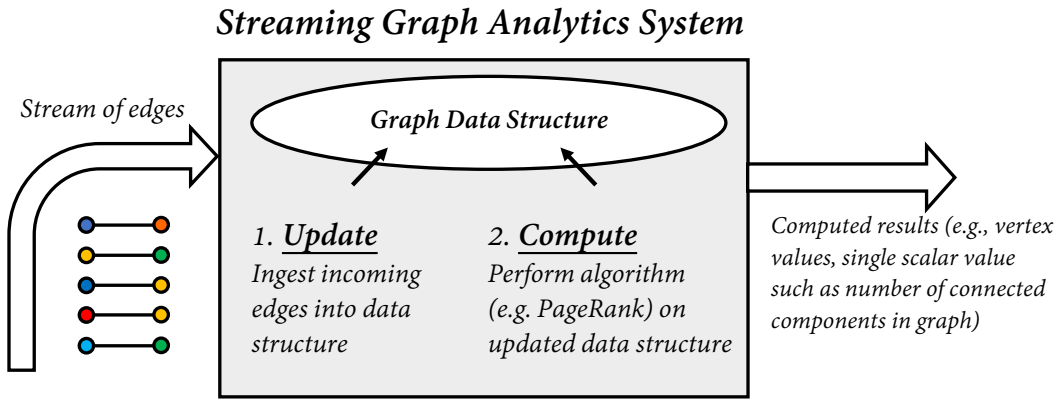


Figure 2.3: Overview of streaming graph analytics

Due to the difference in the execution flow, the optimization targets are different between static and streaming graph processing. In static graphs, the optimization target is the execution time of the compute phase (Fig. 2.2(a)). The graph building phase is considered to be a fixed one-time overhead that can be amortized by performing repeated computations. On the other hand, 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. *Batch processing latency* (Equation 2.1) is the performance metric for streaming graphs. Hence, the *graph update phase lies on the critical path for streaming graphs and cannot be considered as a one-time overhead.*

$$\begin{aligned} &\text{batch processing latency}_{batch\ i} = \\ &\text{update latency}_{batch\ i} + \text{compute latency}_{batch\ i} \end{aligned} \tag{2.1}$$

The critical path characteristic of the graph update phase hinders smooth portability of static graphs' software-hardware solutions to streaming graphs. First, it is inefficient to borrow software solutions from static graph analytics. Borrowing array-based CSR and pre-processing techniques [44] beneficial for the compute phase would substantially hurt the update latency. Similarly, borrowing conventional algorithms [11, 17] would lead to

redundant computations because two successive compute phases may have large overlap in vertices and edges. Second, it is inefficient to borrow 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.

2.2 Choice of CPU Platform for Graph Processing Applications

This dissertation focuses on single-machine large-memory CPU server for performance analysis and domain specialization of graph processing applications. CPUs are predominant in datacenters [45, 46] and supercomputers [47] because they are easily accessible, cost-effective, and highly programmable. Hence, CPUs are still the subject of continuous innovation and improvement. Today, there are numerous efforts to shift towards new types of server-class CPUs equipped with more application-specific customization [48] or different ISAs (e.g., shift from x86 to ARM-based server CPUs [47,49–51]) that are better suited to application needs (e.g., cloud-native applications). This trend in the computer architecture industry provides evidence that CPUs are powerful and it is worthwile and valuable to develop high-performance and energy-efficient CPUs with application-specific customization. Beyond the critical importance of CPUs in the industry, this dissertation

chooses the CPU platform for graph processing because CPU possesses significant advantages compared to alternative hardware platforms such as distributed systems [4,5,52–54], out-of-core systems [55–59], customized accelerators [60–64], and GPUs. First, as opposed to distributed systems, a scale-up big-memory CPU does not need to consider the challenging task of graph partitioning among nodes and avoids network communication overhead. Programming frameworks developed for this kind of single-machine CPU platform in both academia [65–67] and industry [6, 7, 68] have shown excellent performance while significantly reducing the programming efforts compared to distributed systems. Single CPU system is also possible because many common case industry and academic graphs have recently been reported to fit comfortably in the RAM of a single high-end CPU server [65, 69, 70], given the availability and affordability of high memory density (for example, a quad-socket Intel Xeon machine with 1.5TB RAM costs about $35K as of 2017 [71]). Second, both out-of-core systems and customized accelerators require expensive pre-processing to prepare partitioned data structures to improve locality. Single-machine in-memory CPU graph analytics can avoid costly pre-processing which has been shown to often consume more time than the algorithm execution itself [72, 73]. Third, GPU is a challenging choice for graph applications due to the difficulty of programming graph algorithms on GPUs and due to the limited memory capacity of GPUs. During the early years of this dissertation, performance analysis work [74, 75] provided evidence that, for graphs that fit in the limited GPU memory (12GB), CPU-only programming frameworks perform comparably or better than GPU-only programming frameworks. For example, [74] shows that, compared to CPU-based Ligra [76], Gunrock's performance is comparable for most tested primitives. Moreover, compared to CPU-based Galois [77,78], Gunrock shows speedup on traversal-based graph primitives (Breadth First Search, Single Source Shortest Paths, and Betwenness Centrality) and less performance advantage on Pagerank and Connected Components. Only recently has GPU programming become

more competitive than CPU-oriented graph algorithms. SIMD-X [79], a GPU-only programming framework developed concurrently with this dissertation, outperforms both CPU-based Galois and Ligra by 6× and 3×, respectively. However, SIMD-X is only applicable for graphs that fit in the tested setup containing 16GB GPU memory, whereas real-world graphs have larger memory capacity requirement. For such large graphs, the role of CPU is essential in scalable graph processing. Due to the above reasons, single-machine large-memory CPU server is the platform of choice in this dissertation.

Chapter 3

Memory-Level Parallelism and Cache Hierarchy Analysis in Static Graph Workloads

3.1 Introduction and Contributions Overview

As discussed in Chapter 1, the performance of single-machine in-memory static graph analytics is bounded by the inefficiencies in the memory subsystem, making the cores stall as they wait for data to be fetched from the DRAM (Fig. 1.1). We develop an in-depth understanding of the memory-bound behavior observed in Fig. 1.1 by characterizing two features on a simulated multi-core architecture: (1) the memory-level parallelism (MLP) [80] in an out-of-order (OoO) core and (2) the request reuse distance in cache hierarchy. We extend or fill the gaps in prior characterization work [9–13] in two aspects. First, we perform a data-aware profiling which provides clearer guidelines on the management of specific data types for performance optimization. Second, with the flexibility of a simulated platform, we vary the instruction window and cache configuration design

parameters to explicitly explore their performance sensitivity. In contrast, prior work on performance analysis of static graph workloads [9–13] is not explicitly aware of the application data types. In addition, prior work misses the opportunity to perform a sensitivity analysis due to a real hardware based profiling. Beyond prior profiling work, our key findings include:

- Load-load dependency chains that involve specific application data types, rather than the instruction window size limitation, make up the key bottleneck in achieving a high MLP.

- Different graph data types exhibit heterogeneous reuse distances. The architectural consequences are (1) the private L2 cache shows negligible impact on improving system performance, (2) the shared L3 cache shows higher performance sensitivity, and (3) the graph property data type benefits the most from a larger shared L3 cache.

3.2 Experimental Setup

Profiling Platform: Profiling experiments have been done using SNIPER simulator, an x86 simulator based on the interval simulation model [81]. We selected SNIPER over other simulators because it has been validated against Intel Xeon X7460 Dunnington [81] and, with an enhanced core model, against Intel Xeon X5550 Nehalem [82]. Moreover, a recent study of x86 simulators simulating the Haswell microarchitecture has shown that SNIPER has the least error when validated against a real machine [83]. Cache access timings for different cache capacities were extracted using CACTI [84]. The baseline architecture is described in Table 6.1. We used fewer cores than typically present in a server node because previous profiling work has shown that resource utilization for

parallel and single-core executions are similar [9]. Hence, we do not expect the number of cores to change our observations. We marked the region of interest (ROI) in the application code. We ran the graph reading portion in cache warm-up mode and, upon entering the ROI, collected statistics for 600 million instructions across all the cores.

Table 3.1: Baseline Architecture

core	4 cores, ROB = 128-entry, load queue = 48-entry, store queue = 32-entry, reservation station entries = 36, dispatch width = issue width = commit width = 4, frequency = 2.66GHz
caches	3-level hierarchy, inclusive at all levels, writeback, least recently used (LRU) replacement policy, data and tags parallel access, 64B cacheline, separate L1 data and instruction caches
L1D/I cache	private, 32KB, 8-way set-associative, data access time = 4 cycles, tag access time = 1 cycle
L2 cache	private, 256KB, 8-way set-associative, data access time = 8 cycles, tag access time = 3 cycles
L3 cache (LLC)	shared, 8MB, 16-way set-associative, data access time = 30 cycles, tag access time = 10 cycles
DRAM	DDR3, device access latency = 45ns, queue delay modeled

Benchmark and datasets: We use the GAP benchmark [17] which consists of optimized multi-threaded C++ implementations of some of the most representative algorithms in graph analytics. For our profiling, we select GAP over a software framework to rule out any framework-related performance overheads and extract the true hardware bottlenecks[1]. We use five algorithms from GAP, which are summarized in Table 3.2. A summary of the datasets is shown in Table 6.2 (size = unweighted/weighted).

3.3 Analysis of the Memory-Level Parallelism

Observation 1: *Instruction window size is not the factor impeding MLP.* In general, a larger instruction window improves the hardware capability of utilizing more MLP

[1]Previous study shows a 2-30X slowdown of software frameworks compared to hand-optimized implementations [85].

Table 3.2: Algorithms

Algorithm	**Description**
Betweenness Centrality **(BC)**	Measure the centrality of a vertex, i.e., the number of shortest paths between any two other nodes passing through it
Breadth First Search **(BFS)**	Traverse a graph level by level
PageRank **(PR)**	Rank each vertex on the basis of the ranks of its neighbors
Single Source Shortest Path **(SSSP)**	Find the minimum cost path from a source vertex to all other vertices
Connected Components **(CC)**	Decompose the graph into a set of connected subgraphs

Table 3.3: Datasets

Dataset	**vertices**	**edges**	**Size**	**Description**
kron [17]	16.8M	260M	2.1GB/2GB*	synthetic
urand [17]	8.4M	134M	1.1GB/2.1GB	synthetic
orkut [86]	3M	117M	941MB/1.8GB	social network
livejournal [86]	4.8M	68.5M	597MB/1.1GB	social network
road [17]	23.9M	57.7M	806MB/1.3GB	mesh network

* Weighted graph is smaller due to generation from a smaller degree for a manageable simulation time.

for a memory-intensive application [87]. In addition, previous profiling work on a real machine concludes that the ROB size is the bottleneck in achieving a high MLP for graph analytics workloads [9]. However, by changing the design parameters in our simulator-based profiling, we observe that even a 4X larger instruction window fails to expose more MLP. As shown in Fig. 3.1a, for a 4X instruction window, the average increase in memory bandwidth utilization is only 2.7%. Fig. 3.1b shows the corresponding speedups. The average speedup is only 1.44%, which is very small for the large amount of allotted instruction window resources.

Observation 2: *Load-load dependency chains prevent achieving high MLP.* To understand why a larger ROB does not improve MLP, we track the dependencies of the load instructions in the ROB and find that the MLP is bounded by an inherent application-level dependency characteristic. For every load, we track its dependency backward in the ROB until we reach an older load instruction. We call the older load a producer

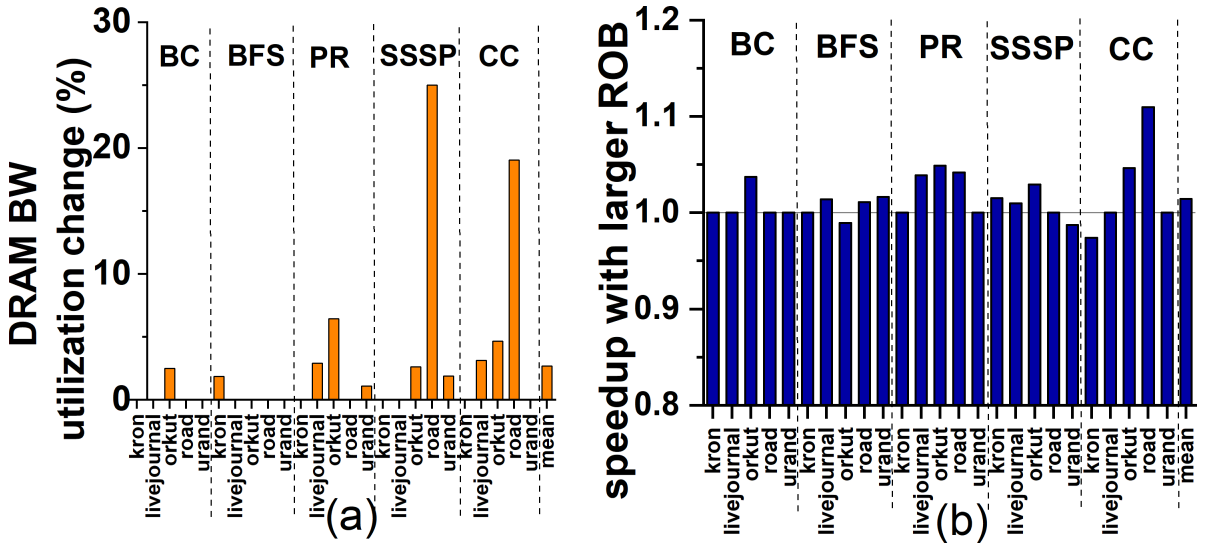


Figure 3.1: (a) Increase in DRAM bandwidth utilization and (b) overall speedup from a 4X larger ROB

load and the younger load a consumer load. We find that short producer-consumer load dependency chains are inherent in graph processing and can be a serious bottleneck in achieving a high MLP even for a larger ROB. The two loads cannot be parallelized as they are constrained by true data dependencies and have to be executed in program order. Fig. 3.2 shows that, on average, 43.2% of the loads are part of a dependency chain with an average chain length of only 2.5, where we define chain length as the number of instructions in the dependency chain.

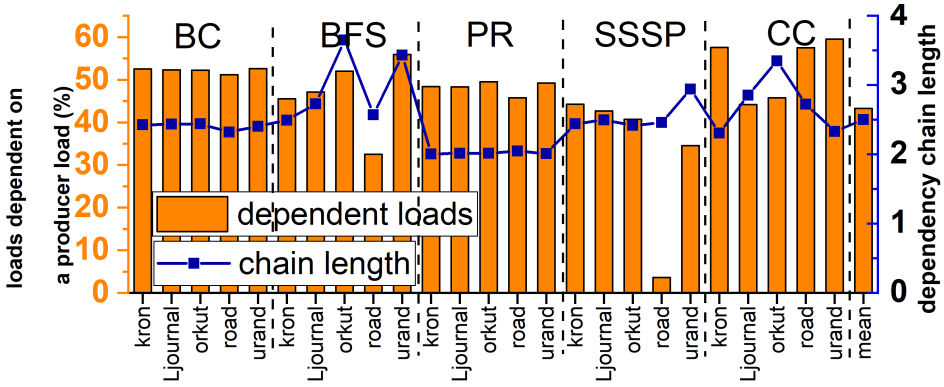


Figure 3.2: load-load dependency in ROB

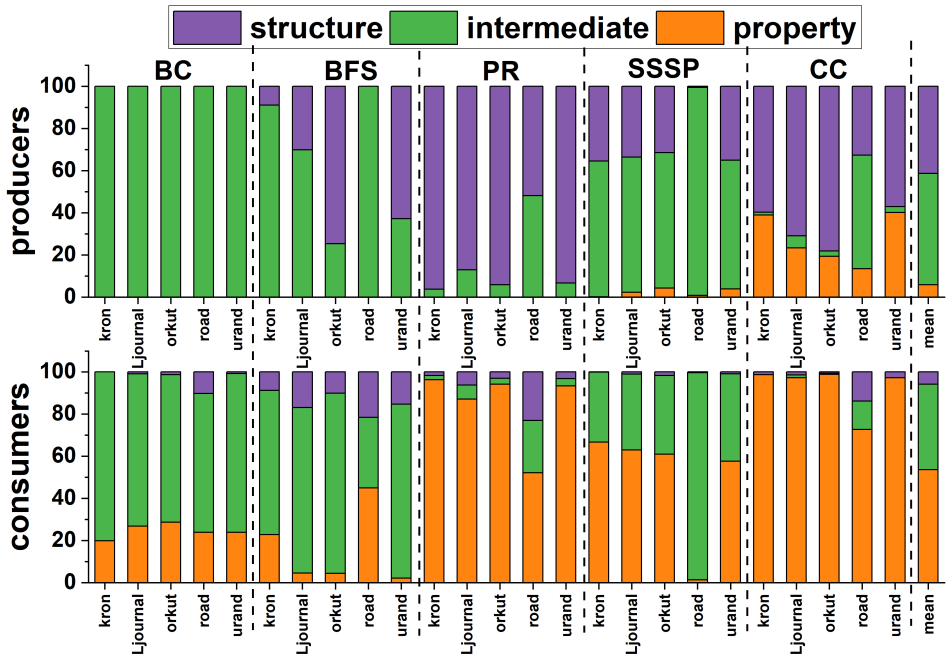


Figure 3.3: Breakdown of producer and consumer loads by application data type

Observation 3: *Graph property data is the consumer in a dependency chain.* To identify the position of each application data type in the observed load-load dependency chains, we show the breakdown of producer and consumer loads by data type in Fig. 3.3. On average, we find that the graph property data is mostly a consumer (53.6%) rather than a producer (5.9%). Issuing graph property data loads is delayed and cannot be parallelized because it has to depend on a producer load for its address calculation. Fig. 3.3 also shows that, on average, graph structure data is mostly a producer (41.4%) rather than a consumer (6%).

3.4 Analysis of the Cache Hierarchy

Observation 1: *The private L2 cache shows negligible performance sensitivity, whereas the shared LLC shows higher performance sensitivity.* As shown in Fig. 3.4, we vary the LLC size from 8MB to 64MB and find the optimal point of 17.4% (max 3.25X) perfor-

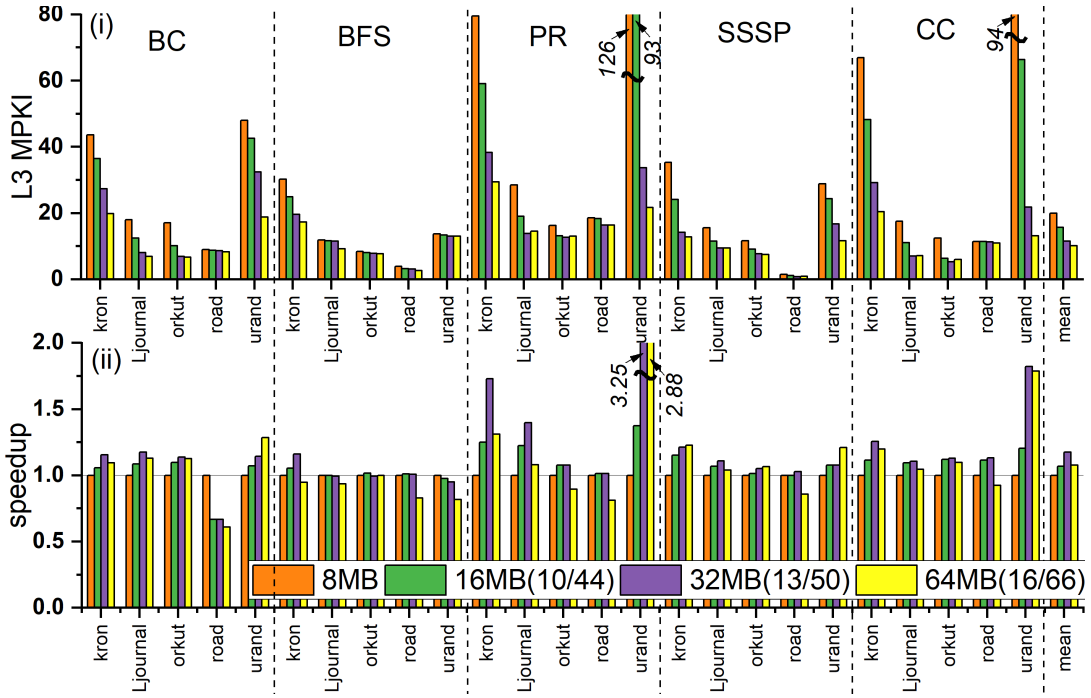


Figure 3.4: Sensitivity of i) L3 MPKI and ii) system performance to shared L3 cache size ((X/Y)=access times for (tags/data) in cycles)

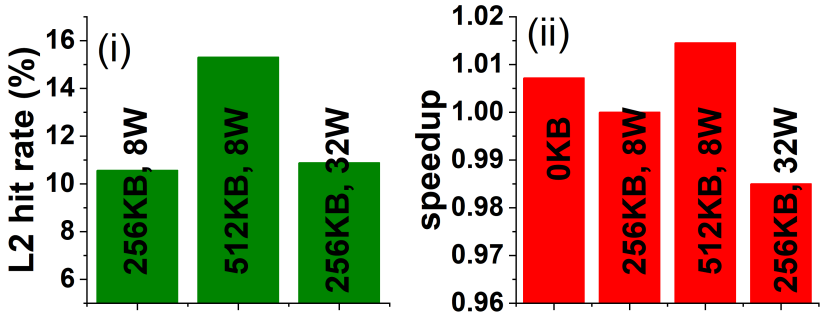


Figure 3.5: Sensitivity of (i) L2 cache hit rate and (ii) system performance to private L2 cache configurations (average across all benchmarks))

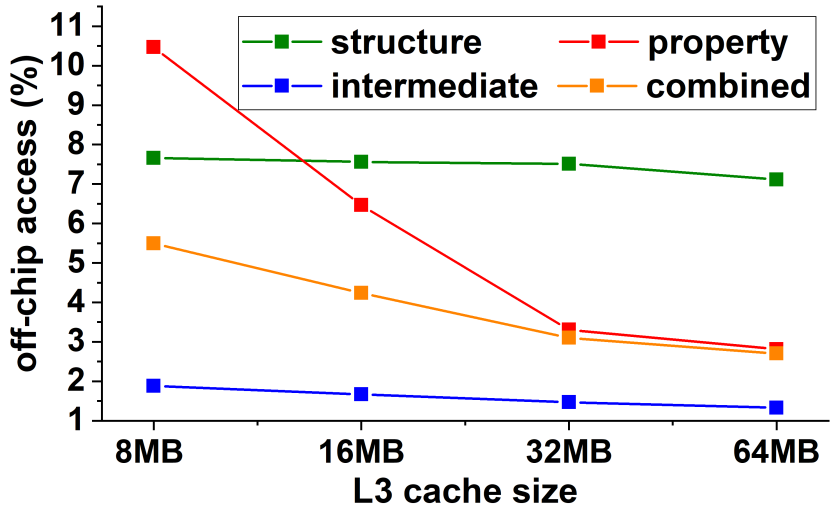


Figure 3.6: Effect of larger L3 cache on off-chip accesses of different data types (average across all benchmarks)

mance improvement for a 4X increase in the LLC capacity. The mean LLC MPKI (misses per kilo instructions) is reduced from 20 in the baseline to 16 (16MB) to 12 (32MB) to 10 (64MB). The corresponding speedups are 7%, 17.4%, and 7.6%. The optimal point is a balance between a reduced miss rate and a larger LLC access latency.

Fig. 3.5(i) shows that the L2 hit rate (which is already very low at 10.6% in the baseline) increases to only 15.3% after a 2X increase in the capacity while a 4X increase in set associativity has no impact (hit rate rises to only 10.9%). Fig. 3.5(ii) shows that the system performance exhibits little sensitivity to different L2 cache configurations (in both capacity and set associativity). The leftmost bar represents an architecture with no private L2 caches and no slowdown compared to a 256KB cache. Therefore, an architecture without private L2 caches is just as fine for graph processing.

Observation 2: *Property data is the primary beneficiary of LLC capacity.* To understand which data type benefits from a larger LLC, Fig. 3.6 shows, for each data type, the percentage of memory references that ends up getting data from the DRAM. We observe that the most reduction in the number of off-chip accesses comes from the property data. Structure and intermediate data benefit negligibly from a higher L3

capacity. Intermediate data is already accessed mostly in on-chip caches since only 1.9% of the accesses to this data type is DRAM-bound in the baseline. On the other hand, structure data has a higher percentage of off-chip accesses (7.5%), which remains mostly irresponsive to a larger LLC capacity.

Observation 3: *Graph structure cacheline has the largest reuse distance among all the data types. Graph property cacheline has a larger reuse distance than that serviced by the L2 cache.* To further understand the different performance sensitivities of the L2 and L3 caches, we break down the memory hierarchy usage by application data type as shown in Fig. 3.7. In most benchmarks, accesses to the structure data are serviced by the L1 cache and the DRAM, which indicates that a cacheline missed in L1 is one that was referenced in the distant past such that it has been evicted from both the L2 and L3 caches. The fact that the reuse distance is beyond the servicing capability of the LLC explains why a larger LLC fails to significantly reduce the proportion of off-chip structure accesses in Fig. 3.6. On the other hand, most of the property data loads missed in the L1 cache cannot be serviced by the L2 cache but can be serviced by the LLC and the DRAM. Overall, the LLC is more useful in servicing property accesses rather than structure accesses. Thus, the property cacheline has a comparatively smaller reuse distance that is still larger than that captured by the L2 cache. Finally, Fig. 3.7 provides evidence that the accesses to intermediate data are mostly on-chip cache hits in the L1 cache and the LLC. The reuse distances of the three data types explain why the private L2 cache fails to service memory requests and shows negligible benefit.

3.5 Conclusion

We perform a data-aware characterization of static graph workloads in the GAP benchmark suite on a simulated multi-core architecture in order to study the bottlenecks in the MLP and the cache hierarchy in graph analytics. We show that, load-load de-

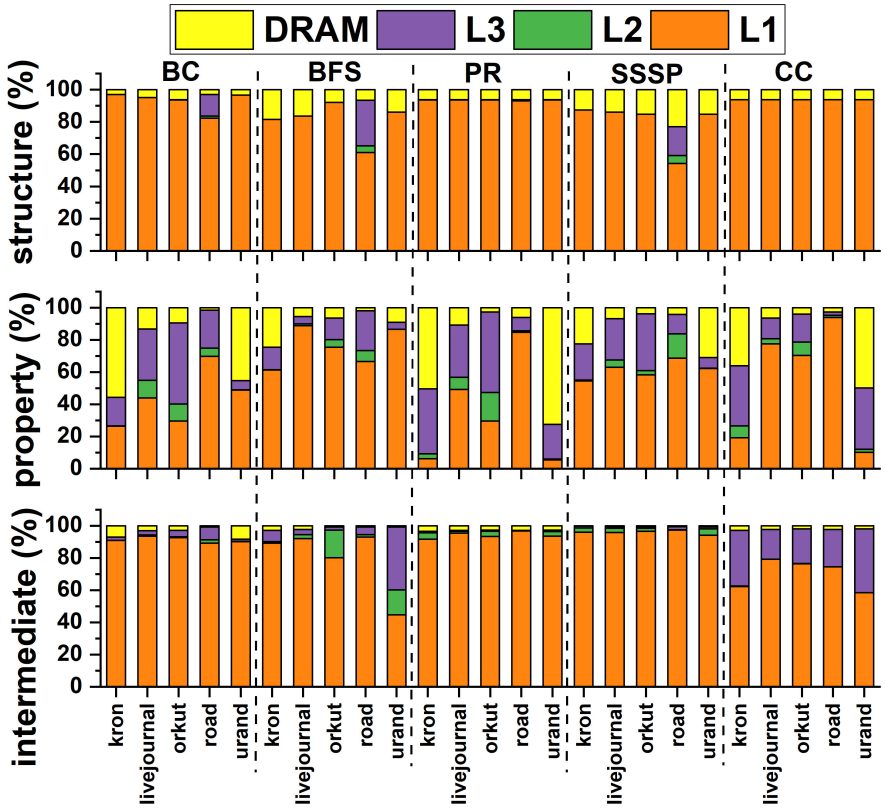


Figure 3.7: Breakdown of memory hierarchy usage by application data type

pendency chains that involve specific application data types, rather than the instruction window size limitation, make up the key bottleneck in achieving a high MLP. In addition, we find that heterogeneous reuse distances of the application data types cause different performance impacts of the L2 and the L3 cache levels. In the next chapter, we show how our analysis provides opportunities and guidelines for a prefetching solution to improve the performance of static graph workloads.

Chapter 4

DROPLET: a Data-awaRe decOuPLed prEfeTcher for Static Graphs

4.1 Introduction and Contributions Overview

We use the guidelines from our characterization of static graph workloads in Chapter 3 to design DROPLET, a Data-awaRe decOuPLed prEfeTcher for graphs. From Chapter 3, we find that the memory-bound stalling behavior in graph analytics arises due to two issues. First, heterogeneous reuse distances of different data types leads to intensive DRAM accesses to retrieve structure and property data. Second, MLP is low due to load-load dependency chains, limiting the possibility of overlapping DRAM accesses. Our profiling motivates the adoption of prefetching as a latency tolerance technique for graph analytics. A good prefetcher can fix the first issue by locating data in on-chip caches ahead of the demand accesses. Prefetching can also bring an additional benefit by mitigating the effect of low MLP. A dependency chain with a consumer property data means serialization in the address calculation and a delay in issuing the property data load. However, once issued, prefetching ensures that the property data

hits on-chip, mitigating low-MLP induced serially exposed latency. However, due to heterogeneous reuse distances of the different graph data types, it is challenging to employ simple hardware prefetchers that can efficiently prefetch all data types. To address this challenge, we leverage our observations from the profiling to make prefetch decisions for DROPLET architecture, as summarized in Table 4.1. DROPLET is a physically decoupled but functionally cooperative prefetcher co-located at the L2 cache and at the memory controller. We adopt a decoupled design to overcome the serialization due to the dependency between different graph data types. Moreover, DROPLET is data-aware because it prefetches different graph data types differently according to their intrinsic reuse distances. DROPLET achieves 19%-102% performance improvement over a no-prefetch baseline, 9%-74% performance improvement over a conventional stream prefetcher, 14%-74% performance improvement over a Variable Length Delta Prefetcher (VLDP) [23], and 19%-115% performance improvement over a delta correlation prefetcher implemented as a global history buffer (GHB) [24]. DROPLET performs 4%-12.5% better than a monolithic L1 prefetcher similar to the state-of-the-art prefetcher for graphs [15].

4.2 DROPLET architecture

We first provide an overview of our proposed prefetcher. Next, we discuss the detailed architecture design of the components of DROPLET.

4.2.1 Overview and Design Choice

We propose a physically decoupled but functionally cooperative prefetcher co-located at the L2 cache level and at the MC. Specifically, DROPLET consists of two data-aware components: the L2 streamer for prefetching structure data (Fig. 4.1 1) and the memory controller based property prefetcher (MPP) (Fig. 4.1 2) which is guided by the

Table 4.1: Prefetch decisions based on profiling observations

Profiling Observation	**Prefetch Design Question**	**Answer**
Negligible impact of L2.	In which cache level to put prefetched data?	L2 cache, because (1) no risk of cache pollution and (2) technique to make the under-utilized L2 cache useful.
Intermediate data are mostly cached, structure/property data are not.	What to prefetch?	Structure and property data.
• Structure data exhibits a large reuse distance with a pattern: a cacheline missed in the L1 cache is almost always serviced by the DRAM. • Property data exhibits a randomly large reuse distance which is larger than the L2 stack depth, leading to heavy LLC and DRAM accesses. • In load-load dependency chains, property data is mostly the consumer, whereas structure data is mostly the producer.	How to prefetch?	• Prefetch structure data from the DRAM in a streaming fashion. • Prefetch property data using explicit address calculation due to difficult-to-predict large reuse distance. • Let the calculation of target property prefetch addresses be guided by structure data. • Address calculation of target property prefetches is a serialized process. Decoupling the prefetcher will help break the serialization.
Load-load dependency chains are short.	When to prefetch?	If property prefetches are guided by structure *demand* data, they could be late since dependency chains are short. Hence, property prefetches should be guided by structure *prefetches*.

structure stream requests from the L2 streamer. Fig. 4.1 shows the entire prefetching flow. The blue path shows the flow of prefetching the structure data using the data-aware L2 streamer. The stream prefetcher is triggered only by structure data and the structure prefetch request is almost always guaranteed to go to the DRAM as observed in our profiling. On the refill path from the DRAM, the prefetched structure cacheline is transferred all the way to L2. Also, a copy of the structure data is forwarded to the MPP, which triggers the property data prefetching. The green path shows the prefetching flow of the property data. The MPP reacts to the prefetched structure cacheline by scanning it for the neighbor node IDs, computing the virtual addresses of the target property prefetches using those neighbor IDs, and performing virtual-physical address translation. To avoid unnecessary DRAM accesses for property data that may already be on-chip, the physical prefetch address is used to check the coherence engine. If not on-chip, the property prefetch address is queued in the MC. Upon being serviced by the DRAM, the cacheline is sent to the LLC and the private L2 cache. Instead, if the to-be-prefetched property cacheline is detected to be on-chip, it is further copied from the inclusive LLC into the private L2 cache.

Our design choices are based on our profiling observations as summarized in Table 4.1. The L2 cache is the location in which the L2 streamer brings in structure prefetches and the MPP sends property prefetches. Despite this fact, we design a decoupled property prefetcher in the MC in order to break the serialization arising from consumer property data in a dependency chain. A single prefetcher at the L2 cache would have to wait until the structure prefetch returns to the L2 cache before the property address can be calculated and issued for prefetching. By decoupling the prefetcher, the property address can be calculated as soon as the producer structure prefetch arrives on the chip at the MC. This overlaps the return of the structure prefetch on the refill path through the caches and the issuing of the subsequent property prefetch, breaking the serialization. A

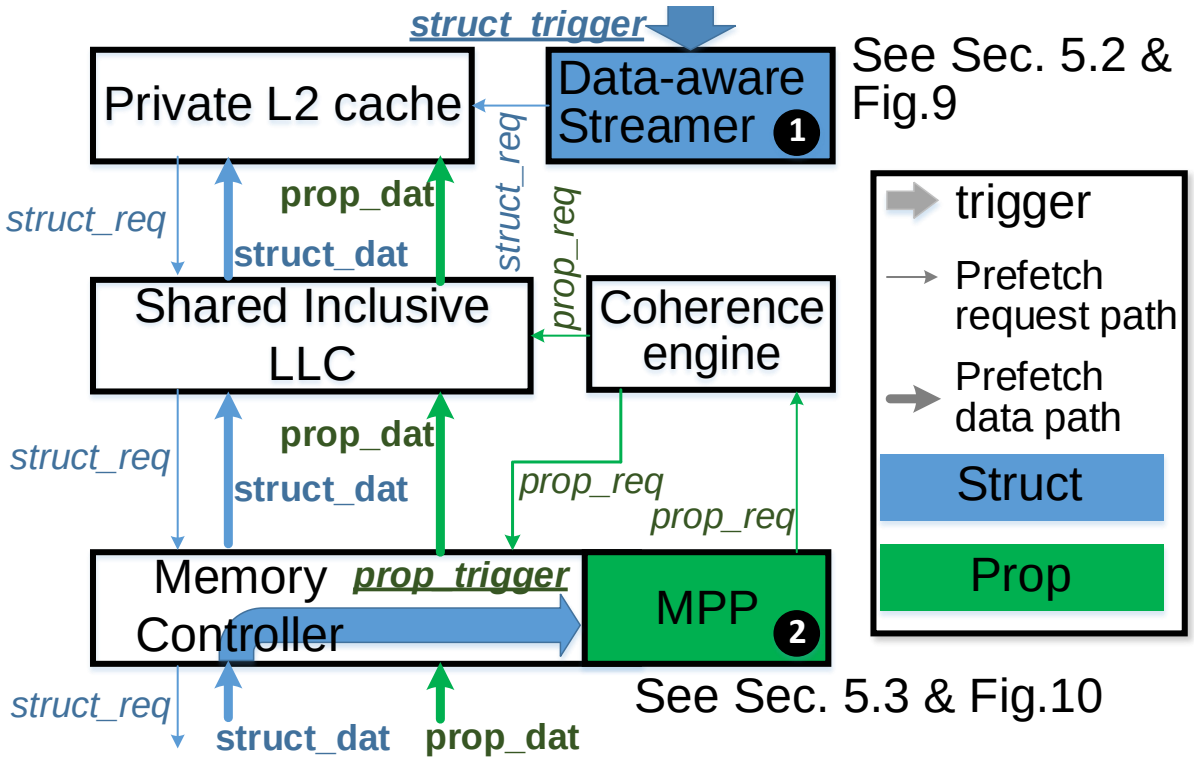


Figure 4.1: Overview of DROPLET

previous work, which *dynamically offloads* and *accelerates* dependent load chains in the MC, shows a 20% lower latency when a dependent load is issued from the MC rather than from the core side [88]. We use this insight in our *prefetching* scheme to achieve better timeliness by aggressively issuing property prefetch requests directly from the MC.

4.2.2 L2 Streamer for Structure Data

First, we discuss the difficulties posed by a conventional streamer for graph workloads. Next, to address these challenges, we propose the design of a data-aware L2 streamer.

Shortcomings of a Conventional Streamer

A conventional L2 streamer, as shown in Fig. 4.2(a), can snoop all cacheblock addresses missed in the L1 cache and can track multiple streams, each with its own tracker (tracking address) [89]. Once a tracker has been allocated, the stream requires two additional miss addresses to confirm a stream before starting to prefetch [89]. In graph processing, such a streamer is detrimental to both structure and property data in terms of prefetch accuracy and coverage. First, due to the streamer's ability to snoop all L1 miss addresses, many trackers in the streamer get wastefully assigned to track pages corresponding to the property and the intermediate data types. These trackers are wasteful due to one of the two reasons: (1) often, a stream is not successfully detected due to large reuse distances, which results in small property and intermediate prefetch volumes and low coverage; (2) due to random accesses, these trackers get the wrong signals, i.e., random streams are detected, leading to mostly useless prefetches (low prefetch accuracy). The direct consequence of property and intermediate data trackers is the reduction in the effective number of trackers available for structure data (the data type which actually shows stream behavior) at any given time. This reduces the volume of structure

prefetches and the coverage for structure data.

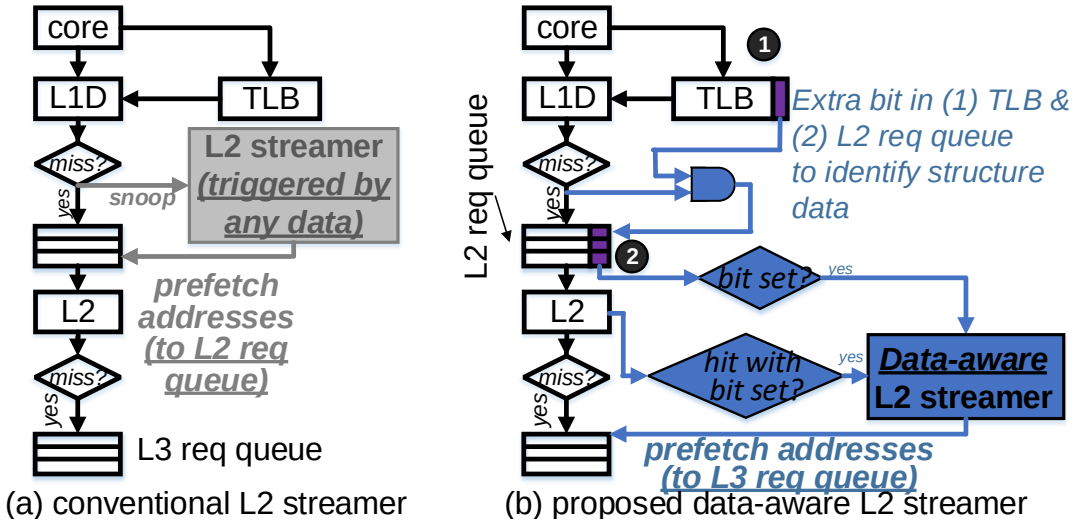


Figure 4.2: L2 streamer for prefetching structure data. Extra bits in purple and newly added cache controller decision-making in blue.

Data-aware Streamer in DROPLET

To address the problems of a conventional streamer, our proposed streamer is data-aware, i.e., it operates only on structure data to bring in a high volume of accurate structure prefetches. Fig. 4.2(b) shows how this is achieved. We use a variant of *malloc* to allocate graph structure data (see Section 4.2.5), which updates page table entries with an extra bit ("1"=structure data). Upon address translation at the L1D cache, this extra bit in the TLB entry (Fig. 4.2(b) 1) is copied to the L1D cache controller. When an access misses in the L1D cache, the L1D controller buffers this extra bit, together with the miss address, into the L2 request queue. Hence, the L2 request queue is augmented with an additional bit (Fig. 4.2(b) 2) to identify whether a request address corresponds to the structure data. The L2 cache controller in our design preferentially guides into the streamer only the addresses in the request queue which belong to structure data (as recognized by the extra bit being "1"). Another input to the streamer is a feedback

from the L2 cache. When a L2 hit occurs to an address corresponding to structure data, the address is sent to the streamer. Finally, the L2 streamer buffers its target prefetch addresses in the L3 request queue and stores the prefetched data in the L2 cache.

In contrast to a conventional streamer, as shown in Fig. 4.2(a), our design contains two fundamental modifications guided by our observations in Section **??**. First, our data-aware L2 streamer is triggered only by *structure data* to overcome the shortcomings of a conventional streamer mentioned in Section 4.2.2. Second, as opposed to a conventional L2 streamer that buffers the target prefetch addresses in the L2 request queue, we buffer them in the L3 request queue. This design choice is guided by our observation that new structure cachelines are serviced by a lower level in the memory hierarchy. Later in Section 4.3, we quantitatively analyze the superiority of our data-aware structure streamer over the conventional design in terms of both performance and bandwidth savings.

4.2.3 Memory Controller (MC)

First, the memory request buffer (MRB) is used to give the MC the knowledge of (1) whether a cacheline coming from the DRAM corresponds to a structure prefetch and (2) which core sent the structure prefetch request. Second, we incorporate the MPP unit.

Memory Request Buffer (MRB)

When cachelines are on their refill path from the DRAM, the MC needs to filter the ones corresponding to structure prefetches to transfer their copies to the MPP for property prefetching. To enable this, we reinterpret the C-bit (criticality bit) in the baseline MRB, as shown in Fig. 4.3 (in the lower left corner), which differentiates a prefetch request from a demand request for priority-based scheduling purposes in modern MCs [90]. If the C-bit is set, in addition to indicating a prefetch request, it specifically

indicates a *structure* prefetch since it comes from the L2 streamer, which only sends structure prefetch requests. We also add a core ID field to give the MPP the knowledge of which core sent the structure prefetch request so that it can later send the property prefetches into that core's private L2 cache.

MC based Property Prefetcher (MPP)

As shown in Fig. 4.3, the MPP consists of a property address generator (PAG), a FIFO virtual address buffer (VAB), a near-memory TLB (MTLB), and a FIFO physical address buffer (PAB). If a cacheline from the DRAM is detected to be a structure prefetch (see Section 4.2.3), its copy is transferred to the MPP.

The prefetched structure cacheline is propagated to the PAG which is shown in detail in Fig. 4.3. The PAG scans the cacheline in parallel for neighbor IDs that are indices to the property data array. To calculate the target virtual address of the property prefetch from a scanned neighbor ID, we use the following equation:

$$\textbf{property address} = \textbf{base} + \textbf{4} \times \textbf{neighbor ID} \tag{4.1}$$

where *base* is the base virtual address of the property array and the granularity of the property data is 4B. The PAG receives two pieces of information from the software (see Section 4.2.5): (1) the *base* in Equation 4.1 and (2) the granularity at which the structure cacheline needs to be scanned (4B for unweighted graphs and 8B for weighted graphs). One structure cacheline can generate 8 and 16 property addresses per cycle for weighted and unweighted graphs, respectively.

The virtual addresses generated by the PAG, together with the corresponding core ID, are buffered in the VAB for translation. The translation happens through a small MTLB (see Section 4.2.3), upon which the resulting physical address is buffered in the

PAB. The physical address is then used to check the coherence engine and determine the subsequent prefetch path of the property data, as described in Section 4.2.1.

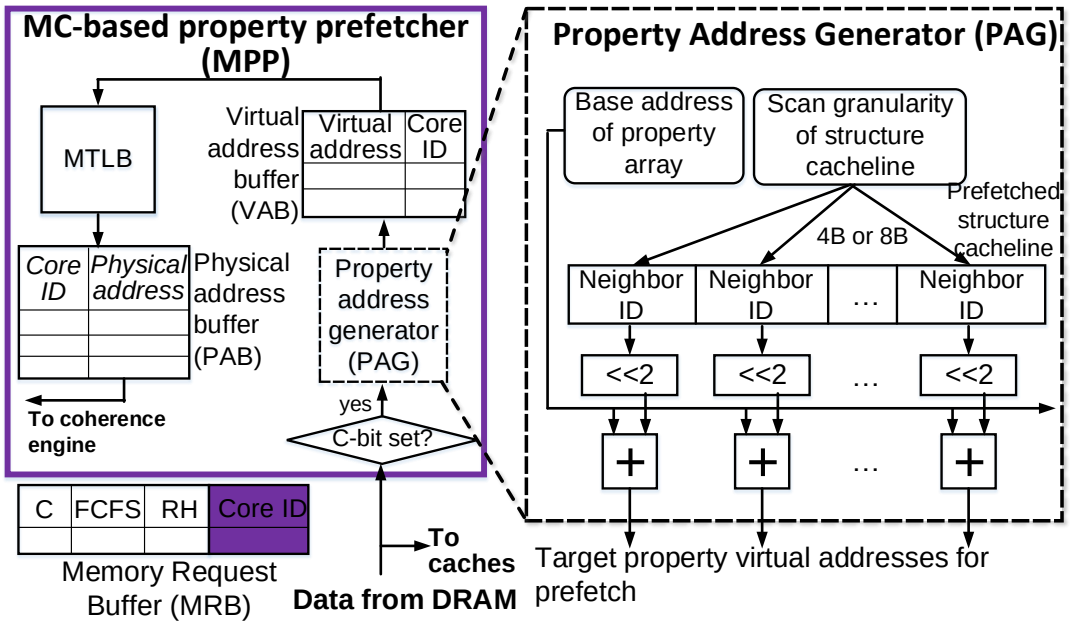


Figure 4.3: MC-based Property Prefetcher (MPP)

Virtual Address Translation in MTLB

The MTLB caches mappings of the property data pages to allow virtual-physical translation of the property prefetch addresses in the MPP. A MTLB miss is handled by page-walking, but a property prefetch address encountering a page fault is simply dropped. Like core-side TLBs, the MTLB should also be involved in the TLB shootdown process so that the MPP does not continue using stale mappings to prefetch wrong data. To optimize the coherency maintenance between the core-side TLBs and the MTLB, and to reduce the number of invalidations to the MTLB, we leverage (1) the extra bit in the TLBs used to differentiate between structure and non-structure data and (2) the fact that the MTLB caches only property mappings. Due to these two features, during a TLB shootdown, MTLB entries are invalidated using only the core-side TLB invalidations for

entries that have an extra bit with the value "0" (indicating non-structure data).

4.2.4 Hardware Overhead

DROPLET incurs negligible area overhead. The data-aware streamer in DROPLET requires storing extra information in the page table and the L2 request queue entries. In the hierarchical paging scheme in x86-64, each paging structure consists of 512 64-bit entries, leading to a 4KB storage [91]. The addition of an extra bit in each page table entry to identify structure data results in a 64B (1.56%) storage overhead in the paging structure. Assuming a 32-entry L2 request queue [92], with each entry containing the miss address and the status [93], the addition of an extra bit results in a 4B (1.54%) queue storage overhead. In addition, property prefetching in DROPLET introduces the MPP and requires additional storage in the MRB. Using McPAT integrated with SNIPER, we find the area of the baseline chip to be 188 mm^2 in a 45nm technology node. The area of the MPP, with its parameters shown in Table 4.2, is 0.0654 mm^2, resulting in a 0.0348% area overhead with respect to the entire chip. With a 7.7KB storage, the VAB, the PAB, and the MTLB comprise 95.5% of the total MPP area. For the MRB, the additional storage required for the core ID field for our quad-core system is only 64B, if assuming a 256-entry MRB.

4.2.5 Design Discussion

System support for address identification: DROPLET is data-aware in two aspects. First, the L2 streamer is only triggered by structure data. Second, the MPP prefetches only property data with the knowledge of the base address of the property array and the scan granularity of the structure cacheline. To obtain this information, DROPLET needs help from the operating system and the graph data allocation layer

of graph processing frameworks. Similar to prior work [94, 95], we develop a specialized *malloc* function which can label all allocated pages with an extra bit in the page table to help identify structure data during address translation at the L1D cache (the extra bit being "1" indicates structure data). The specialized *malloc* function can also pass in some information to be written into the hardware registers in the MPP. When allocating the structure data, the *malloc* function writes the scan granularity of the structure cacheline in the MPP. The *malloc* function for the property data writes the base address of the property array in the MPP. We add a special store instruction to provide an interface for the processor to write these two registers in the MPP [1].

User Interface: DROPLET requires the above *malloc* modification only at the framework level and is transparent to user source code. This is because, in the system stack for graph processing frameworks, the layer of graph data allocation and management (the layer at which we introduce a specialized *malloc*) is separate from user code [95]. Hence, our modification can be handled "behind the scenes" by the data management layer without modifying the API for users.

Applicability to different programming models and data layouts: In this work, we assume a CSR data layout and a programming abstraction focused on the vertex. DROPLET is easily extensible to other abstractions because a common aspect across most abstractions is that they maintain data in two distinct parts which can be mapped to our data type terminology (structure and property data). For example, in edge-centric graph processing [55, 72], structure data would be equivalent to a sorted or unsorted edge array which is typically streamed in from slower to faster memory [55, 72]. DROPLET can prefetch these edge streams and use them to trigger a MPP located near the slower memory to prefetch property data. Moreover, DROPLET can easily handle

[1] The configuration registers of the MPP are part of the context and are backed up and restored when switching between graph analytics processes.

multi-property graphs by using the information from prefetched structure data to index one property array (in case the property data layout is an array of vectors) or multiple property arrays (in the case of separate arrays for each property).

Applicability to larger graphs: DROPLET is based on the observations obtained from profiling graph processing workloads on a simulated multi-core architecture. As opposed to a real machine, a simulated platform provides much higher flexibility for data-aware profiling and for studying the explicit performance sensitivities of the instruction window and the cache hierarchy. However, the trade-off is the restriction to somewhat small datasets to achieve manageable simulation times. However, our profiling conclusions (hence the effectiveness of our proposed DROPLET design) still hold for larger graphs because of two aspects. First, we use small datasets that are still larger than that fully captured by the on-chip caches, which stresses the memory subsystem sufficiently. Second, we explain the observed architecture bottlenecks in terms of inherent data type features such as dependency characteristics and reuse distances, which are independent of the data size.

Multiple MCs: Our experimental platform consists of a single MC in a quad-core system. However, platforms with more cores may contain multiple MCs and data may be interleaved across DRAM channels connected to different MCs. It is possible that a property prefetch address generated by the MPP of one MC may actually be located in a DRAM channel connected to another MC. In such a case, we adopt a technique similar to prior work [88]. The property prefetch request, together with the core ID, is directly sent to the MRB of the destination MC for prefetching.

4.3 DROPLET Evaluation

In this section, we discuss the experimental setup and the evaluation results for DROPLET.

4.3.1 Prefetch Experiment Setup

Beyond Table 6.1, the features of the prefetchers used for evaluation are shown in Table 4.2. Evaluation is performed for six prefetcher configurations listed below.

GHB [24]: This is a G/DC (Global/Delta Correlation) L2 prefetcher.

VLDP [23]: This is a L2 prefetcher which, unlike GHB, can use longer and variable delta histories to make more accurate prefetch predictions.

stream: This is the conventional L2 streamer which can snoop all L1 miss addresses.

streamMPP1: This is a conventional L2 streamer together with a MPP. This configuration shows the benefit of letting property prefetching be guided by structure streaming. However, since the L2 streamer is not data-aware, relying only on the C-bit in the MRB to identify structure data does not work. So, we use MPP1 which is equivalent to MPP equipped with the ability to recognize structure data.

DROPLET: Unlike stream and streamMPP1, the L2 streamer is structure-only. When compared to streamMPP1, DROPLET shows the additional benefit by restricting the streamer to work on structure data only.

monoDROPLETL1: This prefetcher is a data-aware streamer + MPP1[2] implemented monolithically at the L1 cache, an arrangement close to the state-of-the-art graph prefetching proposition of Ainsworth et al. [15]. Although [15] uses a worklist-triggered recursive prefetcher instead of a data-aware streamer + MPP1, monoDROPLETL1 imitates their design philosophies of 1) a monolithic prefetcher and 2) bringing all prefetches into the

[2]The MPP1 in monoDROPLETL1 can reuse the core-side TLB and does not require a MTLB.

L1 cache. Hence, we select this design point to evaluate if DROPLET provides any performance benefit over a [15]-like **monolithic L1** prefetcher.

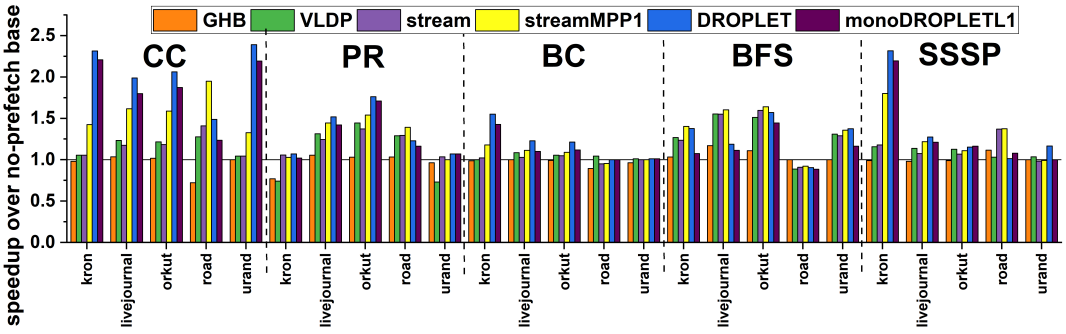
Table 4.2: Prefetchers for evaluation

L2 GHB	index table size = 512, buffer size = 512
L2 VLDP	implemented using code from [92], last 64 pages tracked by DHB, 64-entry OPT, 3 cascaded 64-entry DPTs
L2 streamer	implemented as described in section 2.1 of [89], prefetch distance=16, number of streams=64, stops at page boundary
MPP	address generation latency in PAG = 2 cycles, 512-entry VAB and PAB, 128-entry MTLB, 2 64-bit registers, coherence engine checking overhead=10 cycles
MPP1	MPP augmented with the ability to identify a prefetched structure cacheline

4.3.2 Performance Results

Fig. 4.4(a) and 4.4(b) show the performance improvement of the six configurations normalized to a no-prefetch baseline. Among all of them, DROPLET provides the best performance for CC (102%), PR (30%), BC (19%), and SSSP (32%). In these four algorithms, the only exception is the *road* dataset (CC-*road*, PR-*road*, and SSSP-*road*) where streamMPP1 is the best performer (see Section 4.3.3). For BFS, DROPLET provides a 26% performance improvement, but the best prefetcher is streamMPP1 with an average improvement of 36% (see Section 4.3.3). DROPLET could easily be extended to adaptively turn off the streamer's data-awareness to convert it into the streamMPP1 design. In that case, our design would be no worse than streamMPP1 for BFS and the *road* dataset.

Compared to a conventional stream prefetcher, DROPLET provides performance improvements of 74% for CC, 9% for PR, 19% for BC, and 16.8% for SSSP. These im-



(a)

Scheme	CC	PR	BC	BFS	SSSP
GHB	0.94	0.96	0.97	1.06	1.01
VLDP	1.16	1.06	1.04	1.28	1.1
stream	1.16	1.19	1	1.29	1.13
streamMPP1	1.57	1.26	1.06	**1.36**	1.27
DROPLET	**2.02**	**1.30**	**1.19**	1.26	**1.32**
monoDROPLETL1	1.82	1.25	1.12	1.12	1.27

(b)

Figure 4.4: (a) Performance improvement of the six configurations; (b) Performance summary for Fig. 4.4(a) (each entry is the geomean of the speedups across all the five datasets).

provements come from both the MPP and the data-aware streamer, as shown by the progressive speedup from stream to DROPLET. For BFS, streamMPP1 outperforms the conventional streamer by 5.4% but DROPLET slightly degrades performance by 2.3%.

Compared to monoDROPLETL1 (close to [15]), DROPLET provides performance improvements of 11% for CC, 4% for PR, 6% for BC, 12.5% for BFS, and 4% for SSSP. DROPLET differs from monoDROPLETL1 by 1) adopting a physically decoupled design and 2) prefetching into the L2 instead of the L1. DROPLET performs better because it achieves better prefetch timeliness by decoupling the structure and property prefetching (Section 4.2.1). The benefit of decoupling is more so because computation per memory access is very low in graph processing. In addition, DROPLET does not pollute the more useful L1 cache. Instead, it prefetches into the L2 cache which is poorly utilized in graph processing (Section 5.5.3). In addition to the performance benefit, DROPLET offers two

critical advantages over [15] in terms of practicality and simplicity. First, [15] triggers its prefetcher with an *implementation specific* data structure called the worklist. However, many all-active algorithms are the simplest to implement without having to maintain a worklist [17]. For [15], these algorithms have to re-written to integrate a worklist data structure. In contrast, DROPLET does not depend on any worklist, leverages the intrinsic reuse distance feature of graph structure data to trigger the prefetcher, and introduces a special *malloc* which is transparent to user code (Section 4.2.5). Second, unlike [15], we do not require additional hardware to ensure prefetch timeliness. We achieve timeliness by using a decoupled design to break the serialization between structure and property data.

DROPLET outperforms G/DC GHB by 115% for CC, 35% for PR, 23% for BC, 19% for BFS, and 31% for SSSP. GHB is overall the least performing prefetcher in our evaluation. This is because it is hard to identify consistent correlation patterns in graph processing due to the heterogeneous reuse distances of different data types.

Compared to VLDP, DROPLET provides performance improvements of 74% for CC, 23% for PR, 14% for BC, and 20% for SSSP. For BFS, streamMPP1 outperforms VLDP by 6% but DROPLET slightly degrades performance by 1.6%. On average, the performance improvement of DROPLET compared to VLDP is similar to its improvement compared to the conventional stream prefetcher. Due to complex reuse distances of different data types, delta histories may not always make effective prefetch predictions.

4.3.3 Explaining DROPLET's Performance

In this section, we zoom in on DROPLET to demystify its performance benefits. Since DROPLET is an enhancement upon the L2 streamer to include a data-aware streamer and a MPP, we compare to stream and streamMPP1 configurations to show the additional

effect of each component.

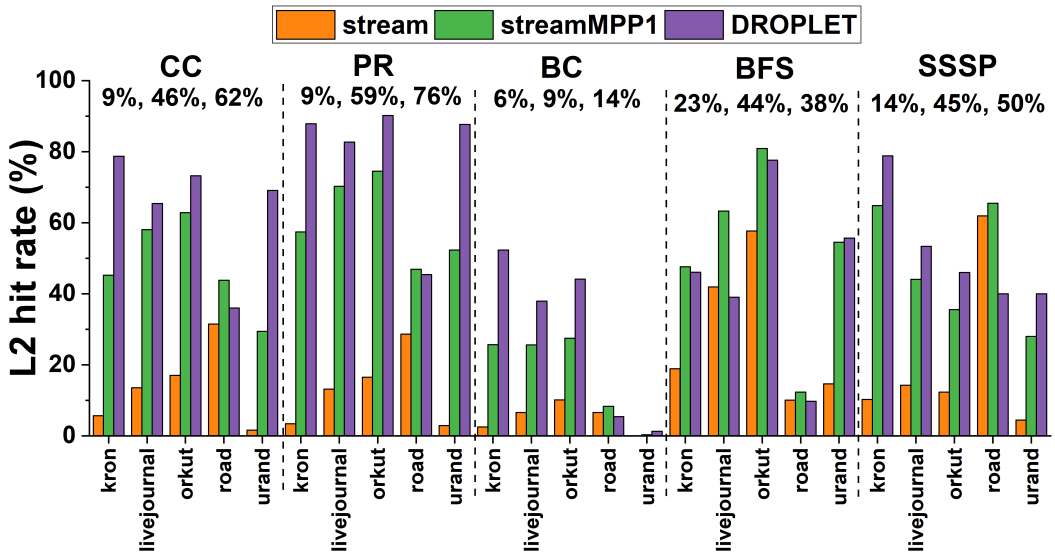


Figure 4.5: L2 cache performance. For each algorithm, the set of numbers shows the average for the three prefetch configurations across the five datasets.

Benefit to Demand Accesses

L2 cache performance: Fig. 4.5 shows that DROPLET converts the seriously under-utilized L2 cache (Fig. 3.5) into a higher performance resource by increasing the L2 hit rate to 62%, 76%, 14%, 38%, and 50% for CC, PR, BC[3], BFS, and SSSP, respectively. The L2 hit rate for DROPLET is the highest on average for CC, PR, BC, and SSSP. Fig. 4.5 also explains why streamMPP1 is the ideal prefetcher for the *road* dataset and most BFS benchmarks. For these benchmarks, the conventional streamer provides a comparatively higher cache performance, indicating that the streamer may also be effective at capturing some property prefetches. Hence, by making the streamer structure-only in DROPLET, we lose the streamer-induced property prefetches, leading to a lower L2 cache hit rate when using DROPLET over streamMPP1.

[3]The value is relatively lower due to BC-*road* and BC-*urand* exceptions which do not benefit highly from any prefetcher configuration. However, DROPLET causes no slowdown for them (Fig. 4.4(a)).

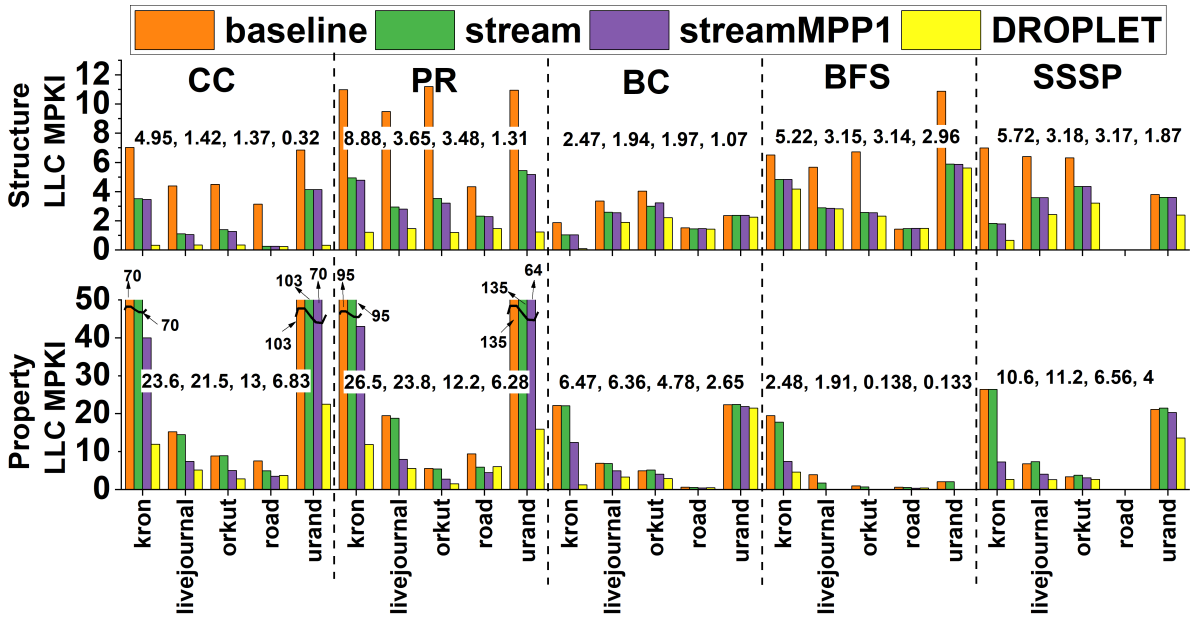


Figure 4.6: Off-chip demand accesses. For each algorithm, the set of numbers shows the average for the baseline and the three prefetch configurations across the five datasets.

Off-chip demand accesses: To understand how the two components of DROPLET (MPP and data-aware streamer) affect the off-chip demand accesses to structure and property data, Fig. 4.6 breaks down the *demand* MPKI from the LLC by data type. Below, we summarize our observations on the additive benefit of each prefetch configuration:

(1) Compared to the no-prefetch baseline, stream reduces structure demand MPKI in all cases (71.3%, 58.9%, 21.5%, 39.7%, and 44.4% for CC, PR, BC, BFS, and SSSP, respectively), but it mostly fails in significantly reducing the property demand MPKI because of its inability to capture such difficult-to-predict large reuse distances. The only exception is BFS, in which the property MPKI is reduced by 23%, corroborating the earlier observation that stream is comparatively better at capturing property prefetches for most BFS benchmarks.

(2) Compared to stream, streamMPP1 significantly reduces the property MPKI for all algorithms (39.5%, 48.7%, 24.8%, 92.8%, and 41.4% for CC, PR, BC, BFS, and SSSP,

respectively), but it does not significantly reduce the structure MPKI. The property MPKI reduction comes from the MPP, which follows structure prefetches to bring in a high volume of useful property prefetches. Structure MPKI does not benefit because the streamers in stream and streamMPP1 are the same.

(3) Compared to streamMPP1, DROPLET further reduces structure demand MPKI (76.6%, 62.4%, 45.7%, 5.73%, and 41% for CC, PR, BC, BFS, and SSSP, respectively) because a structure-only streamer leads to a larger volume of structure prefetches by dedicating all the trackers to structure memory regions. Correspondingly, the property MPKI is also reduced since property prefetches accompany the structure prefetches (47.5%, 48.5%, 44.6%, 3.6%, and 39% for CC, PR, BC, BFS, and SSSP, respectively). However, we observe that the benefit for BFS is small compared to other algorithms. Hence, the small reduction in the LLC MPKI, together with a lower L2 cache performance, further explains why streamMPP1 rather than DROPLET is the ideal prefetcher for the BFS algorithm.

Prefetch accuracy: Fig. 4.7 shows the prefetch accuracies for the three prefetch configurations and the two data types. For structure data, DROPLET has the highest prefetch accuracy for all the algorithms. The accuracies are 100% for CC, 95% for PR, 53% for BC, 66% for BFS, and 64% for SSSP. For property data, DROPLET's prefetch accuracy is the highest for CC (94%), PR (95%), BC (46%), and SSSP (70%). For BFS, the property prefetch accuracy for DROPLET (47%) is lower than that of stream (70%). Overall, CC and PR have a higher prefetch accuracy for structure and property data than BC, BFS, and SSSP. This is because the former two algorithms process vertices in very sequential order. On the other hand, the BC, BFS, and SSSP algorithms depend on intermediate data structures, such as bins or worklists, when processing vertices. Consequently, the access pattern for structure data in BC, BFS, and SSSP consists of random starting points from which data can start being streamed.

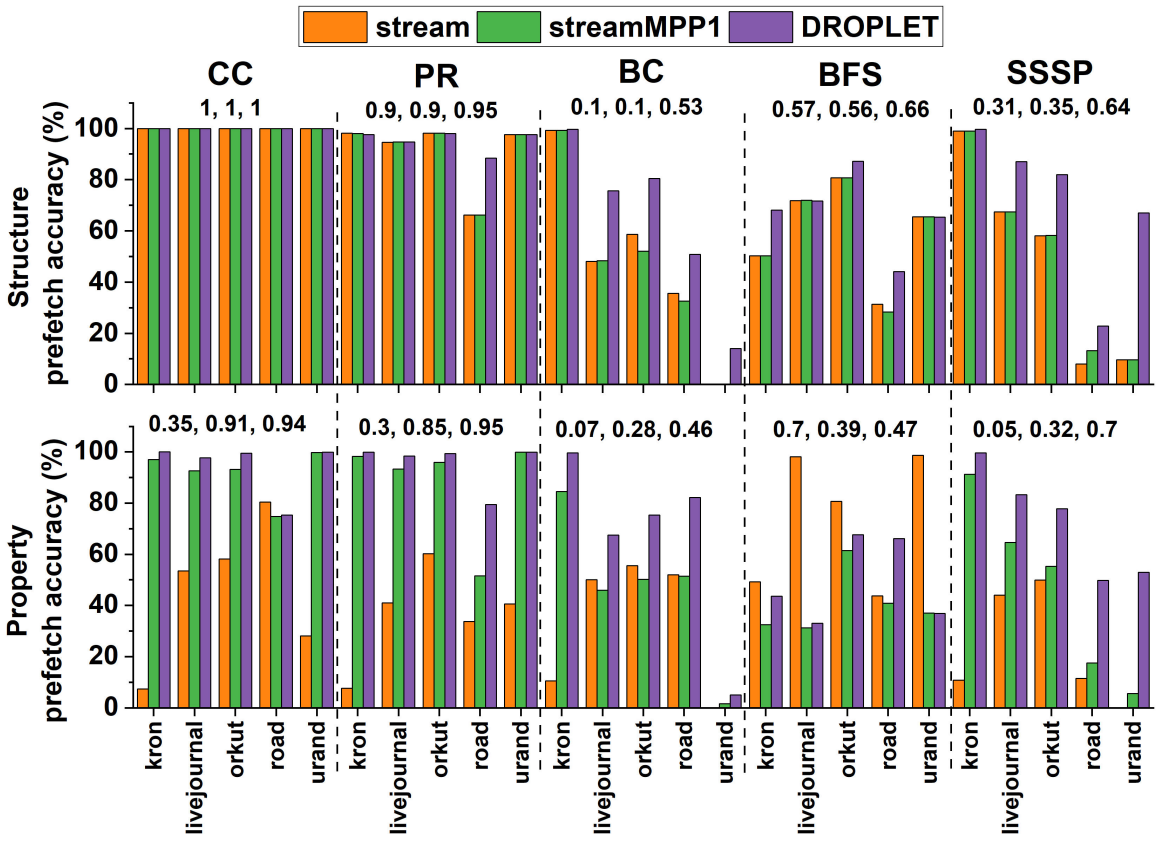


Figure 4.7: Prefetch accuracy. For each algorithm, the set of numbers shows the average for the three prefetch configurations across the five datasets.

Overheads of Prefetching

One of the side effects of prefetching is additional bandwidth consumption due to inaccurate prefetches, which may offset performance gains. DROPLET incurs low extra bandwidth consumption. Fig. 4.8 shows the extra bandwidth consumption for the three configurations measured in BPKI (bus accesses per kilo instructions). Compared to a no-prefetch baseline, DROPLET requires 6.5%, 7%, 11.3%, 19.9%, and 15.1% additional bandwidth for CC, PR, BC, BFS, and SSSP, respectively. The bandwidth requirement for CC and PR is smaller due to the comparatively higher structure and property prefetch accuracies.

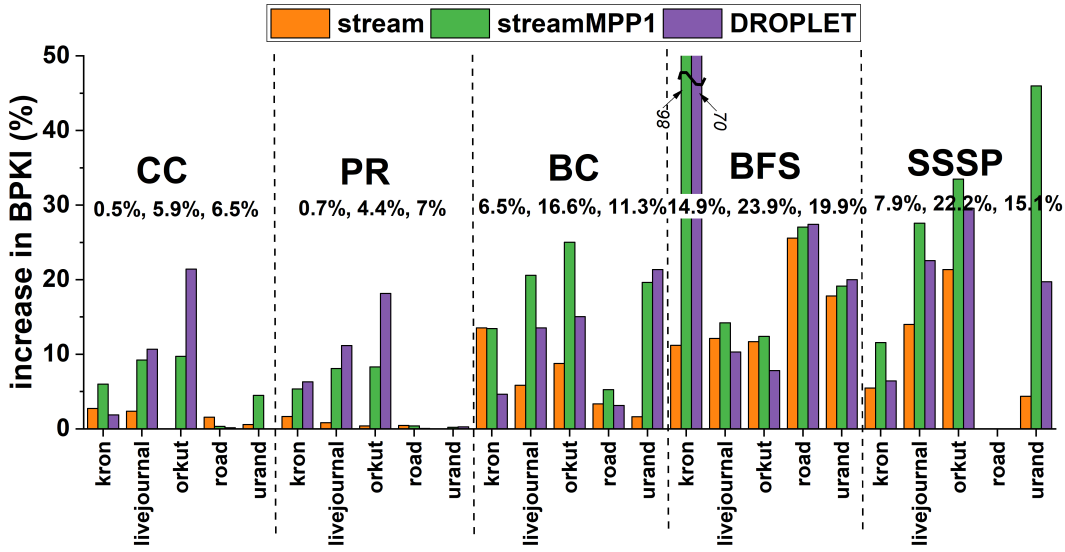


Figure 4.8: Additional off-chip bandwidth consumption. For each algorithm, the set of numbers shows the average for the three prefetch configurations across the five datasets.

4.4 Related Work

Graph characterization: Prior characterization on real hardware [9–13] provides insights such as high cache miss rates [11], under-utilization of memory bandwidth [9, 10], and limited instruction window size [9]. Prior work characterizes graph analytics workloads on Intel Xeon [9–12] or Intel Xeon Phi [13] and provides insights such as high cache miss rates [11], under-utilization of memory bandwidth [9, 10], and limited instruction window size [9]. Our profiling fills in the gaps in previous work through its data-aware feature and an explicit performance sensitivity analysis of the instruction window and the caches.

Prefetching: Although prefetching is a well-studied topic [23, 24, 96–105], our work shows that many state-of-the-art prefetchers are not adequate for graph processing (Section 4.3.2). Our contributions lie in the in-depth analysis of the data type behavior in graph analytics and in the observation-oriented prefetching design decisions (data-aware and decoupled prefetcher).

IMP [106] is a hardware-only L1 prefetcher for indirect memory references. Unlike IMP, we use the system support for data awareness to avoid long and complex prefetcher training. A data-aware prefetcher removes the need for IMP's strict and not-widely-applicable assumption of very long streams to be eventually able to train the prefetcher. In addition, we show that, for graph processing, a decoupled architecture provides performance benefits over a monolithic prefetcher (the design adopted by IMP) (Section 4.3.2).

Many prefetchers have been proposed for linked-list type data structures [96–99]. However, they prefetch at multiple levels of recursion (over-prefetch), whereas the structure-to-property indirect array access in graph analytics represents only one level of dependency. Runahead execution [102] uses a stalled ROB to execute ahead in the program path but can only prefetch independent cache misses. Dependence graph precomputation [103] is not data-aware and it requires large hardware resources to dynamically identify dependent load chains for precomputation.

Near-memory prefetchers and accelerators: There exist near-memory prefetchers that perform next-line prefetching from the DRAM row buffer [107] or prefetch linked-list data structures [97,108]. To the best of our knowledge, our work is the first to propose a data-aware prefetcher in the MC for indirect array accesses in graph analytics. In addition, our near-memory MPP is significantly different from previous approaches [97, 107, 108] in how it is guided by a data-aware L2 streamer.

Hashemi et al. [88] propose, for SPEC CPU2006 benchmarks, dynamically offloading dependent chains predicted to be cache misses to the MC for *acceleration*. However, in graph analytics, such a scheme could lead to high overheads from very frequent offloading since cache miss rates are much higher. Instead, we use the concept of issuing requests directly from the MC to decouple our prefetcher and to achieve aggressive and timely property *prefetching* (Section 4.2.1). Note that acceleration and prefetching are two orthogonal techniques, and can be combined to gain benefit from both.

4.5 Conclusion

We optimize the memory hierarchy for single-machine in-memory static graph analytics. Based on the workload characterization (Chapter 3), we propose a novel architecture design for an application-specific prefetcher called DROPLET, which is data-aware and prefetches graph structure and property data in a physically decoupled but functionally cooperative manner. The experimental results show that DROPLET can achieve significant improvement over various prior work.

Chapter 5

Broadening Graph Analytics Domain Knowledge with SAGA-Bench Performance Analysis Platform

5.1 Introduction and Contributions Overview

As discussed in Chapter 1, a large body of research in the computer architecture community focuses on static graph processing, whereas streaming graphs remain completely unexplored. Domain knowledge, contributed by some influential workload characterization and benchmarking efforts [9, 11, 17], has been key to driving research on domain-specific architectures for static graphs. However, the domain knowledge is insufficient in its existing state because it ignores the time-evolving nature of graphs. Streaming graphs are neglected in today's domain-specific architectures because of two issues: 1) immature software and lack of systematic performance analysis, and 2) absence of open-source benchmarks. Motivated by these shortcomings, we broaden the domain knowledge for the computer architecture community by 1) providing a performance analysis platform

and benchmark called SAGA-Bench for StreAming Graph Analytics; and 2) performing systematic workload characterization at both the software and the hardware levels on a state-of-the-art CPU server. Specifically, we present three contributions:

Contribution 1: Development of SAGA-Bench (Section 5.2). 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 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.

Contribution 2: Software-level Workload Characterization (Section 5.4). 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 [27]), whereas hash-based data structure is the most scalable for heavy-tailed graphs. The terms *heavy-tailed* and *short-tailed* are defined in Section 5.4.2 in the context of this work.

- 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 5.5). 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 [9, 10, 14, 15, 95, 109–122] focuses on static graphs and does not consider a detailed study of the graph update phase.

5.2 SAGA-Bench Description

SAGA-Bench is implemented in C++ and contains a collection of 4 data structures (Section 5.2.1), 2 compute models (Section 5.2.2), and 6 vertex-centric algorithms (Section 5.2.3) implemented in both the compute models[1].

[1]All implementations are done from scratch. Even when an open-source implementation is available (e.g., Stinger [27]), 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.

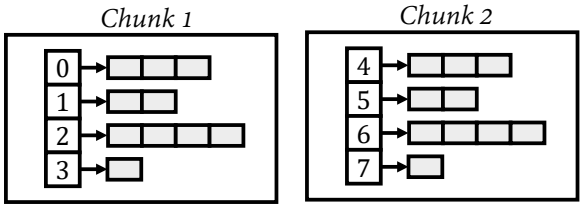


Figure 5.1: Chunked adjacency list (AC)

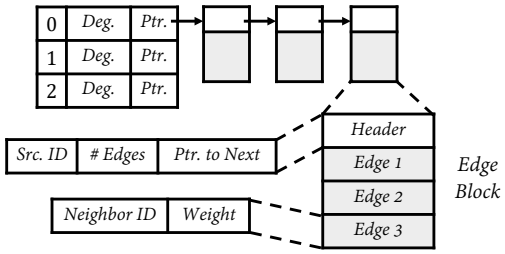


Figure 5.2: Stinger

5.2.1 Data Structures for Graph Topology

SAGA-Bench contains four vertex-centric data structures which support multithreaded edge update, as described below[2], for storing the graph *topology*[3]. As described in prior work [27,123], we implement each edge update only after a search operation so that edges are ingested uniquely.

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.

Adjacency List (chunked style multithreading) (AC): As shown in Fig. 5.1, 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

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

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

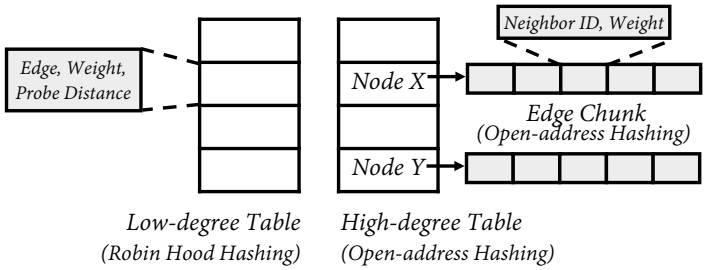


Figure 5.3: Degree-aware Hashing (DAH)

is the same as in AS). Update multithreading is achieved with multiple chunks.

Stinger: Stinger [27] is a shared-memory data structure where multithreading is achieved with OpenMP. As shown in Fig. 5.2, 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 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.

Degree-aware Hashing (DAH): As shown in Fig. 5.3, DAH [123] 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.

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.

5.2.2 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. 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 [17]) during each successive compute phase. We implement Max Computation and Single Source Widest Paths (Section 5.2.3) 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 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* [38, 124]: 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* [19, 30]: 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.

Table 5.1: Vertex functions for algorithms

Alg	Vertex function
BFS	$v.depth \leftarrow \min_{e \in InEdges(v)}(e.source.depth + 1)$ [38]
CC	$v.value \leftarrow \min(v.value, \min_{e \in Edges(v)} e.other.value)$ [124]
MC	$v.value \leftarrow \max(v.value, \max_{e \in InEdges(v)}(e.source.value))$ [38]
PR	$v.rank \leftarrow 0.15 + 0.85 * \sum_{e \in InEdges(v)} e.source.rank$ [124]
SSSP	$v.path \leftarrow \min_{e \in InEdges(v)}(e.source.path + e.weight)$ [38]
SSWP	$v.path \leftarrow \max_{e \in InEdges(v)}(min(e.source.path, e.weight))$ [38]

5.2.3 Algorithms

As summarized in Table 5.1, 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).

5.2.4 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.

5.3 Experimental Setup

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).

Methodology : SAGA-Bench is compiled with gcc-7.3.1. All experiments (except for studies on core scaling in Section 5.5) 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 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 [26–28, 37]).

All the experiments are repeated three times and each experiment provides batchCount (see Table 6.2) 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

Table 5.2: Evaluated datasets. BatchCount computed with batch size of 500K (see Section 5.3).

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

of the average.

Architecture-level profiling of memory, caches, and inter-socket bandwidth (Section 5.5) is performed with Intel Processor Counter Monitor (PCM) [125].

Datasets: The datasets in Table 6.2 are taken from SNAP [86], with the exception of synthetic RMAT [126] 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.

5.4 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.

5.4.1 Best Combination of Data Structure and Compute Model

Table 5.3 and 5.4 show, 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.

Table 5.3: For BFS, CC, and MC, the 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

Table 5.4: For PR, SSSP, and SSWP, the 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)**
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

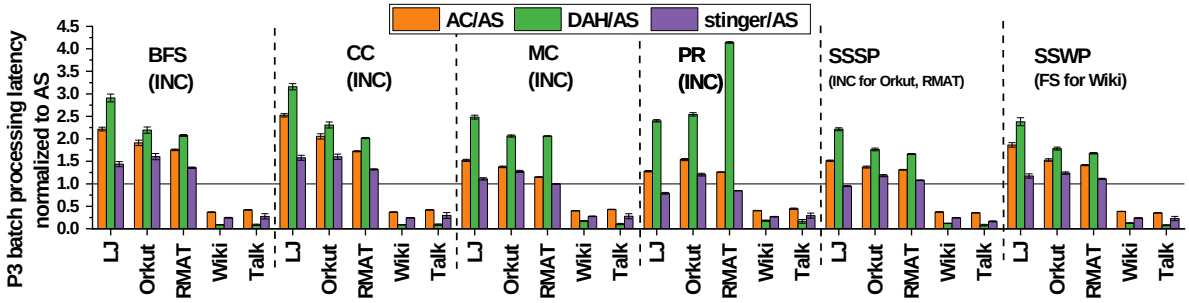
Best data structure. The best data structure depends on the dataset. AS and Stinger are the most competitive data structures over P1, P2, and P3 for LJ, Orkut, and RMAT. For Wiki and Talk, on the other hand, DAH consistently shows good scalability over time, i.e., DAH is the best data structure at P3. 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.

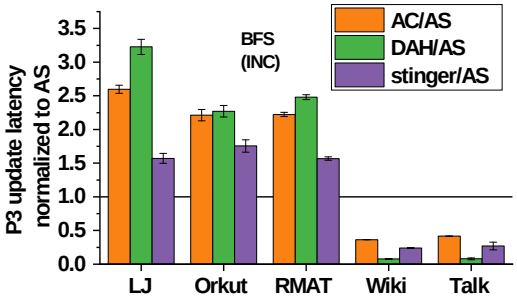
5.4.2 Impact of Data Structure

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.

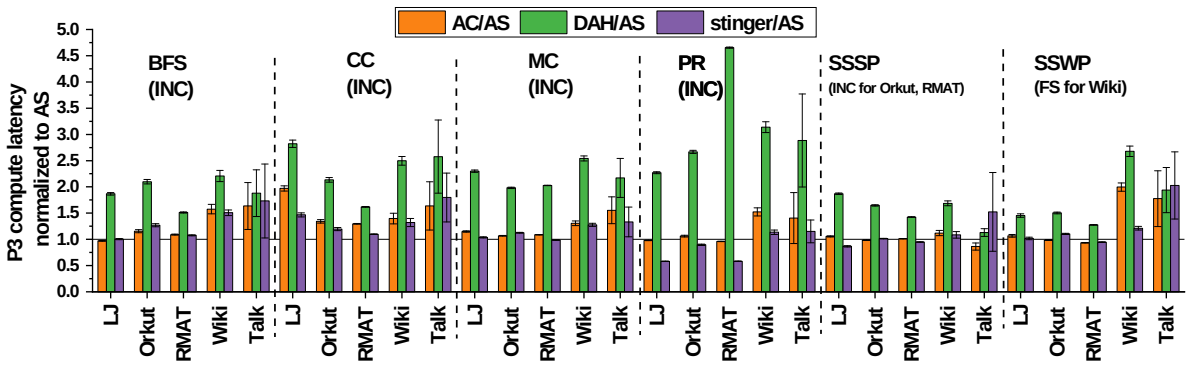
Fig. 5.4(a) shows, for each algorithm (at the best compute model) and dataset, the total batch processing latency of each data structure normalized to AS at P3. The most striking trend is the flipped relative performance benefits of AS and DAH for different datasets. For LJ, Orkut, and RMAT, AS (occasionally 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. 5.4(b) and 5.4(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,



(a) Batch processing latency



(b) Update latency



(c) Compute latency

Figure 5.4: (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 **??**). 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).

the update phase shows flipped behavior for Wiki and Talk. To understand the graph structural property which affects this behavior, we define **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[4]. As shown in Table 5.5, 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. 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. 5.4(b)) because the overhead due to its meta-operations 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 for both update and compute phases:

Update latency for short-tailed LJ, Orkut, and RMAT: Fig. 5.4(b) provides evidence of the following relative ordering of the four data structures for update latency

[4]Our definition is with respect to a batch because it directly impacts the streaming graph processing latency (Equation 2.1). 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 5.3). Table 5.5 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 5.5: 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**

(from highest to lowest): DAH > AC > Stinger > AS[5]. 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. 5.4(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-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.

[5]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.

Impact of data structures and their traversal mechanisms on compute latency: As shown in Fig. 5.4(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.

5.4.3 Impact of Compute Model

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

As shown in Fig. 5.5, 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[6].

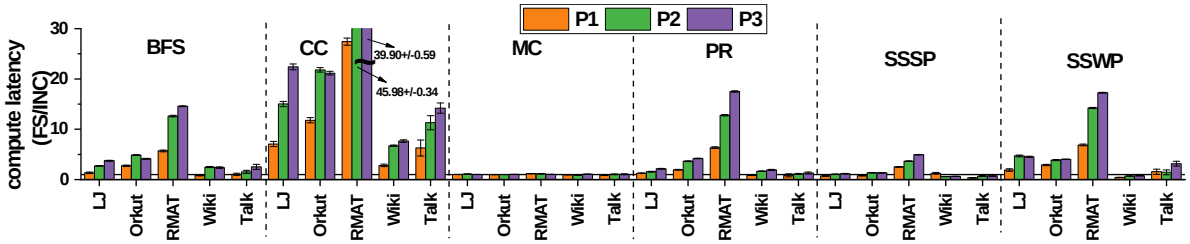


Figure 5.5: Compute latency of FS (normalized to INC) at the best data structure over three stages for all the algorithms. Experiments are performed at the best data structure to isolate the performance impact of only the compute model.

5.4.4 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. 5.6 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.

[6]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 [17] is highly optimized.

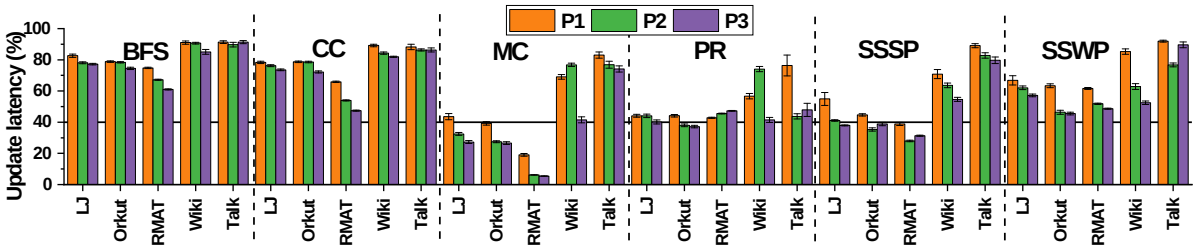


Figure 5.6: Percentage of batch processing latency occupied by the update phase over three stages. Experiments are performed at the combination of the best data structure and the best compute model to get the latency breakdown under the best conditions.

5.5 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 5.4). We use the incremental compute model (INC) for all the algorithms and categorize the results into two groups: 1) **STail**, i.e., average across short-tailed graphs LJ, Orkut, and RMAT on AS across six algorithms; and 2) **HTail**, i.e., average across heavy-tailed graphs Wiki and Talk on DAH across six algorithms.

5.5.1 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

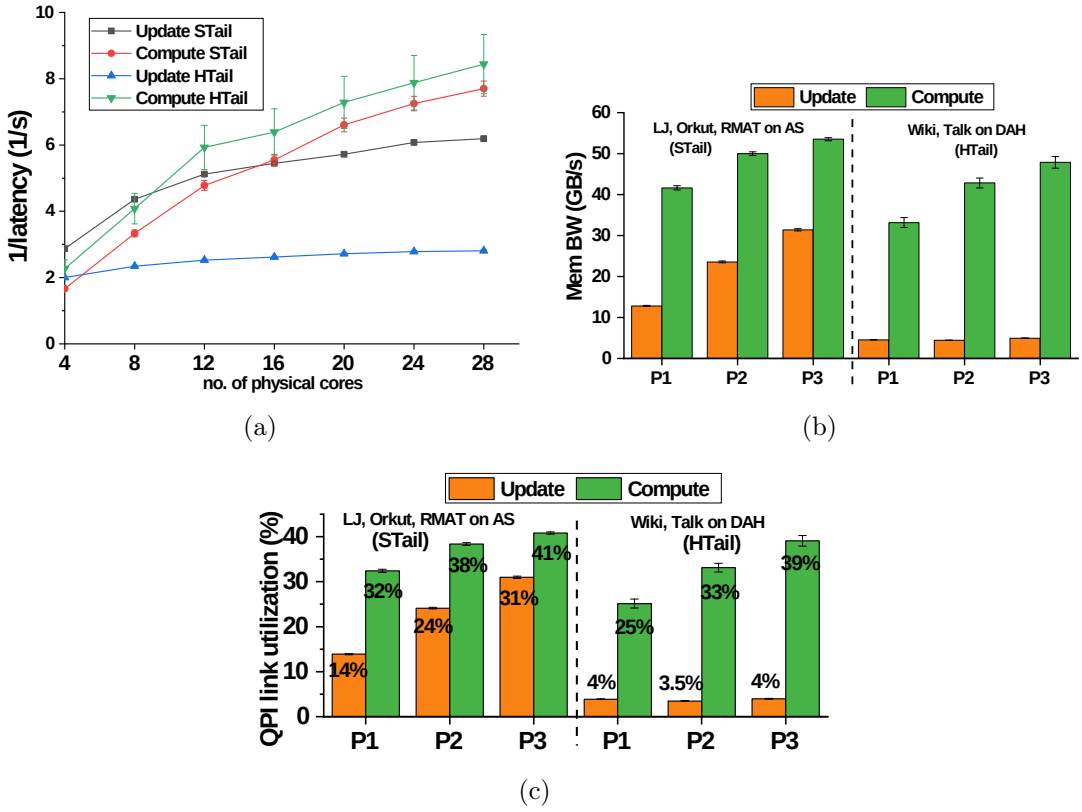


Figure 5.7: (a) STail/HTail update/compute performance scalability to physical core count (cores are distributed equally among 2 sockets at any given core count. Hardware execution threads=2 × number of physical cores), (b) memory bandwidth utilization, and (c) QPI link utilization.

update phase shows lower performance scalability to larger core counts. Fig. 5.7(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. 5.7(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. 5.7(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.

5.5.2 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 5.5.1, supports that the low TLP of the update phase arises from 1) thread contention in short-

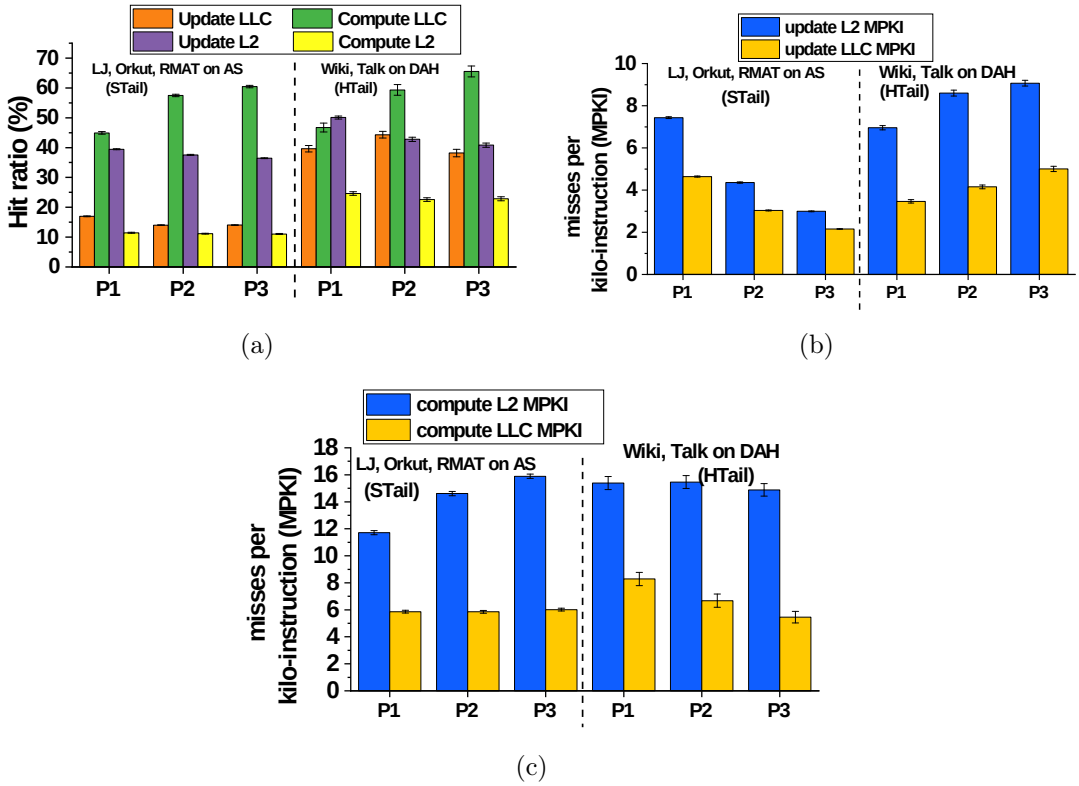


Figure 5.8: (a) Private L2 and shared LLC hit ratios. Private L2 and shared LLC MPKI for (b) update and (c) compute phases.

tailed graphs on AS and 2) workload imbalance in heavy-tailed graphs on DAH. This observation indicates that the parallelism bottleneck of the 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. 5.7(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. 5.7(b) and 5.7(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 Section 5.4.2. 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, 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 (Section 5.5.1). 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 a serious issue for STail because the datasets are not as heavy-tailed or highly imbalanced as HTail (Section 5.4.2).

5.5.3 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. 5.8(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 [121]).

Besides aggregate hit ratios, we measure misses per kilo instructions (MPKI) in Fig. 5.8(b) and 5.8(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. 5.8(b) compared to the compute L2 MPKI (12-16) in Fig. 5.8(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. 5.8(c)).

5.6 Conclusion

We develop SAGA-Bench for streaming graph analytics and characterize 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.

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: $Q_{next}.clear()$

Chapter 6

SPRING: Improving Streaming GraPh PRocessing Performance Using Input KNowledGe

6.1 Introduction and Contributions Overview

Streaming graph processing involves batched updates and analytics on graphs that evolve over time (Fig. 2.2(b), Fig. 2.3, Chapter 5). Effective handling of streaming graph data requires high-performance solutions for 1) update (ingestion of new edges contained in input batches), and 2) compute (analytics on the latest snapshot of the graph). Numerous competitive streaming graph systems have recently proposed novel data structures and computation models [7, 19–22, 25–38, 43, 123, 124, 127–131]. However, the shortcoming of existing systems is that they do not consider the issue of input sensitivity which is critical to optimize both update and compute performances. Input batches may exhibit variations in structural properties such as degree distributions of individual input batches or locality characteristics between consecutive input batches.

These diverse input properties give rise to challenging trade-offs in software performance which, if ignored, can lead to a substantially sub-optimal performance. It is possible to significantly optimize the system performance through a design approach where input knowledge-driven adaptive software and hardware solutions complement each other. For example, batch reordering (RO) is a software optimization which reorganizes an input batch to remove lock-based operations in streaming graph updates [25, 26]. The state-of-the-art input-oblivious RO improves the update performance for *wiki*'s input batches by 2.7× but severely degrades the update performance for *uk*'s input batches (0.69×) (Fig. 6.1(a) and 6.1(b)). Our proposed input-aware adaptive software recovers *uk*'s lost update performance (from 0.69× to 0.92×) by capturing RO's input-dependent performance trade-offs (Fig. 6.1(c)). Our proposed complementary input-aware hardware solution further increases *uk*'s update performance improvement to 1.60× (Fig. 6.1(d)), improving the overall system performance across both the workloads. Despite the potential gains, input knowledge-driven software and hardware design remains unexplored in state-of-the-art streaming graph system design approaches. Motivated by the shortcoming, we adopt the SPRING design approach, i.e., improving Streaming graPh pRocessing performance using Input kNowledGe. We provide input-guided adaptive software and hardware solutions which complement each other.

For efficient graph updates, we propose input-aware batch reordering with software and hardware dynamic execution modes (Section 6.3). We characterize RO across 260 workloads and find that its impact on the update performance depends on the degree distribution of the input batches. Our experiments show that RO performance varies from high speedups to significant degradations. This study motivates the need to adaptively reorder incoming batches based on their input characteristics. We propose adaptive batch reordering (ABR) which uses a low-overhead online technique to collect information on the degree distribution of incoming batches. Specifically, we propose the concept of *order-*

Figure 6.1: (a)/(b): Speedup in update performance from input-oblivious batch reordering for *wiki* and *uk* at input batch size=100K. (c)/(d): Input-aware software and hardware design recover and improve *uk*'s update performance. See Section 6.5.1 for benchmarks and evaluation setup.

λ *clusterable average degree (*CAD_λ*)* which is used by ABR to predict whether an input batch is suitable for batch reordering. Compared to a naive always-RO solution, ABR can save the update performance from degradation in reordering-adverse cases without compromising the high speedup of reordering-friendly cases.

We propose two additional case-specific optimizations which complement ABR during the update phase: software-level update search coalescing (USC) for reordering-friendly cases and hardware-accelerated update (HAU) for reordering-adverse cases. USC leverages the degree distribution and the reordered organization of reordering-friendly input batches to substantially reduce the amount of search operations during edge updates. Since reordering clusters the incoming edges of a vertex, it is possible to search for all the incoming edges in the current edge data of the vertex *in one go*. ABR and USC cooperatively provide effective software optimizations for the updates of reordering-friendly input batches.

HAU complements ABR during the update phase of input batches with reordering-adverse degree distributions. Although ABR successfully recovers the RO performance degradation for these cases, it is unable to provide any additional benefit over the baseline. ABR turns off the optimizations of batch reordering and associated USC because their

software overheads are expensive. Hence, reordering-adverse batches are still limited by 1) lock-based updates and 2) overheads of update search operations. HAU accelerates graph updates to remove these two bottlenecks. To remove lock-based software updates, HAU introduces enhancements to the cache controller and the on-chip processor-network interface to map each update task to a specific core. To mitigate search overheads, HAU uses simple dedicated logic in the cache controller to scan edge data cachelines, removing CPU instruction overhead for search operations. ABR and HAU cooperatively provide high-performance updates in reordering-adverse cases.

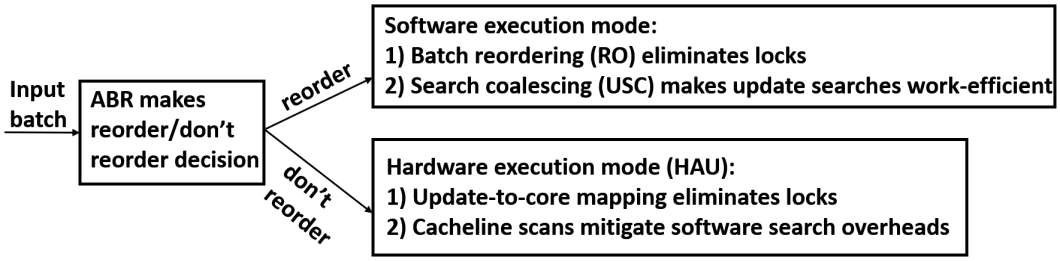


Figure 6.2: Input-aware SW/HW graph updates in SPRING

As summarized in Fig. 6.2, ABR, USC, and HAU together optimize the update performance by dynamically adopting the best suited execution mode based on the input batch characteristics. Input batches with reordering-friendly degree distributions (identified by ABR) are updated in the software execution mode with two optimizations: batch reordering and USC. In contrast, reordering-adverse input batches are updated with our architectural support, HAU. We quantitatively show that input-aware dynamic software/hardware execution for graph updates outperforms input-oblivious updates with either exclusively software or exclusively hardware execution mode.

For higher streaming graph computation performance, we propose input-aware work aggregation (Section 6.4). Consecutive input batches sometimes exhibit a large overlap in graph modifications (i.e., inter-batch locality). Scheduling two separate computation rounds/phases to analyze these high-overlap input batches leads to work redundancy. We

propose overlap-based compute aggregation (OCA) which adaptively modulates the computation granularity based on the inter-batch locality characteristics in input batches. OCA uses a low-cost online technique to identify inter-batch locality, and, for high locality, aggregates the computation. Thus, two input batches of similar graph modifications can be analyzed by scheduling a single computation round, increasing the compute efficiency.

Evaluation across 260 workloads shows that our SPRING approach provides significant improvements in update and compute performances in streaming graphs.

6.2 Novelty and Impact

We discuss the ways in which our overall approach and techniques advance the state-of-the-art prior work.

Streaming graph systems. We propose a novel perspective for efficient streaming graph processing: using input knowledge to optimize the performance for a given underlying data structure and computation model. This is orthogonal to the conventional approach [7, 19–22, 25–27, 27–38, 43, 123, 124, 127–131] of presenting a new system composed of input-oblivious data structures and/or computation models to outperform the state-of-the-art. We perform a novel characterization study across 260 workloads (Section 6.3.2) and show that input-oblivious software optimizations degrade performance in many cases. Our proposed input-dependent optimizations are applicable to most standard data structures and computation models.

Input-dependent graph processing. Regrettably, existing input-aware solutions focus on static graph processing [132–135] and are not readily reusable for streaming graphs. They 1) operate on *static input whole graph*, 2) typically make a decision once in the preprocessing step, and 3) only capture the trade-offs for the graph computation phase. In

contrast, the requirements for streaming graphs are substantially different. Our proposed ABR 1) operates on *input batches* whose size is much smaller than a whole graph, 2) makes online decisions multiple times during the execution on continuously incoming input batches, and 3) captures the performance trade-offs for the graph update phase (i.e., whether RO overhead pays off in terms of performance gains from lock-free updates). Hence, our designed ABR is an *extremely* lightweight (due to requirements 1 and 2) input-dependent algorithm with a novel and minimally intrusive CAD_λ metric which captures the update phase's performance trade-offs. Furthermore, another novelty in our input-dependent approach is using input batch characteristics to dynamically decide *between software and hardware* execution modes to achieve a higher update performance than a SW-only or a HW-only solution.

Hardware support for graph processing. Prior domain-specialized solutions for graph analytics are valuable but restricted to static graph computation [14, 15, 95, 111–118, 120–122, 136–138]. Although a few papers [122, 136] provide APIs for graph updates, the discussion is limited to a subsidiary feature for a design targeted at static graphs and does not address the specific challenges for streaming updates. In contrast to prior proposals, we focus in-depth on the dynamic nature which is indispensable in real-world graphs. We tackle unique acceleration considerations which arise due to the characteristics in streaming graph updates. First, by treating input properties as the first-class design determinant, we provide insights that hardware acceleration is beneficial and meaningful for reordering-adverse input batches where software overheads are too high. Second, we selectively trigger HAU for these input batches to resolve the relevant challenges (overheads of software locks and search operations). Concerning HAU's specific design techniques (Section 6.3.5), HAU's concept of "task" bears some resemblance to GraphPulse's event-driven approach [138]. Although an important work for static graph computation, GraphPulse is not a drop-in replacement for HAU because 1)

its events cannot represent and process streaming graph updates, 2) its design cannot solve update search overheads, and 3) it is a fully customized stand-alone ASIC, whereas HAU acceleration is CPU-coupled where the introduced changes are aware of the CPU architecture.

Search and computation in streaming graphs. *USC*: The novelty of our input-guided USC for reordering-friendly batches is that it resolves update search overheads by opportunistically leveraging the reordered data organization. It is low-overhead and applicable on simple data structures (e.g., adjacency lists). In contrast, prior work has proposed new data structures to solve this problem often at the cost of high additional memory consumption [130] or complex data structures [26]. *OCA*: We propose online modulation of *computation granularity* in response to the degree of overlap between consecutive *input batches*. Our approach is orthogonal to that of prior work [19, 34, 36, 37, 43, 127, 129]. The latter focuses on *incremental computation models* to address the overlap in the *latest snapshot of the graph data structure*.

6.3 Input-Aware Streaming Graph Updates

We first provide some background on batch reordering (RO) and present our characterization study of the performance trade-offs of RO. After explaining ABR, USC, and HAU, we discuss input-aware SW/HW dynamic execution.

6.3.1 Batch Reordering (RO) Basics

RO is a pre-update operation where the input batch of edges is reorganized to ensure lock-free edge updates [25, 26]. In the baseline input batch, during parallel edge updates, two separate threads may update edges for the same vertex, requiring locks to protect against shared memory access conflict. In contrast, in the reordered input batch, the

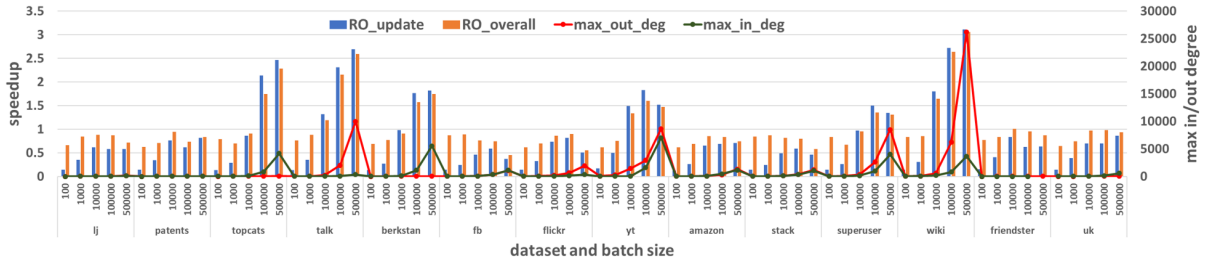


Figure 6.3: Left y-axis: Effect of RO on update and overall (i.e., update and compute combined) performances (see Section 6.5.1 for the evaluation methodology). Right y-axis: Maximum in/out degree in an input batch (average across all batches).

edges of the same vertex v are clustered (parallel stable sort from C++ Boost library [139]) and a carefully designed work division ensures that a specific thread updates all the input edges of v (dynamic OpenMP scheduling ensures load balancing). Batch reordering does not violate the ordering of updates because for a given vertex v, stable sort ensures that the updates for v are ordered. In addition, the updates for two distinct vertices v and w can be performed in any order. In other words, ordering must be guaranteed for a given vertex (achieved through stable sort) but is not necessary between two distinct vertices. In case an input batch contains both insertions and deletions, the two categories can be separated and each sub-batch can undergo batch reordering based update, with the insertions being applied before the deletions.

The baseline (non-reordered) and RO-based update methodologies possess different benefits and costs, giving rise to RO's input-dependent performance trade-offs. *Baseline*: The benefit of the baseline is that it offers fine-grained edge-level parallelism and requires no change to the input batch format. The edge-centric work division is in perfect alignment with the input batch format. Incoming graph changes arrive as edges and the baseline treats the edge as the granularity of parallelism by assigning one thread per edge. However, the baseline's cost constitutes lock operations because separate threads may update edges for the same vertex. *RO*: RO's benefit involves completely eliminating locks by adopting vertex-centric updates (i.e., a thread updates all the edges of a given

vertex). However, this comes at the cost of software overheads because the input batch format is inherently organized as edges instead of in a vertex-centric fashion. First, the input batch must be sorted to cluster edges belonging to the same vertex. Second, sorting should be carried out with respect to both source vertices and destination vertices to account for both in-edges and out-edges. This results in two reordered input batches which must each be updated separately. Finally, lock elimination involves additional scheduling overheads (scheduling must ensure that each thread updates all the edges belonging to a given vertex before moving on to another vertex in its task list). Our characterization in Section 6.3.2 shows how input batches of different degree distributions are affected by the relative costs and benefits of the two update methodologies.

6.3.2 Characterization of RO Performance Trade-offs

Our characterization across 260 workloads shows that the *performance from RO exhibits input sensitivity* (Fig. 6.3, left y-axis). Datasets like *topcats*, *talk*, *berkstan*, *yt*, *superuser*, and *wiki* indeed achieve up to about 3× improvement in update and overall performances at higher batch sizes of 100K and 500K (also at 10K for *talk*, *yt*, and *wiki*). However, at smaller batch sizes of 100 and 1K, these datasets experience degradations in update and overall performances. The remaining datasets (*lj*, *patents*, *fb*, *flickr*, *amazon*, *stack*, *friendster*, and *uk*) experience performance degradation from RO at all batch sizes.

We observe that *high-degree input batches are reordering-friendly, whereas low-degree input batches are reordering-adverse.* We define "high (low)-degree input batch" as an input batch where the *top degrees* are high (low). For example, Fig. 6.3.2 shows the degree distributions of representative input batches of *lj* and *wiki* at batch size of 100K. *Lj*'s input batch is low-degree (e.g., top ten degrees lie in the range of 7-30, 30 being the maximum degree), whereas *wiki*'s input batch is high-degree (e.g., top ten degrees lie in

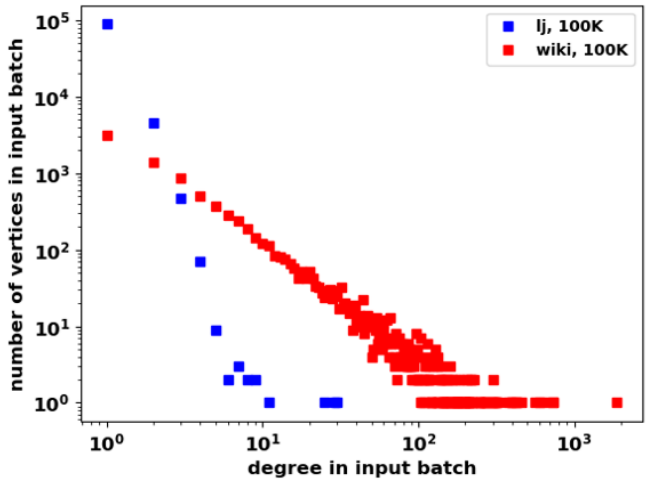


Figure 6.4: Input batch degree distributions of *lj* and *wiki* at batch size 100K (log-log plot)

the range of 401-1881, 1881 being the maximum degree)[1]. Since the maximum degree represents the upper bound of an input batch's top degrees, we use it as an indicator metric in Fig. 6.3 (right y-axis) to show the correlation with the RO performance (left y-axis). Compared to the reordering-adverse cases, the reordering-friendly cases exhibit a higher maximum in-degree or out-degree, indicating a high-degree input batch. For a given dataset, a smaller batch size naturally leads to a low-degree input batch (maximum possible degree = batch size). Therefore, small batches suffer from performance degradation when RO is applied (Fig. 6.3).

The performance trade-offs of RO can be understood by connecting the degree distribution of the input batches to the relative costs and benefits of RO (Section 6.3.1). High-degree input batches are reordering-friendly because:

- In the baseline, a large number of locks needs to be acquired to update a top-degree vertex v. High-degree batch means v possesses a very high edge count. Updating each

[1]**Terminology clarification:** We consistently use the term *top degree* to refer to an intra-input-batch large degree. In contrast, the term *high/low-degree* is used to differentiate between the top degrees across different input batches. Thus, a top-degree vertex in a high-degree input batch has a larger edge count than a top-degree vertex in a low-degree input batch.

incoming edge of v needs a lock to be acquired on v's edge data because multiple threads may update v's incoming edges.

- In the baseline, in addition to the large number of locks described above, the cost of acquiring a lock is high for v. The cost of a lock acquisition involves waiting for another thread to finish updating an incoming edge for v. The wait time is proportional to the length of v's edge data array because updating involves a search scan for duplicate check (Section 6.3.4). The length of v's edge data array is large because v is a top-degree vertex in a high-degree input batch.

The above two factors together lead to high lock overheads for high-degree input batches in the baseline scheme. RO can eliminate these serious lock overheads. The cost of RO is small compared to the savings from baseline's lock overheads. In contrast, low-degree input batches are reordering-adverse because lock overheads in the baseline technique are not serious (i.e., top degrees are relatively small) and RO's extra software overheads are larger than the potential savings.

In addition to the above performance trade-offs, we find *temporal stability in the input batch degree distribution for a given dataset and batch size combination*. As shown with an example in Fig. 6.5 for *lj* at input batch size of 100K, the input batches consistently show much alike degree distribution over time with increasing batch numbers. However, across different dataset-batch size combinations (*lj*-100K versus *wiki*-100K), the degree distribution is clearly different (Fig. 6.3.2). In the next section, we discuss how these insights are used to design a low-cost and effective adaptive batch reordering technique.

6.3.3 Adaptive Batch Reordering (ABR)

ABR is an online technique that adaptively reorders input batches depending on their degree distributions. As shown in Fig. 6.6, ABR instruments the update phase of every

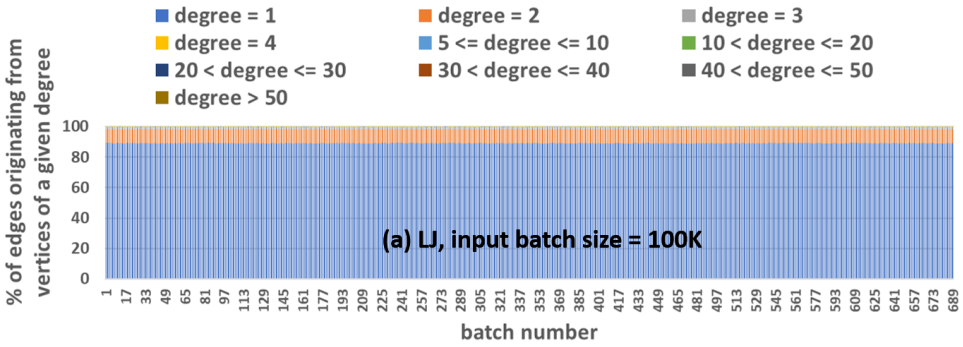


Figure 6.5: Input batch degree distribution over time

n^{th} input batch (called ABR-active batch) to collect information on the input batch's degree distribution. Using this information, ABR makes a binary decision (reorder/don't reorder) and, for the next n update batches (called ABR-inert batch), the latest ABR decision is applied. ABR has a low overhead because of:

- Small number of ABR-active batches: ABR leverages the temporal stability of input batch degree distribution (Section 6.3.2) to reduce the amount of instrumentation. A reordering decision made by observing one ABR-active batch can be applied to multiple subsequent ABR-inert batches.

- Low-cost degree distribution collection in ABR-active batches: In the ABR-active batches, instrumentation is overlapped with the actual edge updates. Moreover, we propose a practical, low-cost, and minimally intrusive metric for instrumentation (explained below).

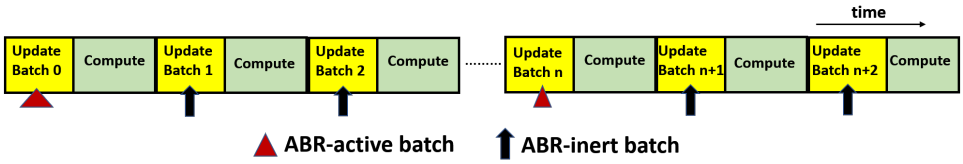


Figure 6.6: Adaptive Batch Reordering (ABR) design

We propose a metric called ***order-λ clusterable average degree (CAD_λ)*** which is computed by ABR during instrumentation in the ABR-active batches to make accurate reordering decisions with small overhead. We describe the metric below:

$$order - \lambda \text{ clusterable average degree } (\text{CAD}_\lambda) = \frac{b - y}{x}$$

where,

$b =$ input batch size

$y =$ number of edges from vertices with $1 \leq$ degree $\leq \lambda$

$x =$ number of unique vertices with degree $> \lambda$

If $\text{CAD}_\lambda \geq TH$, reorder. Else don't reorder.

$TH =$ some experimentally determined threshold.

CAD_λ intuition. CAD_λ is a measure of the *average degree of the top-degree vertices in an input batch* (i.e., the average degree computed by focusing on intra-batch vertices with large edge counts). A high CAD_λ for an input batch indicates a high-degree input batch which is essential to achieve performance benefit from RO (Section 6.3.2). ABR decides to reorder if CAD_λ is greater than or equal to some experimentally determined threshold (TH). The design parameters of ABR are n (determines the instrumentation frequency), λ (distinguishes the top-degree vertices in the input batch), and TH (distinguishes between high CAD_λ and low CAD_λ). λ parameter is a cutoff applied to locate an individual input batch's top degrees. In contrast, TH parameter is used to understand the relative values of the top degrees across different input batches. Section 6.5.2 shows 1) how the parameter values are determined, and 2) the high decision-making accuracy of CAD_λ.

CAD_λ measurement techniques. In an ABR-active batch, the counting of y and

x for CAD_λ metric occurs as edges are updated. This overlapping operation reduces the instrumentation overhead. The method to collect y and x depends on whether the ABR-active batch has batch reordering turned on or off. When reordering is turned on, a thread keeps track of the total number of edges it updates for a given vertex v, and, based on that value, atomically increments one of the two global variables x and y. On the other hand, when reordering is turned off, a concurrent hash map implemented using Intel TBB [140] (key: vertex ID; value: degree) is populated during edge updates (multiple threads may update edges for the same vertex). Upon update completion, a parallel iteration over the hash map provides the values of y and x. When the updates are complete, ABR calculates CAD_λ and makes a reordering decision by comparing it with TH.

Choice of CAD_λ and general applicability. CAD_λ fulfills the two essential requirements for a good metric: 1) high decision accuracy and 2) low overhead (see Section 6.5.2). We also considered other alternatives. For example, *average degree* exhibits poor decision-making accuracy. It is always a consistently low value because most vertices in an input batch possess low degrees. This obscures the distinction between high-degree/low-degree input batch. Rigorous mathematical measures of skewness and heavy-tailedness have been proposed in areas of network and probability theory [141–143]. However, they are computationally heavyweight for our streaming graphs scenario where measurements needs to be made online and multiple times. The lightweight and accurate CAD_λ is a better choice for practical system design where performance is a key metric. Moreover, these proposed measures [141–143] have been applied to large-scale whole graph instead of to input batches. A rigorous statistical analysis of their applicability to input batches is beyond the scope of this work. In addition to being accurate, practical, and low-overhead, CAD_λ is widely applicable. It has been developed by observing the examples of thousands of input batches from our large evaluation suite.

6.3.4 Update Search Coalescing (USC)

USC complements ABR in reordering-friendly cases to reduce update search overheads during duplicate checking. Duplicate checking is a common procedure in graph updates. Before updating an incoming edge $A \rightarrow B$, a search through the edge data of A checks for B so that B is not duplicated (the edge may have appeared in an earlier batch or may have already appeared earlier in the current batch). We identify that a high-degree input batch reordered by ABR provides the opportunity to substantially reduce the number of search scans for duplicates through search coalescing. Since a given thread updates all the input edges of a given vertex A, it is possible to search for *all of A's incoming target vertices during a single scan* of A's edge data. The effectiveness of USC depends on the underlying highly clusterable degree distribution (i.e., very high top degrees in high-degree batches). The higher the clusterability of the input batch, the higher the scope of search savings through search coalescing (see below and Section 6.5.2). These high-degree input batches are also reordering-friendly (hence reordered by ABR) and USC conveniently leverages their reordered data organization.

Fig. 6.7 shows the implementation of USC taking the example of updating three edges for source vertex A. 1 As a thread walks through the chunk of the reordered/sorted input batch consisting of the edges of A, it populates a small hash table with A's targets and weights (Section 6.5.2 shows that this incurs very small overheads). 2 Edge data for A is scanned only once. Each neighbor ID in the edge data array is searched for in the hash table with the ID as the key. A positive match leads to updating the weight only (if weighted graph) and the specific target's entry is deleted from the hash table. Once the scan is complete, the remaining $<$*target*, *weight*$>$ in the hash table are inserted into A's edge data array (either in some empty spot identified during the scan or appended to the end of the array). In contrast, a non-reordered batch requires three separate scans

through the edge data of A because multiple threads update the edges. Therefore, the higher the clusterability (i.e., per-vertex edges), the higher the benefits of USC. Note that USC does not impact the granularity or amount of parallelism with respect to batch reordering. The contribution of USC is that it saves search-related work for individual threads participating in the parallel update process.

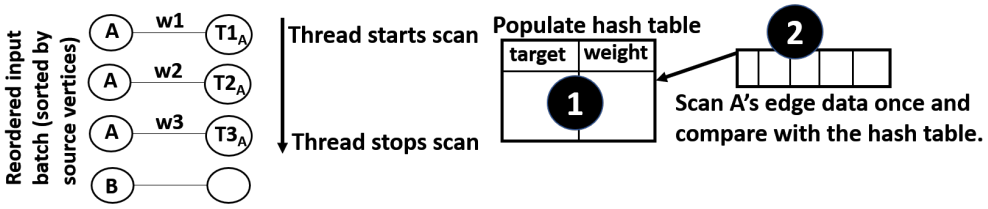


Figure 6.7: Overview of update search coalescing (USC)

6.3.5 Hardware-Accelerated Update (HAU)

HAU provides architectural support for the update of reordering-adverse input batches. Although their RO performance degradation is successfully recovered by ABR, they are still limited by 1) lock-based updates and 2) update search overheads. To resolve the former, HAU assigns each incoming update to a specific core. To resolve the latter, HAU uses specialized logic in the cache controller to scan returning cachelines, removing CPU instruction overhead for searches.

Design overview

(Fig. 6.8). The task-producing core triggers the HAU by feeding update tasks from the software 1. An update task for an incoming edge <*src, target*> takes the form of <*src's edge data start address, src's current degree, target*>. It is routed through the network-on-chip (NOC) to a task-consuming core 2[2] obtained by *src* mod N where N is

[2]1) A core can be both task-producing and task-consuming. Fig. 6.8 illustrates a decoupled behavior only for clarity. 2) For weighted graphs, HAU includes an extra field *weight* in the update task.

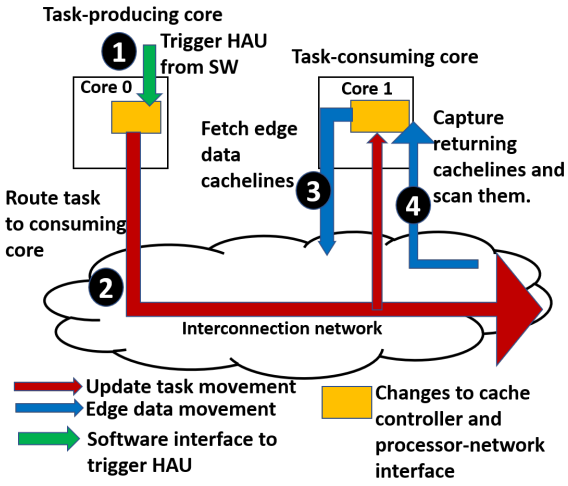


Figure 6.8: HAU overview

the number of task-consuming cores (Section 6.3.5 discusses this algorithm). The task-consuming core's cache controller uses the task description received from the NOC to fetch the edge data cachelines 3. Upon each cacheline's return to the L1D cache, the cache controller captures and searches it for *target* (duplicate check) using its dedicated scan logic 4.

For example, we consider a toy graph in memory consisting of vertices *V0*, *V1*, *V2*, and *V3*. We also consider a small input batch of size 1 consisting of the incoming edge *V1->V2*. The update task *<V1's edge data start address, V1's current degree, V2>* is assigned to core 1 (*V1* mod 2). Core 1's cache controller fetches *V1*'s edge data cachelines, searches for the target *V2* before updating the edge. A similar set of operations needs to be performed to update *V2*'s edge list to include *V1*.

Design details

We take the reference network interface in [144] and highlight our enhancements (Fig. 6.9/6.10).

Task production (Fig. 6.9): Driven by the software, the core initiates a request of

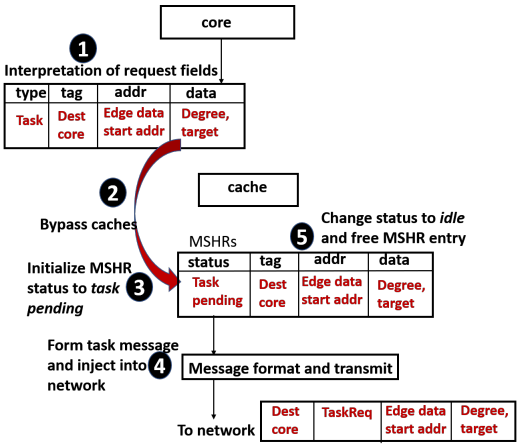


Figure 6.9: Task production

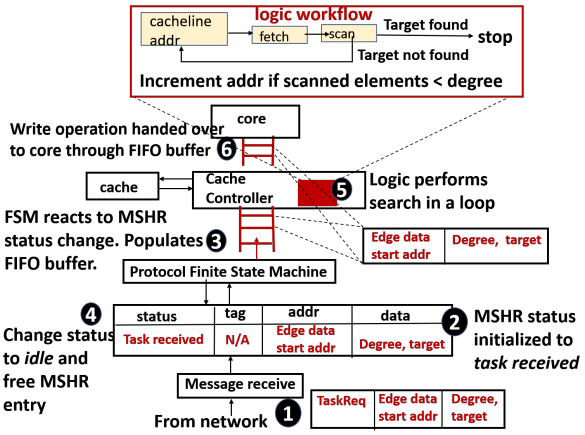


Figure 6.10: Task consumption

a new type called *task* 1 (already existing types are *read*, *write*, etc.). The address corresponding to this request is the start address of the edge data of vertex *src* and the data fields encode *src*'s current degree and *target*. We use the *tag* field to encode the destination core ID (in a conventional request, this field helps the core identify the corresponding reply). Unlike a *read* or *write* request, the address field of a *task* request does not mean this core expects data from this address. It only encodes some information, together with the data field, that needs to be sent to another core. The control flow for a *task* request involves bypassing caches 2 and initializing a new miss status handling register (MSHR) entry with a new type of status called *task pending* 3. Allocating an MSHR entry is essential because the *message transmit unit* only reacts to MSHR status changes [144]. The *message transmit unit* formats the NOC message of the update task and injects it into the network 4. The MSHR entry status is changed to *idle* and it is freed 5.

Task consumption (Fig. 6.10): Upon receiving a *TaskReq* message from the NOC at the message receive unit 1, a new MSHR entry is allocated with the status *task received* 2. The protocol FSM takes appropriate actions on the MSHR status change: the task is forwarded to a FIFO buffer to the cache controller 3 and the MSHR entry is freed 4.

Simple dedicated logic in the controller 5 uses the *edge data start address* to fetch the edge data cachelines. Each returning cacheline to the L1D cache is scanned to check for the *target* node. If found, the loop stops. Otherwise, the controller brings in consecutive cachelines until the number of scanned elements matches the *degree*. If the *target* is not found even after the entire edge data has been exhausted, the controller hands over the write operation to the core through the FIFO buffer 6. The core takes over this action because new memory region may need to be allocated to accommodate the *target*.

Discussion

We discuss HAU's important details.

Task assignment: Hashing-based task scheduling is very low-cost and requires no tracking overhead of scheduling history or progress status of large core counts in a modern multicore architecture. Moreover, this scheduling ensures that all incoming edges for vertex v are updated at the same core where v's edge data resides, implicitly guaranteeing safety against race conditions from concurrent accesses (allowing us to eliminate software lock overheads). A more complex assignment would require expensive design to explicitly guarantee race-safe accesses (e.g., GraphPulse [138] requires additional cycles to coalesce events for identical vertex in large hierarchical queues). Minimizing design complexity and scheduling overheads is critical because, in contrast to static whole graph processing [138], acceleration granularity for dynamic graph updates is a much smaller input batch, making the design constraints tighter. Section 6.5.2 discusses workload distribution.

Interaction with SW: To achieve the interaction between software and HAU, we adopt the technique used in previous work [14] where low-level software API methods translate to two specific instructions. On the task-sending side, the core uses a `supply_task` instruction to communicate the information related to the update task to HAU. On the task-receiving side, the core uses a `fetch_task` instruction to get the information from

the FIFO buffer.

Virtual memory: The cache controller logic requires virtual-to-physical address translation for the address it obtains from the task description in the FIFO buffer. We adopt the approach of previous work [14, 15] to ensure this. The cache controller logic shares the core's address translation machinery and handles page faults like [14].

Coherence protocol: HAU does not affect the cache coherence protocols. An update *task* request is a cache-bypassing point-to-point push-style communication between well-defined sender and destination cores and does not involve any additional coherence messages (Fig. 6.9/6.10). To process the *task*, changes are confined to the cache controller after edge data cachelines return through traditional coherence protocols.

Update ordering: HAU maintains the same consistency as software graph updates because, in the execution model we consider, the programmer expects that consistency is guaranteed at the granularity of an input batch. In a batch, individual updates (i.e., incoming edges) for vertex v may arrive and be processed at task-consuming core c in any order. The final result (i.e., at the end of the update phase for this input batch) is the same (and equivalent to software updates) because all the updates show up in v's edge data (the order of showing up does not matter). To maintain consistency in case the input batch contains both edge insertions and deletions, software triggers HAU to perform all insertions first before performing deletions (deletion requires that the edge already exists). Since tasks are independent, this update ordering policy ensures that updates are deadlock-free, i.e., no circular dependencies exist between any subset of tasks.

MSHR management: Keeping the baseline NOC topology, routing, and buffering unchanged, we introduce a new request/response type and show how it fits into the reference processor-network interface [144]. We add ten new MSHR entries (2× increase) reserved for outgoing/incoming *tasks* to avoid MSHRs becoming a performance bottleneck. *Task*-MSHRs are proactively freed, making space for new tasks. A *task pending* MSHR is freed

as soon as the *task* is released into the network (Fig. 6.9 4, 5). A *task received* MSHR is freed as soon as the FIFO buffer is populated (Fig. 6.10 3, 4). The total volume of *task* traffic is limited to input batch size, which is much smaller than a whole graph (Fig. 6.3).

Hardware overhead: HAU incurs small hardware overhead. Using McPAT integrated with Sniper [145], the area of the baseline chip (Table 6.1) is 212 mm^2 in a 22nm technology node. We implement the cache controller logic in RTL and synthesize it with Synopsys Design Compiler. We obtain an area of 0.0058 mm^2, leading to an overhead of ~0.044%. Each FIFO buffer entry consists of four 64-bit fields (fourth field is *weight* to account for weighted graphs). Ten new MSHR entries and two 32-entry FIFO buffers lead to an additional 1KB and 2KB storage per core tile, respectively.

Generality: Since graph processing is an important application domain, domain specialization for higher performance and efficiency is a common approach in previous work on static graphs [14,15,121,136,146,147]. HAU follows the similar approach of specialization to handle the *more general* case of dynamic graphs which is important but remains unexplored in prior proposals.

6.3.6 Input-Aware SW/HW Dynamic Execution

To address the software performance trade-offs arising from input sensitivity (Section 6.3.2), we adopt input-aware SW/HW dynamic execution for optimized performance and efficiency across all input types (experiments in Section 6.5.2). A SW-only approach (i.e., RO+USC) is sub-optimal for low-degree input batches because the SW overheads are higher than the performance gains. Without RO and USC, low-degree batches are still bottlenecked by software locks and search. HAU removes these bottlenecks and further improves graph update performance for low-degree batches. A HW-only approach (HAU)

is sub-optimal for high-degree batches because HAU design is sophisticated only enough for low-degree batches, minimizing its hardware overhead and functional complexity. For example, HAU does not support search coalescing because it is not necessary (e.g., *lj's* input batches mostly contain of degree=1 vertices (Fig. 6.5), making search coalescing superfluous and a source of inefficiency). Adding more functionality to HAU is possible but only increases engineering effort, design complexity, and overhead when effective software solutions are realizable.

6.4 Input-Aware Streaming Graph Computation

Input-aware computation aggregation adaptively modulates the streaming computation granularity during the runtime based on the locality characteristics between consecutive input batches. Fig. 6.11 explains how it differs from the baseline computation workflow. In the latter (Fig. 6.11 (a)), update and streaming computation are interleaved like numerous previous streaming graph systems [19,34,36,37,43,127,129]. Once a batch of updates are applied to the graph, an algorithm re-executes on the latest snapshot of the graph to reflect the changed data structure. Thus, the update batch size indicates the streaming computation granularity because the computation considers the changes to the graph data structure caused directly and indirectly by an amount of modifications equal to the batch size. On the other hand, the proposed input-aware computation aggregation (Fig. 6.11 (b)) uses overlap-based compute aggregation (OCA) technique to adaptively increase the computation granularity when there is high inter-batch locality, i.e., high overlap between the graph modifications contained in batches n and $n+1$. OCA increases compute efficiency for high inter-batch locality because TC_{agg} is less than $TC_{\text{n}} + TC_{\text{n+1}}$, i.e., aggregating computation helps amortize the scheduling and data access overheads of launching two separate computation rounds.

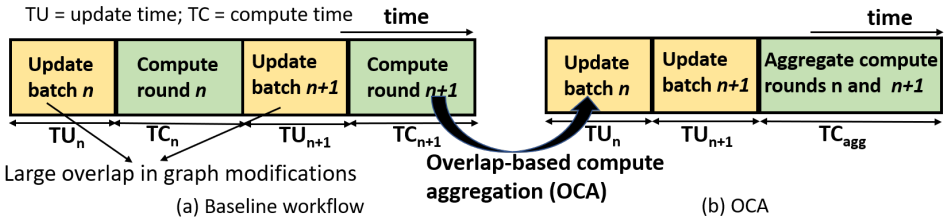


Figure 6.11: Overview of OCA

Design details. We classify the inter-batch locality between batches n and $n+1$ to be high when a large percentage of the unique vertices in batch $n+1$ also appeared in batch n (i.e., edge updates affect a lot of identical vertices across the two batches). This is reasonable because incremental computation models concentrate computation at or around the affected vertices. So, identical affected vertices across two input batches means that consecutive computation rounds touch similar regions of the graph. Scheduling two separate computation rounds to perform operations on similar regions of the graph leads to work redundancy in scheduling and data accesses. Computation aggregation eliminates this redundancy with a single aggregated round. We implement a low-cost online mechanism for measuring inter-batch locality. The graph representation is augmented with an additional per-vertex field `latest_bid` which tracks the last batch where a vertex appeared. This field is updated along with edge updates during each update phase. During an ABR-active batch (Section 6.3.3) (batch $n+1$), an update for vertex *src* increments a global counter `overlap_counter` if the `latest_bid` field for *src* reads n. In addition, another global counter `node_counter` is incremented to record the total number of unique vertices that appear in the ABR-active batch. When the updates of the ABR-active batch are over, the ratio of `overlap_counter` and `node_counter` provides the value of inter-batch locality. It is high if it exceeds a certain threshold which is determined empirically as follows: starting from a high threshold of 0.5, we progressively decrease the threshold and note the batch sizes where aggregation is activated and

the corresponding level of speedup. We choose a value of 0.25 where most of the larger batch sizes experience high performance improvement. Below 0.25, we note that aggregation is triggered for smaller batch sizes. However, we avoid granularity aggregation for smaller batch sizes (see below). In addition, the speedup from these smaller batch sizes is small due to a low overlap. For example, *yt* dataset at a batch size of 10K experiences computation aggregation at a threshold of 0.15 (but not at higher thresholds) but the corresponding speedup is only 8%.

Application scenarios. Extremely latency-sensitive applications (e.g., security applications such as financial fraud detection) utilize a fine-grained computation granularity or small batch size for faster reaction to graph modifications. Trading off granularity for a higher computation performance is not a good choice in these application scenarios. We experimentally show that the adaptive OCA deactivates at small batch sizes and only activates at relatively larger batch sizes. A larger batch size indicates an application scenario which can trade off some granularity for a higher compute efficiency. Moreover, when OCA is activated, we coarsen the granularity by only one additional batch size worth of graph modifications. In addition, OCA is an adaptive optimization and can be easily entirely turned off if the application does not tolerate any sacrifice in granularity even for the larger batch sizes.

6.5 Evaluation

6.5.1 Experimental Setup and Methodology

ABR, USC, and OCA are evaluated on a dual-socket Intel Xeon Platinum 8180 (Skylake) server with a total of 112 hardware execution threads (28 cores per socket, 2-way SMT). The server contains 38.5MB last-level cache per socket and 768GB memory. Per-

formance evaluation of HAU is done on Sniper-7.2 [145] with the baseline architecture in Table 6.1.

Table 6.1: Simulated Baseline Architecture on Sniper-7.2

core	16 cores, 2.5GHz, 4-issue
L1D/I	32KB private, 8-way, 3 cycles
L2	256KB private, 8-way, 8 cycles
L3	16MB NUCA (2MB slices), 16-way, 8 cycles bank access latency
NOC	4x4 mesh, 2-cycle hop, per-link per-direction bandwidth = 256 bits/cycle
DRAM	4 memory controllers, 17GB/s per controller, 40ns device access latency, queue delay modeled

Table 6.2: Evaluated Datasets

dataset (short name)	**vertices**	**edges**
Wiki-Talk (Talk) [86]	2,394,385	5,021,410
WebBerkStan (BerkStan) [86]	685,230	7,600,595
cit-Patents (Patents) [86]	3,774,768	16,518,948
Wiki-Topcats (Topcats) [86]	1,791,489	28,511,807
soc-LiveJournal (LJ) [86]	4,847,571	68,993,773
com-Friendster (Friendster) [86]	65,608,366	1,806,067,135
UK-Union-2006-2007 (UK) [148, 149]	133,633,040	5,507,679,822
Facebook-wall (FB) [150]	46,952	876,993
Flickr-photo (Flickr) [151]	11,730,773	34,734,221
Youtube (YT) [152]	3,223,589	12,223,774
Amazon-ratings (Amazon) [152]	2,146,057	5,838,041
Stack-overflow (Stack) [86]	2,601,977	63,497,050
Superuser (Superuser) [86]	194,085	1,443,339
Wiki-talk-temporal (Wiki) [86]	1,140,149	7,833,140

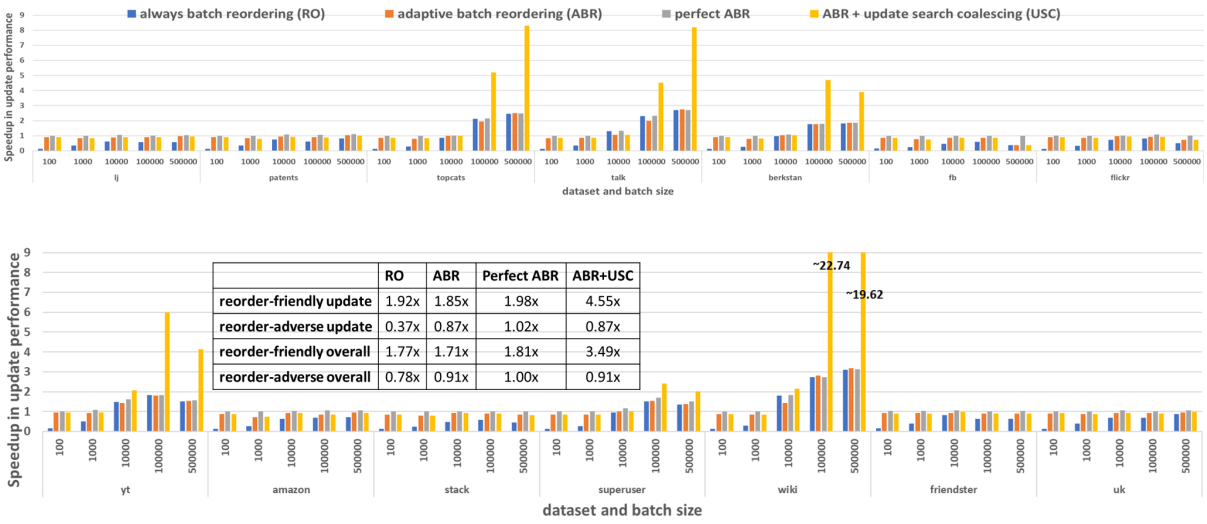
Table 6.2 shows the evaluated datasets. The first seven (*talk-uk*) are static datasets that are randomly shuffled to break any ordering in the input files (they are often ordered in increasing source vertex ID, which is not the likely scenario of edge appearance for real-world streaming graphs). The remaining datasets (*fb-wiki*) are timestamped, i.e., the input file specifies the order in which the edges appear in the graph. We use SAGA-Bench [153] and perform experiments on the adjacency list data structure because it is used

in multiple existing systems [25, 124, 130]. Four algorithms are evaluated: incremental PageRank (PR), incremental Single Source Shortest Paths (SSSP), static PR (start-from-scratch), and static SSSP. SAGA-Bench uses the computation model proposed in prior work [19,124] for incremental algorithms and takes the static versions from GAP [17]. The evaluated input batch sizes are 100, 1K, 10K, 100K, and 500K. Combining 14 datasets, 5 batch sizes, and 4 algorithms, we run 260 workloads. The largest datasets *friendster* and *uk* are run on only the incremental algorithms because prior work [153] has shown that incremental compute models provide significantly better performance for larger datasets. The speedup in update/compute performance for each workload represents the ratio (between the baseline and the proposed technique) of the total update/compute time across all the batches (we start from an empty graph). Real hardware experiments of ABR, USC, and OCA are repeated three times. For simulation-based evaluation of HAU, it is not feasible to perform experiments as extensively as on a real hardware. Each of the 260 workloads consists of hundreds of batches; months of simulation time would be required. Therefore, we evaluate HAU on a subset of 9 datasets and 4 batch sizes (Fig. 6.13). The datasets cover different sizes (vertex/edge counts) and types (shuffled/timestamped).

6.5.2 Experimental Results

Performance

Fig. 6.12 shows that ABR does not substantially compromise the high RO update performance of reordering-friendly cases. As summarized in the table-inset, the geometric mean across reordering-friendly cases shows that the update speedups of always-RO and ABR are 1.92× and 1.85×, respectively. For reordering-adverse cases, ABR successfully recovers the update performance from degradation in a naive always-RO



	RO	ABR	Perfect ABR	ABR+USC
reorder-friendly update	1.92x	1.85x	1.98x	4.55x
reorder-adverse update	0.37x	0.87x	1.02x	0.87x
reorder-friendly overall	1.77x	1.71x	1.81x	3.49x
reorder-adverse overall	0.78x	0.91x	1.00x	0.91x

Figure 6.12: Speedup in update performance from ABR and USC. Each bar is the average across runs with different algorithms. ABR parameters are n=10, λ=256, and TH=465 (Section 6.5.2). The inset-table shows the average update/overall performances for both the categories (averaged across all the dataset and batch size combinations which fall under the given category).

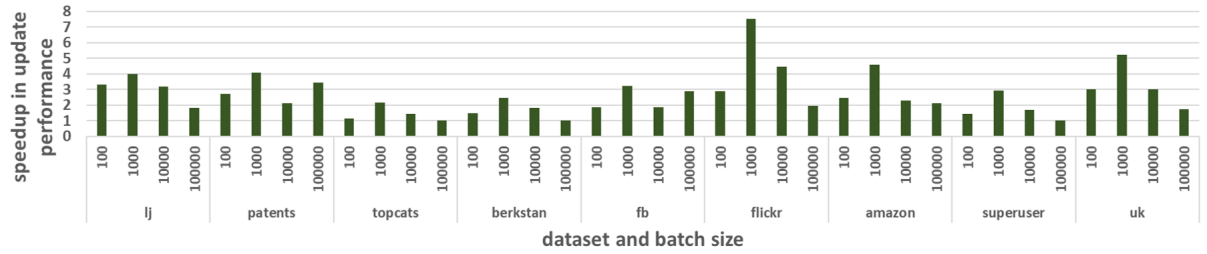


Figure 6.13: Speedup in update performance from ABR+USC+HAU (normalized to ABR+USC)

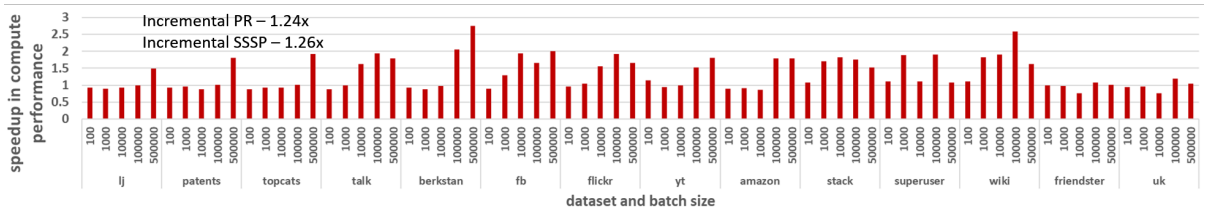


Figure 6.14: Speedup in compute performance from OCA

solution. Geometric mean across reordering-adverse cases shows that ABR pushes up the update performance closer to the baseline (0.37× to 0.87×). The table-inset shows that the benefits of ABR are carried over to the overall performance (i.e., update and compute combined), providing evidence that graph updates have an important contribution to the overall performance. ABR saves the overall performance from degradation for reordering-adverse cases (0.78× to 0.91×) while minimally disturbing the overall speedup of reordering-friendly cases (1.77× to 1.71×). The *perfect ABR* bars and inset column show that ABR performs close to a perfect adaptive technique with zero overheads. ABR performs at 93%, 85%, 94%, and 91% of the perfect ABR for reordering-friendly update, reordering-adverse update, reordering-friendly overall, and reordering-adverse overall performances, respectively. For the updates of reordering-friendly input batches, ABR and USC together provide average speedups of 4.55× (max 23× for *wiki*-100K and 20× for *wiki*-500K) and 3.49× (max 17× for *wiki*-100K, 500K) in update and overall performances, respectively. Reordering and USC software optimizations are not applied on reordering-adverse cases. Instead, HAU complements ABR in these scenarios to improve the update performance (Fig. 6.13). The update speedup obtained from ABR+USC+HAU is normalized to ABR+USC running on the simulated architecture in Table 6.1 (note that ABR+USC+HAU means reordering-adverse input batches undergo ABR and HAU, whereas reordering-friendly ones undergo ABR and USC). Compared to ABR only, HAU provides on average 2.6× (max 7.5×) improvement in update performance across the reordering-adverse cases. HAU is not applied to reordering-friendly *topcats*-100K, *berkstan*-100K, and *superuser*-100K, as shown by the ~1× speedup. They are executed in software mode with reordering and USC optimizations (Fig. 6.12).

For streaming graph computation, OCA is activated in cases of higher inter-batch overlap and can provide up to 2.7× speedup in compute performance (Fig. 6.14). Averaged across all datasets and batch sizes, the performance benefits experienced by in-

cremental PR and incremental SSSP are 1.24× and 1.26×, respectively. OCA is predominantly triggered at relatively larger batch sizes (a desirable feature as explained in Section 6.4). This happens because of: 1) inherent traits of large batches, and 2) our choice of a relatively high overlap threshold. Larger input batches inherently contain larger number of vertices, leading to an increased possibility of high overlap in unique vertices between consecutive batches. Smaller input batches also exhibit some extent of inter-batch overlap which fails to satisfy the overlap threshold.

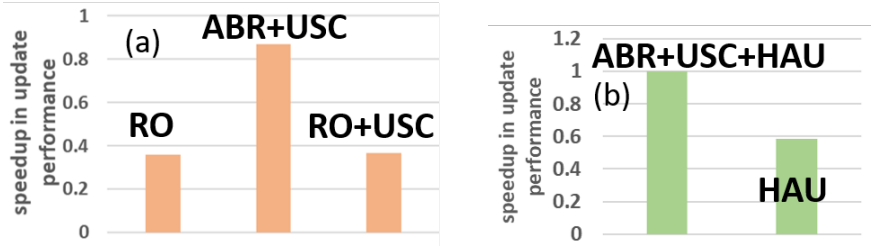


Figure 6.15: (a) Extension of Fig. 6.12 across reordering-adverse cases; shows the impact of enforcing software optimizations (RO+USC) (b) Extension of Fig. 6.13 across reordering-friendly cases; shows the impact of enforcing HAU

Dynamic SW/HW graph updates

We quantitatively show that input-aware SW/HW execution mode outperforms an input-oblivious HW-only or SW-only update technique (see Section 6.3.6 for insights and explanation). SW-only: Our input-aware solution deviates from an input-oblivious SW-only solution by applying HAU on reordering-adverse input batches. We instead enforce RO+USC on them (Fig. 6.15 (a)) and find that RO+USC performs almost as poorly as RO. Since ABR+USC outperforms RO+USC (Fig. 6.15 (a)) and ABR+USC+HAU outperforms ABR+USC (Fig. 6.13), it follows that ABR+USC+HAU outperforms RO+USC. HW-only: Our solution deviates from a HW-only solution by dynamically applying RO+USC on reordering-friendly input batches. We extend Fig. 6.13 by enforcing HAU on them and show that the performance degrades (Fig. 6.15 (b)).

Further analysis

We further quantitatively analyze different techniques. Any overheads analyzed in this section are already included in the speedups reported in Section 6.5.2.

ABR and OCA overheads (Fig. 6.16): Reordered ABR-active batches experience negligible overhead (0.90×) due to CAD_λ collection. Non-reordered ABR-active batches experience a higher overhead on average (0.54×) because of instrumentation with a concurrent hash map. However, a small number of ABR-active batches ensures a small combined overhead across all batches. Normalized to ABR+USC, the average overhead incurred by OCA is very small (Fig. 6.16 (b)).

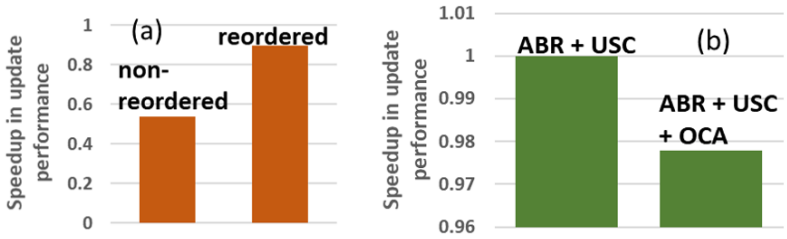


Figure 6.16: Overhead of (a) ABR and (b) OCA

USC insights and overheads: We study the examples of *superuser*-100K and *wiki*-500K to provide three key insights:

- High/low-degree input batches: *Wiki*-500K predominantly achieves a larger speedup than *superuser*-100K (Fig. 6.17) because the input batches of the former are high-degree in terms of both CAD_λ (1072 vs. 528) and maximum degree (43992 vs 3171). The exception are the first two batches of *wiki*-500K which are low-degree and where the graph is small (see below). A high-degree input batch means more coalescing, leading to more search savings.

- Graph size: For a given combination of dataset and batch size, USC's performance benefit increases as the graph becomes larger (Fig. 6.17). Over time, search becomes

more expensive as vertex degrees increase, and the benefit from search savings becomes higher.

- Negligible overhead: USC does not degrade the update performance even when the scope of speedup is smaller. This provides evidence that USC incurs negligible overhead of preparing the hash table (Fig. 6.7).

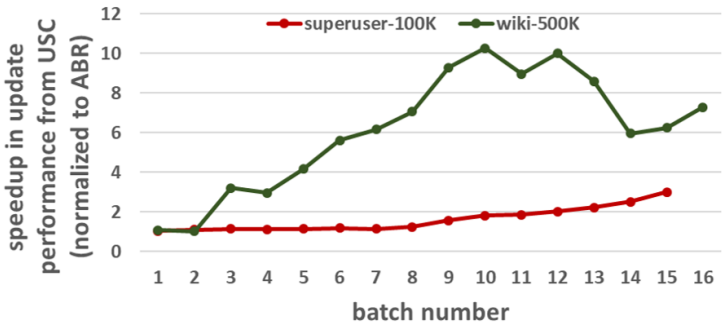


Figure 6.17: Temporal speedup from USC (normalized to ABR)

ABR parameters/accuracy: The design parameters of ABR are n, λ, and TH. By analyzing the batches from different combinations of dataset and batch size, we choose the combination of λ and TH which maximizes the decision-making accuracy at 97% (λ=256 and TH=465) (Fig. 6.18(a)). The parameter n impacts both the decision accuracy and the overhead. A large n can reduce ABR overhead by reducing the frequency of instrumentation. However, it leads to coarse-grained decision-making which can miss temporal fluctuations in degree distributions, compromising ABR accuracy and performance. Fig. 6.18(b) shows that a larger n leads to a slightly better update performance *on average* (1.04× at n=10 versus 1.07× at n=100). However, *flickr*-500K, *yt*-100K, and *stack*-500K experience a poorer performance. For example, *stack*-500K has 127 batches in total, leading to 2 ABR decisions at n=100 and 12 ABR decisions at n=10. Therefore, n=100 misses some over-time fluctuations.

HAU analysis: We study *uk*-100K at a batch number of 100. For work distribution,

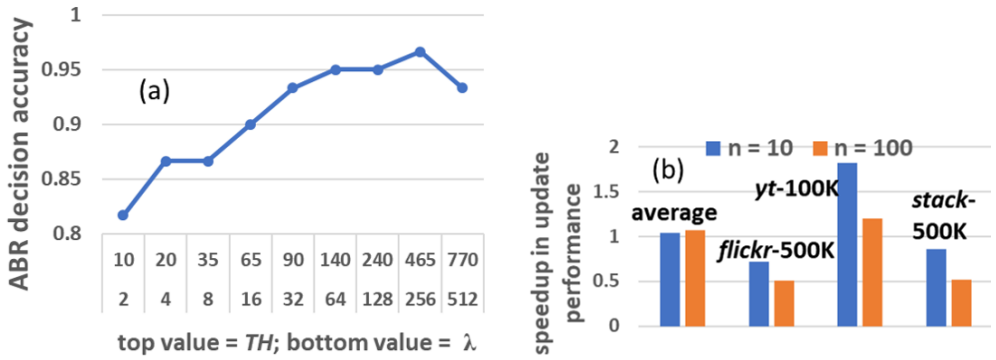


Figure 6.18: (a) ABR accuracy as a function of λ-TH combination. (b) Sensitivity of update performance to n.

the maximum value of vertices/core (core 6) is 3% higher than the minimum (core 11) and 1.3% higher than the average across all cores (Fig. 6.19; since core 0 hosts the master thread in SAGA-Bench setup, we show the information on cores 1-15 which host the worker threads for graph updates). The maximum number of cachelines accessed per dedicated cache controller logic (core 13) is 600% higher than the minimum (core 11) and 148% higher than the average across all cores. A heavy-workload core does not straggle substantially because HAU eliminates i) remote cache accesses and ii) CPU instruction overheads for searches. In other words, HAU substantially minimizes time per unit of work (cacheline access + cacheline search), ensuring that non-uniform work distribution is not the most significant performance limiter for HAU (Fig. 6.13 shows HAU's existing design can achieve on average 2.6× update performance improvement). First, our update task assignment ensures that 98%-99% of the accessed edge data cachelines hit in the local core tile (Fig. 6.20), eliminating straggling due to more expensive remote cache accesses. In fact, HAU eliminates all remote cache accesses that would otherwise be present in the baseline software updates. Second, specialized logic in the cache controller eliminates the overheads of several CPU instructions for searches, limiting straggling due to time-consuming searches. We believe that, in future, HAU can easily be optimized

with well-known load balancing schemes such as work-stealing [154]. Finally, Fig. 6.20 shows the impact of using NOC for update task distribution. The increase in average packet latency is within 10%, and some cores also experience a decrease in packet delay, depending on the relative change in the number of different types of packets.

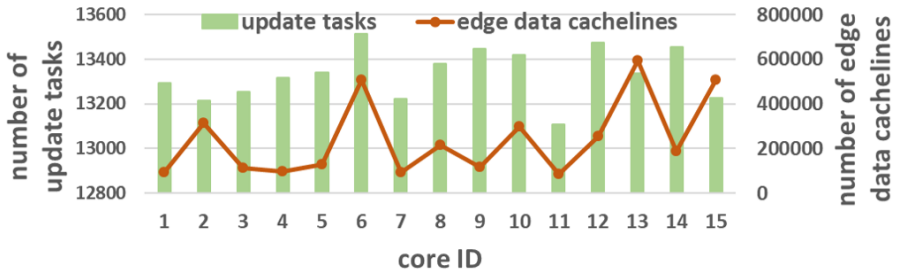


Figure 6.19: Work distribution among cores in HAU

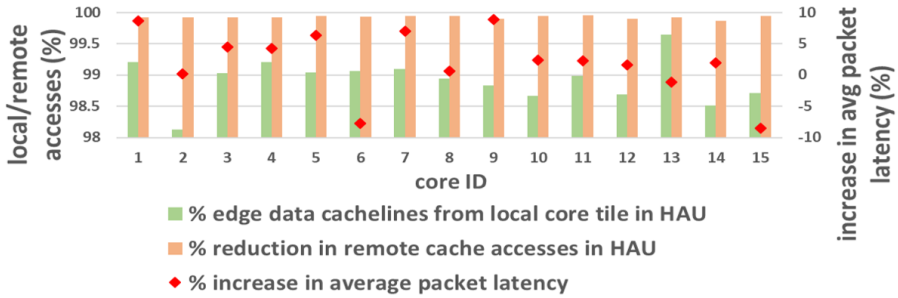


Figure 6.20: Remote cache accesses and NOC performance

6.6 Conclusion

We propose the SPRING design approach which consists of input-aware software and hardware solutions to improve the performance of streaming graph workloads. Evaluated across 260 workloads, our proposed techniques provide on average 4.55× and 2.6× speedup in graph update for different input types (on top of eliminating the performance degradation from input-oblivious batch reordering). The graph compute performance is improved by 1.26× (up to 2.7×).

Chapter 7

Summary

7.1 Thesis Contributions

The contributions of this dissertation lie at the intersection of important emerging applications and computer architecture. For the former, we focus on both static and streaming graph processing which are important in data analytics scenarios such as recommendation systems, financial fraud detection, and social network analysis. The rich space of graph applications poses several challenges for the computer architecture community. First, standard static graph algorithm performance is sub-optimal on today's general-purpose architectures such as CPUs due to inefficiencies in the memory subsystem. It is currently increasingly difficult to rely on relative compute/memory technology scaling for continued performance improvement for a given optimized static graph algorithm on a general-purpose CPU. This is because graph applications become more memory-bound due to the explosion of data, whereas the memory technology does not scale as fast as the compute technology. Hence, it is worthwile to specifically study the application bottlenecks and address them using domain-specific computer architecture. Second, while a large body of research in the computer architecture community focuses on

static graph workloads, streaming graphs remain completely unexplored. In reality, however, most graphs are fast-changing in today's big data era where data evolves rapidly. The primary practical barriers for computer architecture researchers toward studying streaming graphs are immature software, a lack of systematic software analysis, and an absence of open-source benchmarks. This dissertation seeks to solve these challenges for both static and streaming graph workloads through benchmarking, performance analysis, and CPU-centric domain-specific architectures using software/hardware co-design.

Chapters 3 and 4 of the dissertation analyze and optimize static graph workloads. In Chapter 3, we perform a data-aware performance analysis of the GAP benchmark suite [17], focusing on the memory-level parallelism and the cache hierarchy. We show that load-load dependency chains that involve specific application data types, rather than the instruction window size limitation, make up the key bottleneck in achieving a high memory-level parallelism. Moreover, we find that different graph data types exhibit heterogeneous reuse distances. The architectural consequences are (1) the private L2 cache shows negligible impact on improving system performance, (2) the shared L3 cache shows higher performance sensitivity, and (3) the graph property data type benefits the most from a larger shared L3 cache. Based on these profiling observations, we propose, in Chapter 4, a domain-specific prefetcher called DROPLET to solve the memory access bottleneck. DROPLET is a physically decoupled but functionally cooperative prefetcher co-located at the L2 cache and at the memory controller. Moreover, DROPLET is data-aware because it prefetches different graph data types differently according to their intrinsic reuse distances. DROPLET achieves 19%-102% performance improvement over a no-prefetch baseline and 14%-74% performance improvement over a Variable Length Delta Prefetcher. DROPLET also performs 4%-12.5% better than a monolithic L1 prefetcher similar to the state-of-the-art prefetcher for graphs.

Chapters 5 and 6 of the dissertation focus on streaming graph workloads. For the

latter, this thesis develops a performance analysis framework called SAGA-Bench and performs workload characterization at both the software and the architecture levels. SAGA-Bench (Chapter 5) is targeted at software and hardware studies of the essential data structures and compute models proposed across various existing streaming graph systems. For software performance analysis, we enable systematicness by using comparable implementations of the core software components (without system-specific optimizations) and identical measurement methodology, thus alleviating the problem of difficult-to-interpret comparisons across heterogeneous stand-alone systems. For hardware studies, SAGA-Bench provides a benchmark to study the architecture bottlenecks for these workloads. Workload characterization on SAGA-Bench leads to several findings: 1) the performance limitation of the graph update phase, 2) the input-dependent software performance trade-offs in graph updates, and 3) the difference in architecture resource utilization (core counts, memory bandwidth, and cache hierarchy) between the graph update and the graph compute phases. In addition, in Chapter 6, using the SPRING design approach, we demonstrate that input knowledge-driven software and hardware co-design is critical to optimize the performance of streaming graph processing. To improve graph update efficiency, we first characterize the performance trade-offs of an input-oblivious software technique called batch reordering [25, 26]. Guided by our findings, we propose input-aware batch reordering to adaptively reorder input batches based on their degree distributions. To complement adaptive batch reordering, we propose updating graphs dynamically, based on their input characteristics, either in software (via update search coalescing) or in hardware (via acceleration support). To improve graph computation efficiency, we present input-aware work aggregation which adaptively modulates the computation granularity based on inter-batch locality characteristics. Evaluated across 260 workloads, our input-aware techniques provide on average 4.55× and 2.6× improvement in graph update performance for different input types (on top of eliminating the

performance degradation from input-oblivious batch reordering). The graph compute performance is improved by 1.26× (up to 2.7×). Evaluated across 260 workloads, our input-aware techniques provide on average 4.55× and 2.6× improvement in graph update performance for different input types. The graph compute performance is improved by 1.26× (up to 2.7×).

7.2 Future Directions

As highlighted below, the work developed in this thesis opens the path for multiple future research directions:

- *Flexible prefetchers for graph workloads*: DROPLET provides performance improvement for static graphs represented in CSR, the most common graph representation. However, there exist alternative static graph representations such as edge list [111, 155], shards [55], doubly compressed sparse row/column (DCSR/DCSC) [156], etc. which, although not the most common case, may be valuable in certain cases (e.g., DCSR is more space-efficient than CSR for hypersparse graphs). A practical and valuable future direction is to extend DROPLET to be a flexible and programmable prefetcher to be able to handle multiple underlying graph representations.

- *Application-specific software-assisted hardware prefetching*: DROPLET provides evidence for the benefits of software-assisted hardware prefetching (i.e., with data type hints from the software, a hardware prefetcher is capable of achieving high accuracy and performance). This prefetching methodology can be practical and a source of high performance for not just graph workloads but also for other sparse workloads such as sparse linear algebra. For these types of workloads, it is difficult

to rely on a completely hardware prefetcher due to the challenging and difficult-to-predict access patterns. On the other hand, data type hints from the application layer can make prefetching easier for the hardware.

- *Performance analysis of future novel data structures and compute models for streaming graph workloads*: SAGA-Bench simultaneously provides 1) a common platform for performance analysis studies of software techniques and 2) a benchmark for architecture studies. As streaming graph software is still in active research, the API of SAGA-Bench is designed to be flexible so that future streaming graph codes can be easily implemented and analyzed on this common platform. Our framework improves research productivity during the stage of novel workload discovery and characterization. A common platform 1) relieves the difficulty of navigating through heterogeneous stand-alone systems for software performance comparison, and 2) provides a means to easily implement the core software of the workload to quickly characterize it on the hardware to understand the architecture bottlenecks.

- *Architectural support for streaming graph workloads to make more practical domain-specific architectures for graph processing*: Today's graph-targeted domain-specific architectures ignore streaming graphs, whereas, in reality, graphs are seldom static. By neglecting the dynamic nature of graphs, the research community is missing the opportunity to design more adequate architecture solutions for practical and realistic graph processing. Motivated by this situation, we provide input-dependent architectural support for streaming graph workloads in this thesis. However, this is only the first step and there are numerous opportunities to further pursue this direction. For example, the update and the compute stages in streaming graphs exhibit different levels of parallelism and working set sizes, leading to heterogeneous architecture resource utilizations in terms of core counts, cache hierarchy levels,

and memory and inter-socket bandwidths. Future graph-targeted domain-specific architectures need to be equipped with dynamic resource allocation strategies between the two stages to achieve a high utilization. In addition, an interleaved update/compute execution flow provides opportunities for inter-phase optimizations. For example, the slack in hardware resource utilization of the update phase can be leveraged to optimize the compute phase.

Bibliography

[1] *Spatial and graph anaytics with oracle database 18c*, tech. rep., Oracle, 2018.

[2] R. Dillet, "Amazon introduces an aws graph database service called amazon neptune." `https://techcrunch.com/2017/11/29/amazon-introduces-an-aws-graph-database-service-called-amazon-neptune/`, 2017.

[3] B. Shao, H. Wang, and Y. Li, *Trinity: A distributed graph engine on a memory cloud*, in *International Conference on Management of Data*, pp. 505–516, ACM, 2013.

[4] 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 *SIGMOD*, pp. 135–146, ACM, 2010.

[5] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, *One trillion edges: Graph processing at facebook-scale*, *Proceedings of the VLDB Endowment* **8** (2015), no. 12 1804–1815.

[6] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, *Wtf: The who to follow service at twitter*, in *International conference on World Wide Web*, pp. 505–514, ACM, 2013.

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

[8] N. Ismail, "Graph databases lie at the heart of $7tn self-driving car opportunity." `http://www.information-age.com/graph-databases-heart-self-driving-car-opportunity-123468309/`, Nov 2017.

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

[10] 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 *MASCOTS*, pp. 49–58, IEEE, 2016.

[11] 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-International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, IEEE, 2015.

[12] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, P. Faraboschi, and S. Katti, *Parallel graph processing: Prejudice and state of the art*, in *International Conference on Performance Engineering*, pp. 85–90, ACM, 2016.

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

[14] 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.

[15] S. Ainsworth and T. M. Jones, *Graph prefetching using data structure knowledge*, in *International Conference on Supercomputing*, p. 39, ACM, 2016.

[16] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout, *Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads*, in *International Symposium on Workload Characterization*, pp. 38–49, IEEE, 2011.

[17] S. Beamer, K. Asanović, and D. Patterson, *The gap benchmark suite*, *CoRR abs/1508.03619* (2015).

[18] A. Pareja, G. Domeniconi, J. Chen, T. Ma, T. Suzumura, H. Kanezashi, T. Kaler, T. B. Schardl, and C. E. Leiserson, *EvolveGCN: Evolving graph convolutional networks for dynamic graphs*, in *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

[19] 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.

[20] 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* **11** (2018), no. 12 1876–1888.

[21] 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.

[22] 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.

[23] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, *Efficiently prefetching complex address patterns*, in *International Symposium on Microarchitecture*, pp. 141–152, IEEE, 2015.

[24] K. J. Nesbit and J. E. Smith, *Data cache prefetching using a global history buffer*, in *Software, IEEE Proceedings-*, pp. 96–96, IEEE, 2004.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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* **10** (2017), no. 8 877–888.

[42] 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.

[43] 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.

[44] 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.

[45] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, *The architectural implications of facebook's dnn-based personalized recommendation*, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 488–501, 2020.

[46] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, *Applied machine learning at facebook: A datacenter infrastructure perspective*, in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 620–629, 2018.

[47] F. Limited, "Fugaku retains title as world's fastest supercomputer." `https://www.hpcwire.com/off-the-wire/fugaku-retains-title-as-worlds-fastest-supercomputer/#:~:text=In%20addition%2C%20Fugaku%20is%20currently,which%20starts%20in%20April%202021)`, Nov 2020.

[48] B. Akin, Z. A. Chishti, and A. R. Alameldeen, *Zcomp: Reducing dnn cross-layer memory footprint using vector extensions*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, p. 126–138, 2019.

[49] A. Frumusanu, "Amazon's arm-based graviton2 against amd and intel: Comparing cloud compute." `https://www.anandtech.com/show/15578/cloud-clash-amazon-graviton2-arm-against-intel-and-amd`, Mar 2020.

[50] T. P. Morgan, "Ampere steams ahead with arm server chips." `https://www.nextplatform.com/2021/01/19/ampere-steams-ahead-with-arm-server-chips/`, Jan 2021.

[51] *Nvidia announces cpu for giant ai and high performance computing workloads*, Apr 2021.

[52] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, *Powergraph: Distributed graph-parallel computation on natural graphs.*, in *OSDI*, vol. 12, p. 2, ACM, 2012.

[53] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, *Distributed graphlab: a framework for machine learning and data mining in the cloud*, *VLDB Endowment* **5** (2012), no. 8 716–727.

[54] V. Kalavri, V. Vlassov, and S. Haridi, *High-level programming abstractions for distributed graph processing*, *IEEE Transactions on Knowledge and Data Engineering* **30** (2018), no. 2 305–324.

[55] A. Kyrola, G. E. Blelloch, and C. Guestrin, *Graphchi: Large-scale graph computation on just a pc*, USENIX, 2012.

[56] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, *Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc*, in *International conference on Knowledge discovery and data mining*, pp. 77–85, ACM, 2013.

[57] X. Zhu, W. Han, and W. Chen, *Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning.*, in *USENIX Annual Technical Conference*, pp. 375–386, 2015.

[58] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, *Wonderland: A novel abstraction-based out-of-core graph processing system*, in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 608–621, ACM, 2018.

[59] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, *Mosaic: Processing a trillion-edge graph on a single machine*, in *European Conference on Computer Systems*, pp. 527–543, ACM, 2017.

[60] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, *A scalable processing-in-memory accelerator for parallel graph processing*, in *ISCA*, pp. 105–117, IEEE, 2015.

[61] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, *Graphicionado: A high-performance and energy-efficient accelerator for graph analytics*, in *MICRO*, pp. 1–13, IEEE, 2016.

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

[63] 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 *International Symposium on Cluster, Cloud and Grid Computing*, pp. 731–734, May, 2017.

[64] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, *Energy efficient architecture for graph analytics accelerators*, in *International Symposium on Computer Architecture*, pp. 166–177, IEEE, 2016.

[65] Y. Perez, R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec, *Ringo: Interactive graph analytics on big-memory machines*, in *International Conference on Management of Data*, pp. 1105–1110, ACM, 2015.

[66] J. Leskovec and R. Sosič, *Snap: A general-purpose network analysis and graph-mining library*, *ACM Trans. Intell. Syst. Technol.* **8** (July, 2016) 1:1–1:20.

[67] 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.

[68] 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*, *CoRR* **abs/1711.07601** (2017) [arXiv:1711.0760].

[69] Z. Shang, F. Li, J. X. Yu, Z. Zhang, and H. Cheng, *Graph analytics through fine-grained parallelism*, in *International Conference on Management of Data*, pp. 463–478, ACM, 2016.

[70] J. Lin, *Scale up or scale out for graph processing?*, *IEEE Internet Computing* **22** (May, 2018) 72–78.

[71] *System memory at a fraction of the dram cost*, tech. rep., Intel Corporation, 2017.

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

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

[74] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, *Gunrock: A high-performance graph processing library on the gpu*, in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[75] T. K. Aasawat, T. Reza, and M. Ripeanu, *How well do cpu, gpu and hybrid graph processing frameworks perform?*, in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 458–466, 2018.

[76] J. Shun and G. E. Blelloch, *Ligra: A lightweight graph processing framework for shared memory*, in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), p. 135–146, Association for Computing Machinery, 2013.

[77] D. Nguyen, A. Lenharth, and K. Pingali, *A lightweight infrastructure for graph analytics*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), p. 456–471, Association for Computing Machinery, 2013.

[78] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, *The tao of parallelism in algorithms*, in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), p. 12–25, Association for Computing Machinery, 2011.

[79] H. Liu and H. H. Huang, *Simd-x: Programming and processing of graph algorithms on gpus*, in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 411–428, USENIX Association, July, 2019.

[80] Y. Chou, B. Fahs, and S. Abraham, *Microarchitecture optimizations for exploiting memory-level parallelism*, in *International Symposium on Computer Architecture*, pp. 76–87, IEEE, 2004.

[81] W. Heirman, T. Carlson, and L. Eeckhout, *Sniper: Scalable and accurate parallel multi-core simulation*, in *ACACES*, pp. 91–94, HiPEAC, 2012.

[82] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, *An evaluation of high-level mechanistic core models*, *TACO* **11** (2014), no. 3 28.

[83] A. Akram and L. Sawalha, *×86 computer architecture simulators: A comparative study*, in *International Conference on Computer Design*, pp. 638–645, IEEE, 2016.

[84] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, *Cacti 6.0: A tool to model large caches*, *HP laboratories* (2009).

[85] 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 *International conference on Management of data*, pp. 979–990, ACM, 2014.

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

[87] Y. Kora, K. Yamaguchi, and H. Ando, *Mlp-aware dynamic instruction window resizing for adaptively exploiting both ilp and mlp*, in *International Symposium on Microarchitecture*, pp. 37–48, IEEE, 2013.

[88] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt, *Accelerating dependent cache misses with an enhanced memory controller*, in *ISCA*, pp. 444–455, IEEE Press, 2016.

[89] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, *Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers*, in *International Symposium on High Performance Computer Architecture*, pp. 63–74, IEEE, 2007.

[90] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, *Prefetch-aware dram controllers*, in *Proceedings International Symposium on Microarchitecture*, pp. 200–209, IEEE Computer Society, 2008.

[91] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, vol. 3A. 9, 2016.

[92] T. 2nd Data Prefetching Championship (DPC2). `http://comparch-conf.gatech.edu/dpc2/`, 2015.

[93] F. Liu and R. B. Lee, *Random fill cache architecture*, in *International Symposium on Microarchitecture*, pp. 203–215, Dec, 2014.

[94] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, *Gather-scatter dram: in-dram address translation to improve the spatial locality of non-unit strided accesses*, in *Proceedings International Symposium on Microarchitecture*, pp. 267–280, ACM, 2015.

[95] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, *Graphpim: Enabling instruction-level pim offloading in graph computing frameworks*, in *International Symposium on High Performance Computer Architecture*, pp. 457–468, IEEE, 2017.

[96] R. Cooksey, S. Jourdan, and D. Grunwald, *A stateless, content-directed data prefetching mechanism*, in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 279–290, ACM, 2002.

[97] C.-L. Yang and A. R. Lebeck, *Push vs. pull: Data movement for linked data structures*, in *International Conference on Supercomputing*, pp. 176–186, ACM, 2000.

[98] A. Roth, A. Moshovos, and G. S. Sohi, *Dependence based prefetching for linked data structures*, in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 115–126, ACM, 1998.

[99] E. Ebrahimi, O. Mutlu, and Y. N. Patt, *Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems*, in *International Symposium on High Performance Computer Architecture*, pp. 7–17, IEEE, 2009.

[100] D. Joseph and D. Grunwald, *Prefetching using markov predictors*, *IEEE transactions on computers* **48** (1999), no. 2 121–133.

[101] J. Kim, S. H. Pugsley, P. V. Gratz, A. Reddy, C. Wilkerson, and Z. Chishti, *Path confidence based lookahead prefetching*, in *International Symposium on Microarchitecture*, p. 60, IEEE Press, 2016.

[102] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, *Runahead execution: An alternative to very large instruction windows for out-of-order processors*, in *International Symposium on High-Performance Computer Architecture*, pp. 129–140, IEEE, 2003.

[103] M. Annavaram, J. M. Patel, and E. Davidson, *Data prefetching by dependence graph precomputation*, in *ISCA*, pp. 52–61, IEEE, 2001.

[104] P. Michaud, *Best-offset hardware prefetching*, in *International Symposium on High Performance Computer Architecture*, pp. 469–480, IEEE, 2016.

[105] Y. Ishii, M. Inaba, and K. Hiraki, *Access map pattern matching for high performance data cache prefetch*, *Journal of Instruction-Level Parallelism* **13** (2011) 1–24.

[106] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, *Imp: Indirect memory prefetcher*, in *International Symposium on Microarchitecture*, pp. 178–190, ACM, 2015.

[107] P. Yedlapalli, J. Kotra, E. Kultursay, M. Kandemir, C. R. Das, and A. Sivasubramaniam, *Meeting midway: Improving cmp performance with memory-side prefetching*, in *International conference on Parallel architectures and compilation techniques*, pp. 289–298, IEEE Press, 2013.

[108] C. J. Hughes and S. V. Adve, *Memory-side prefetching for linked data structures for processor-in-memory systems*, *Journal of Parallel and Distributed Computing* **65** (2005), no. 4 448–463.

[109] 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.

[110] 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.

[111] 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.

[112] 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.

[113] 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.

[114] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, *et. al.*, *Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 615–628, 2019.

[115] 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, IEEE, 2018.

[116] 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* **43** (2016), no. 3 105–117.

[117] 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.

[118] 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.

[119] 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.

[120] 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.

[121] 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.

[122] 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* **68** (2019), no. 8 1131–1146.

[123] 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.

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

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

[126] 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.

[127] M. Mariappan and K. Vora, *Graphbolt: Dependency-driven synchronous processing of streaming graphs*, in *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–16, 2019.

[128] A. Iyer, L. E. Li, and I. Stoica, *Celliq: Real-time cellular network analytics at scale*, in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pp. 309–322, 2015.

[129] 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.

[130] G. Feng, Z. Ma, D. Li, X. Zhu, Y. Cai, W. Han, and W. Chen, *Risgraph: A real-time streaming system for evolving graphs*, *arXiv preprint arXiv:2004.00803* (2020).

[131] 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.

[132] H. Omar, M. Ahmad, and O. Khan, *Graphtuner: An input dependence aware loop perforation scheme for efficient execution of approximated graph algorithms*, in *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 201–208, IEEE, 2017.

[133] M. Ahmad, H. Dogan, C. J. Michael, and O. Khan, *Heteromap: A runtime performance predictor for efficient processing of graph analytics on heterogeneous multi-accelerators*, in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 268–281, IEEE, 2019.

[134] M. Ahmad and O. Khan, *Gpu concurrency choices in graph analytics*, in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–10, IEEE, 2016.

[135] V. Balaji and B. Lucia, *When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs*, in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 203–214, IEEE.

[136] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, *Graphssd: graph semantics aware ssd*, in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 116–128, 2019.

[137] A. Segura, J.-M. Arnau, and A. González, *Scu: a gpu stream compaction unit for graph processing*, in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 424–435, IEEE, 2019.

[138] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, *Graphpulse: An event-driven hardware accelerator for asynchronous graph processing*, in *2020 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-53),*, IEEE, 2020.

[139] `https://www.boost.org/doc/libs/1_67_0/libs/sort/doc/html/sort/parallel/parallel_stable_sort.html`, 2017.

[140] `https://software.intel.com/en-us/node/506191`, 2020.

[141] A. D. Broido and A. Clauset, *Scale-free networks are rare*, *Nature communications* **10** (2019), no. 1 1–10.

[142] Y.-H. Eom and H.-H. Jo, *Tail-scope: Using friends to estimate heavy tails of degree distributions in large-scale complex networks*, *Scientific reports* **5** (2015) 09752.

[143] G. Buzsáki and K. Mizuseki, *The log-dynamic brain: how skewed distributions affect network operations*, *Nature Reviews Neuroscience* **15** (2014), no. 4 264–278.

[144] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[145] T. E. Carlson, W. Heirman, and L. Eeckhout, *Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations*, in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 52:1–52:12, Nov., 2011.

[146] P. Faldu, J. Diamond, and B. Grot, *Domain-specialized cache management for graph analytics*, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 234–248, IEEE, 2020.

[147] A. Samara and J. Tuck, *The case for domain-specialized branch predictors for graph-processing*, *IEEE Computer Architecture Letters* **19** (2020), no. 2 101–104.

[148] P. Boldi, M. Santini, and S. Vigna, *A large time-aware graph*, *SIGIR Forum* **42** (2008), no. 2 33–38.

[149] "Laboratory for web algorithms." `http://law.di.unimi.it/datasets.php`.

[150] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, *On the evolution of user interaction in facebook*, in *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*, August, 2009.

[151] M. Cha, A. Mislove, and K. P. Gummadi, *A Measurement-driven Analysis of Information Propagation in the Flickr Social Network*, in *In Proceedings of the 18th International World Wide Web Conference (WWW'09)*, (Madrid, Spain), April, 2009.

[152] R. A. Rossi and N. K. Ahmed, *The network data repository with interactive graph analytics and visualization*, in *AAAI*, 2015.

[153] A. Basak, J. Lin, R. Lorica, X. Xie, Z. Chishti, A. Alameldeen, and Y. Xie, *Saga-bench: Software and hardware characterization of streaming graph analytics workloads*, in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[154] R. D. Blumofe and C. E. Leiserson, *Scheduling multithreaded computations by work stealing*, *Journal of the ACM (JACM)* **46** (1999), no. 5 720–748.

[155] 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.

[156] A. Buluc and J. R. Gilbert, *On the representation and multiplication of hypersparse matrices*, in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–11, 2008.