Data-Aware Cache Management for Graph Analytics

Neelam Sharma*§ Varun Venkitaraman*§ Newton*†§ Vikash Kumar* Shubham Singhania* Chandan Kumar Jha*
* Indian Institute of Technology Bombay † National University of Singapore

***Abstract*—Graph analytics is powering a wide variety of applications in the domains of cybersecurity, contact tracing, and social networking. It consists of various algorithms (or workloads) that investigate the relationships between entities involved in transactions, interactions, and organizations. CPU-based graph analytics is inefficient because their cache hierarchy performs poorly owing to highly irregular memory access patterns of graph workloads. Policies managing the cache hierarchy in such systems are ignorant to the locality demands of different data types within graph workloads, and therefore are suboptimal.**

In this paper, we conduct an in-depth data type aware characterization of graph workloads to better understand the cache utilization of various graph data types. We find that different levels of the cache hierarchy are more sensitive to the locality demands of certain graph data types than others. Hence, we propose GRACE, a graph data-aware cache management technique, to increase cache hierarchy utilization, thereby minimizing off-chip memory traffic and enhancing performance. Our thorough evaluations show that GRACE, when augmented with a vertex reordering algorithm, outperforms a recent cache management scheme by up to $1.4\times$, with up to 27% reduction in expensive off-chip memory accesses. Thus, our work demonstrates that awareness of different graph data types is critical for effective cache management in graph analytics.

***Index Terms*—Cache Management, Graph Analytics, Cache Bypassing**

I. Introduction

Graph analytics is an emerging technology with applications in diverse fields such as network analysis, cybersecurity, social media analysis, and COVID therapeutic discovery [1]. It is gaining prominence in the big data era due to its vast potential of uncovering valuable insights by processing graphs representing various entities (vertices) and relational links (edges) connecting them. Hence, graph analytics-powered technologies will be facilitators of the developing data-driven economy; for example, the graph analytics-powered self-driving vehicle sector is anticipated to be a $7 trillion market by 2050 [2].

The most commonly used graph analytics solutions on the market today are distributed hardware systems. They have been developed as an efficient way of graph processing by a fairly large body of research. However, such systems have many constraints, including 1) efficient graph distribution across compute nodes, and 2) avoiding the resulting network communication overheads [3]. With increased main memory capacity and core count, processing graphs on a single high-end-server machine is becoming a promising paradigm for graph analytics [1], [3]. Single-machine graph analytics do not have the aforementioned challenges, making it a viable alternative to distributed systems. However, such systems have its own set of challenges.

The large size of input graphs makes single-machine graph analytics difficult since inefficiencies in such machines' existing memory subsystem limit their performance [1], [3]. On such machines, processing a typical input graph results in a working dataset size larger than the size of on-chip caches, resulting in many off-chip memory accesses. Prior research has demonstrated that graph algorithms can spend up to 80% of their execution time just waiting for data from DRAMs [1]. The highly irregular access pattern of graphs workloads while processing graphs is one of the causes of frequent cache misses. Previous research has shown that modern cache management methods are inefficient at utilizing cache locality with such irregular accesses [1], [4]. The incorrect assumption that data accesses particular to a program-counter or memory-region will have similar data locality [1] has been a significant factor for the poor performance of signature-based cache management techniques like SHiP [5] and Hawkeye [6].

GRASP [4], a recent cache management technique designed specifically for graph analytics, has been more effective than prior approaches. It observed that a graph has a few high-degree vertices due to the power-law degree distribution of real-world graphs, indicating that such vertices are highly reused. However, it makes the incorrect assumption that such vertices would have similar reuse and that the reuse remains consistent across various phases of graph processing [1]. Moreover, it is dependent on a specific graph re-ordering scheme to preprocess the input graph, thus increasing the preprocessing cost of a graph. We observed that understanding different graph data types and their data localities is a critical insight that can improve the effectiveness of cache management approaches. Furthermore, the data awareness of these techniques can be augmented by understanding the interactions among these data types. Hence, we propose *GRACE*, a novel **GRA**ph data-aware **C**ach**E** management scheme. *GRACE* considers the interference between different graph data types and is sensitive to their cache locality demands while managing the cache hierarchy.

To summarize, our key contributions are as follows:

1) We quantitatively show the destructive interference among different graph data types on cache performance.
2) We propose, *GRACE*, a graph data-aware cache hierarchy management strategy for single-machine graph analytics.
3) We augment *GRACE* with a vertex reordering scheme to propose *GRACE++* that outperforms recent cache management scheme by up to $1.4\times$ and also reduces costly off-chip DRAM accesses by 27%.

II. Background and Motivation

In this section, we briefly describe a graph, its representation in a compressed sparse row (CSR) format, and the terminology to refer graph data types that will be used throughout this work.

A. Graph's CSR Representation and Graph Data Types

Sparse real-world graphs can be efficiently stored in memory using CSR format. Fig. 1 depicts the CSR for an example graph.

§Equal contribution

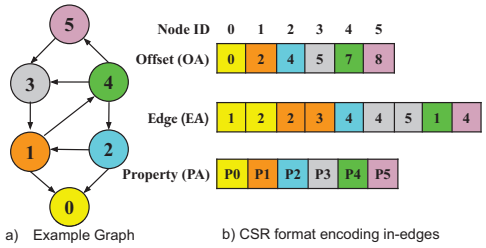

Fig. 1: A Graph and its CSR format using in-edges to a vertex

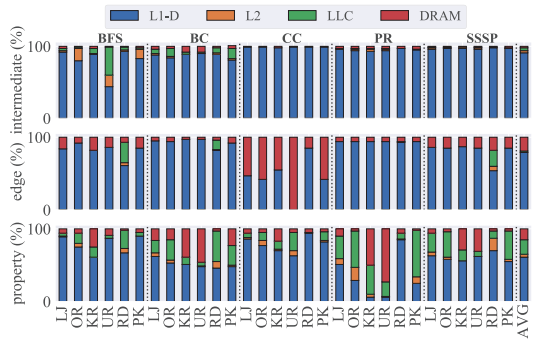

Fig. 2: Memory hierarchy performance of different data types

CSR encodes the in-edges for pull-based computations and out-edges for push-based computations. As shown in Fig. 1, the offset array (OA) maintains the start offset of a vertex's neighbors in the edge array (EA). The EA keeps each vertex's neighbors in a contiguous manner. To identify the neighbors of a vertex i, the application searches the i^{th} and $(i+1)^{th}$ entries of OA to find the range of EA indices containing neighbors of vertex i. Property[i] holds the property of a vertex i. The following terms will be used throughout this work to refer to the memory accesses generated by the respective arrays. 1) *Edge data access:* The memory accesses generated due to edge array traversal. 2) *Property data access:* The memory accesses generated due to property array traversal. 3) *Intermediate data access:* Any other memory accesses. Property data and intermediate data accesses together make non-edge data accesses.

B. Motivation

In this section, we will investigate the memory access patterns shown by the above mentioned three graph data types, namely *edge data, property data*, and *intermediate data*. We will also explore the under-utilization of the cache hierarchy by studying the interference between graph data memory accesses at different cache levels.

Cache Utilization of Different Data Types: To analyze the cache utilization of different graph data types, we profiled their cache hits at different cache levels. For profiling the cache accesses, we evaluated the graph applications on graph datasets (see Table I) mentioned in Section IV using the baseline configuration described in Table II. According to Fig. 2, ∼99% of edge data accesses do not receive any L2 or LLC cache hits (as ∼70% of edge data accesses are L1-D cache hits and ∼29% of edge data accesses result in DRAM access). Therefore, from

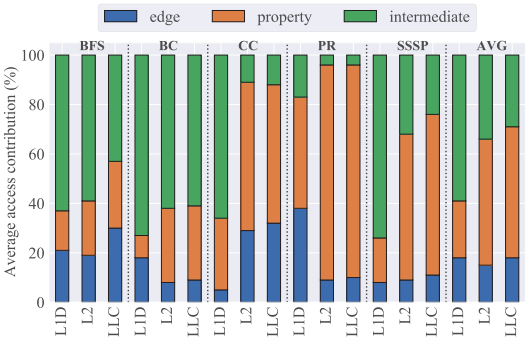

Fig. 3: Cache access profile of different data types

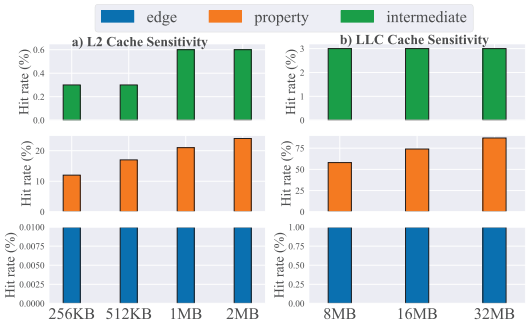

Fig. 4: Sensitivity analysis of L2 and LLC for different data types

the average bars in Fig. 2, we deduce that, *Observation* ❶: *Edge data accesses do not utilize the L2 and LLC caches.*

We recorded the number of accesses to all three graph data kinds at different cache levels to better understand the cache hierarchy utilization of property data. Fig. 3 depicts the access contribution of each data type at various cache levels. According to the average bars in Fig. 3, the number of edge data accesses across all cache levels is comparable to the number of property data accesses (the difference is not in orders of magnitude). Furthermore, as seen in Fig. 2, property data has considerable hits at the L2 and LLC cache levels. Therefore, *Observation* ❷: 1) *L2 and LLC caches do not capture the locality observed in edge data cache blocks, and* 2) *property data leverages the cache hierarchy better than edge data.*

Cache Sensitivity Analysis of Different Data Types: To investigate the sensitivity of different data types to L2 and LLC cache, we examined the performance of the *LJ* dataset on the PageRank application with varying L2 and LLC cache sizes (varying anyone while keeping the other constant). Fig. 4 shows that property data is sensitive to L2 and LLC cache size variation, while edge data is not. Therefore *Observation* ❸: *property data benefits largely from lower-level caches (L2, LLC). The L2 and LLC hit rates for property data improves as cache capacity increases.*

Interference Among Different Data Types: To investigate the cache hierarchy performance bottleneck, we performed a phase-wise analysis of the PageRank program execution on the *LJ* dataset (where the execution phase was divided into

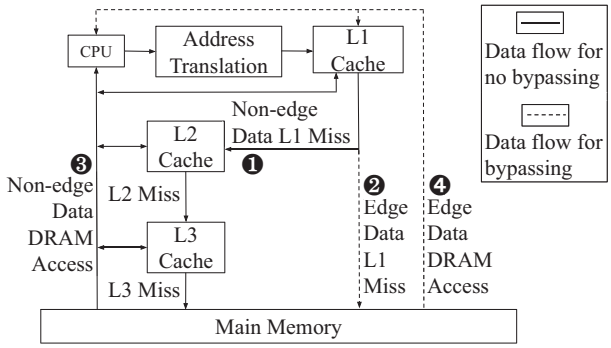

Fig. 5: GRACE Architecture

60 phases with a phase length of 10 million instructions). We recorded the number of edge data cache blocks residents (the minimum and the maximum count for each phase) in the L2 and LLC caches.

According to our findings, the cache occupancy of edge data cache blocks in L2 and LLC cache varies between 7-93% and 10-72%, respectively. As a result, edge data occupies a significant portion of L2 and LLC cache capacities. However, as demonstrated in Fig 2, they have relatively low L2 and LLC hit rates. Therefore, we infer that *Observation* ❹*: the edge data has a reuse distance greater than that served by the L2 and LLC caches. The presence of edge data in L2 and LLC is not benefiting the edge data cache accesses, thereby limiting the performance of the cache hierarchy.*

Summary: Based on the above four observations, it is evident that edge data does not benefit from the L2 and LLC caches, whereas property data benefits greatly from the L2 and LLC caches. Additionally, we identified that providing extra cache space for property data significantly increases cache hierarchy performance. Furthermore, even with minimal utilization, the edge data clogs a significant amount of L2 and LLC cache capacity. As a result, we propose to remove edge data cache blocks from the L2 and LLC cache levels to free up more cache space for property data, resulting in improved cache hierarchy utilization. The section that follows provides a detailed explanation of our proposed cache bypassing scheme.

III. GRACE Architecture

To eliminate edge data cache blocks from the L2 and LLC cache levels, we propose bypassing edge data L1 cache misses directly to the main memory. We do away with the need for L2 and LLC lookups for edge data memory requests by bypassing the edge data L1 cache misses directly to main memory. This frees up additional cache capacity in L2 and LLC for property data cache blocks. In addition, bypassing the edge data L1 misses eliminates the dynamic energy consumption for their L2 and LLC tag lookups.

To bypass the edge data that are L1 misses straight to DRAM, we propose GRACE, an efficient cache hierarchy management strategy for graph analytics, as illustrated in Fig. 5. The core accesses the L1 cache and, on an L1 cache hit, receives the required data from the cache. However, there are two conceivable outcomes in the event of an L1 cache miss.

1) L1 cache miss for non-edge data: The request follows the traditional memory subsystem access flow i.e. through L2 and LLC caches (shown as ❶ and ❸ in Fig. 5).
2) L1 cache miss for edge data: The request bypasses the L2 cache and LLC and accesses the main memory directly(shown as ❷ in Fig. 5). The cache block brought from main memory is directly place in L1 cache (shown as ❹ in Fig. 5), again bypassing L2 and LLC caches.

A. Identification of Edge Data

To identify edge data, we modified the operating system and the graph processing framework. The graph processing framework consists of a graph data allocation layer. We modified the functionality of $malloc$ to identify the edge data at the graph data allocation layer as implemented in prior work [3]. The functionality of $malloc$ is modified to facilitate populating a bit in page permission field of every page at page table. The modified $malloc$ populates this extra bit as *logic* '1' for edge data and *logic* '0' for non-edge data. Since we already have a few unused bits available (for *X86 ISA*) at the page permission field [7], we need not add an extra bit at the page table to accommodate the same.

To identify the data type of the cache block being evicted at the L1 cache, we add a bit for every cache block at the L1 tag array. The bit gets populated when a cache block is inserted in the L1 cache. When we evict a cache block at the L1 cache, it bypasses the L2 and the LLC cache levels and is directly sent to the main memory in case of edge data identified using the added bit. The added bit is decoded in the same way as the extra bit populated at the page table.

B. Cache Coherency Challenges

The *edge data are read-only* type for static graphs. Therefore, the cache blocks that contain edge data will not be modified by any core. There is a definite possibility of a core *(e.g., C1)* requiring the edge data that is already present in the private cache of core *(e.g., C0)*. In a conventional scenario, the *core C1* would receive an LLC cache hit for the requested cache block. However, in our proposed scenario, the *core C1* would not find the requested cache block in the LLC. Thus, *core C1* will acquire the cache block from the main memory eliminating the need to track the cache blocks containing edge data at the tag directory to maintain cache coherency.

IV. Experimental Framework

This section explains the experimental framework used for our studies and performance evaluation of GRACE.

A. Benchmark Applications and Graph Datasets

For evaluating GRACE, we used the GAP benchmark application suite [8]. It consists of multi-threaded C++ implementation of a few representative algorithms used in graph analytics. These implementations do not have any framework-related runtime overheads, and hence they help us to determine the actual hardware bottlenecks [3]. We simulate five algorithms from the suite namely Breadth First Search (*BFS*), Betweenness

TABLE I: Graph Datasets

Dataset	#V	#E	Size	Description
livejournal (LJ) [10]	4.8M	68.5M	597MB / 1.1GB	Social network
orkut (OR) [10]	3M	117M	941MB / 1.8GB	Social network
kron (KR) [8]	16.8M	260M	2.1GB / 2GB	Synthetic
urand (UR) [8]	8.4M	134M	1.1GB / 2.1GB	Synthetic
road (RD) [8]	23.9M	57.7M	806MB / 1.3GB	Mesh network
pokec (PK) [10]	1.6M	30.6M	271MB / 516MB	Social network

TABLE II: Baseline System Configuration

Core	4 cores, ROB = 128 entry, LQ = 48 entry, SQ = 32 entry, RS = 36 entry, Freq. = 2.66 GHz Dispatch width = Issue width = Commit width = 4.
Caches	3 level inclusive hierarchy, 64B cacheline, Writeback & LRU replacement policy, Data/Tags parallel access, Separate L1 I/D cache.
L1-I/D Cache	Private, Size = 32 KB, Associativity = 8, Tag latency = 1 cycle, Data latency = 4 cycles
L2 Cache	Private, Size = 256 KB, Associativity = 8, Tag latency = 3 cycles, Data latency = 8 cycles
L3 Cache (LLC)	Shared, Size = 8 MB, Associativity = 16, Tag latency = 10 cycles, Data latency = 30 cycles
RAM (Main memory)	DDR3, Access latency = 45 ns, Queue delay modeled

Centrality (*BC*), Connected Components (*CC*), Page Rank (*PR*), and Single Source Shortest Path (*SSSP*). We select these algorithms as they are the primitive graph algorithms used to design more complex algorithms [9]. Table I lists the graph datasets that are used in our experimental framework. The size field in Table I shall be interpreted as sizes for unweighted/weighted graph.

B. Applications Profiling

We profile the applications to find their region of interest (ROI). *ROI* is defined as the execution phase that contributes the most to the memory subsystem bottleneck during the entire execution of an application. Since our proposed idea intends to reduce the memory subsystem bottleneck, we extract the value of different micro-architectural parameters for this *ROI* and examine the impact of our proposed scheme on the system's performance. We identify the *ROI* for each application using Intel's VTune profiler [11]. We used *hotspots* identifier by the profiler to determine the *ROI*.

C. Simulation Environment

We evaluated our proposed idea using *Sniper* simulator that is based on *x86 ISA* [12]. *CACTI* is used to model the cache access timings for different cache capacities [13]. Table II shows the baseline system configuration used for evaluation purposes. After marking the *ROI* for each application, we simulated the applications on the Sniper simulator. We collected the necessary statistics for 600 million instructions across all the cores after entering the *ROI*, similar to prior work [3].

V. Performance Evaluation of GRACE

We compare GRACE with prior studies to examine its cache utilization and demonstrate that it significantly enhances performance while reducing off-chip accesses. We consider DRRIP as our baseline LLC replacement policy because DRRIP outperforms LRU replacement policy [14].

A. Performance Analysis

In terms of speed-up, Fig. 6 demonstrates that GRACE outperforms DRRIP by 16% (on average). In the case of PR application for all datasets, GRACE surpasses DRRIP by 12% (on average). The performance gains achieved by GRACE is attributed to its improved cache hierarchy utilization. It utilizes the cache hierarchy by reducing the LLC MPKI across all applications and datasets by 65% (on average), compared to DRRIP (as shown in Fig. 6), with a maximum reduction of 92% (over DRRIP) observed for BFS with *Orkut* dataset. It also reduces L2 MPKI by 28% (on average) when compared to DRRIP, with a significant reduction of 55% (over DRRIP) for BFS application (average over simulated datasets). GRACE reduced the L2 and LLC MPKIs by devoting more cache space to property data while eliminating edge data cache blocks from the L2 and LLC caches.

In the case of *road* dataset and *BC* program combinaton, the DRAM APKI due to cache bypassing in GRACE and GRACE++ is significantly higher than that in GRASP. As a result, although GRACE and GRACE++ have lower L3 MPKI than GRASP for the combination, they perform worse than GRASP due to the performance penalty of off-chip memory accesses. In the next section, we do a detailed comparison of GRACE to GRASP, a recently proposed cache replacement policy for graph analytics.

B. Comparison with GRASP

GRASP is a cache replacement policy designed specifically for graph analytics that utilizes graphs' skewed degree distribution. It requires a pre-processed vertex array as input. The vertices are sorted using degree-based grouping (DBG) before processing. It uses DRRIP to manage the LLC. To compare GRACE and GRASP fairly, we reordered our graphs using DBG. We refer this augmentation of GRACE as GRACE++.

Fig. 6 shows that GRACE++ outperforms GRASP in terms of performance by 8% (on average). It outperforms GRASP by 1.4× for BC application and *kron* dataset. GRACE++ improves system's performance by decreasing the LLC MPKI by 49% as compared to GRASP across all applications (as shown in Fig. 6, with maximum LLC MPKI reduction of 88% for PR application with pokec dataset. It additionally reduces the L2 MPKI by 18% (on average) as compared to GRASP, thereby efficiently utilizing the cache hierarchy. When compared to GRASP, GRACE++ also minimizes the off-chip memory traffic. To measure off-chip memory traffic, we record the DRAM APKI for all simulation runs. Fig. 7 shows that GRACE++ also reduces off-chip memory traffic by 13% (on average) compared to GRASP, with maximum reduction of 27% relative to GRASP for SSSP application. The following section examines the effect of GRACE and GRACE++ on cache hierarchy performance.

C. Cache Performance Analysis

GRACE is a cache hierarchy management strategy proposed to improve cache hierarchy utilization by bypassing edge data L1 misses directly to DRAM, resulting in additional L2 and

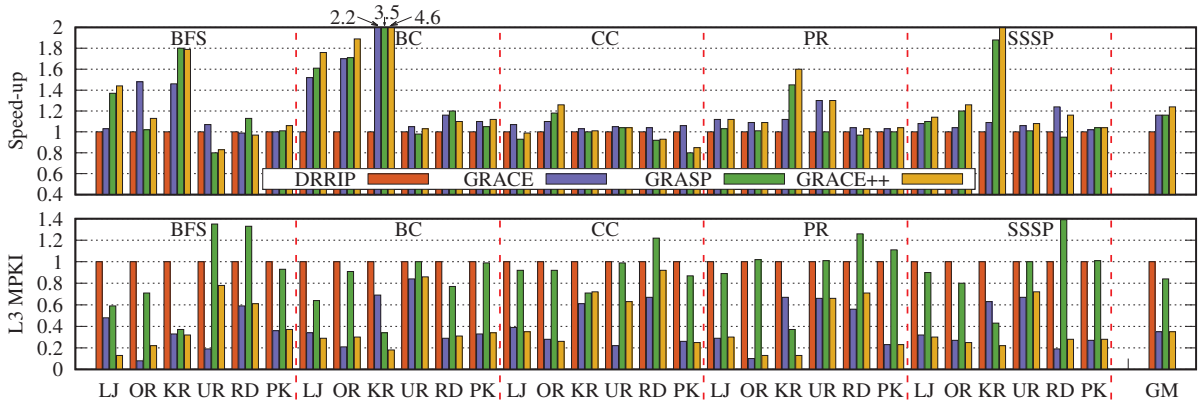

Fig. 6: Performance comparison of GRACE and GRACE++: Speedup (top), L3 MPKI (bottom)

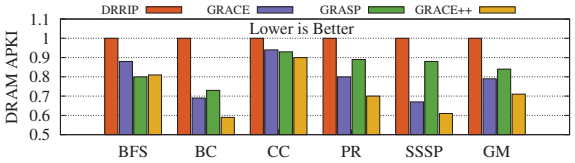

Fig. 7: Comparison of DRAM APKIs

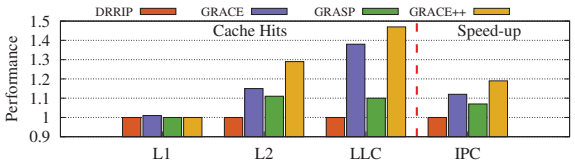

Fig. 8: Performance of GRACE and GRACE++

LLC cache capacity for property graph data type. We record hit rates of different cache levels to measure their respective performance. The performance of different cache levels and the corresponding speedup gains for PageRank application across datasets are depicted in Fig. 8. According to Fig. 8, GRACE or GRACE++ have no detrimental effect on L1 cache performance when compared to the baseline (DRRIP). Fig. 8 further demonstrates that GRACE improved the performance of L2 and LLC caches by 15% and 38%, respectively. GRACE++ enhanced L2 and LLC cache performance by 29% and 47%, respectively, due to the impact of DBG.

Therefore, from Fig. 8, we conclude that bypassing edge data from L2 and LLC caches helps the system to better utilize the cache hierarchy, resulting in enhanced system performance of 12% and 19% (relative to DRRIP) for GRACE and GRACE++, respectively (for PageRank application). Fig. 8 also shows that the cache performance of different schemes translates to their system performance. The following section discusses the effect of different LLC sizes on GRACE and GRACE++.

D. LLC Sensitivity Analysis

Although GRACE and GRACE++ improve cache utilization, as discussed in the last section, we investigated their sensitivity to different LLC sizes. Furthermore, to examine the possibility of shrinking the size of large LLC caches (by reducing the cache capacity), we evaluated GRACE++ with 2MB, 4MB, 8MB and 16MB LLC cache sizes for the *LJ* dataset across all applications. To perform a comparative analysis, we also evaluate GRASP and GRACE with 8MB and 16MB LLC cache sizes. Fig. 9 shows the effect of different LLC capacities on system's performance.

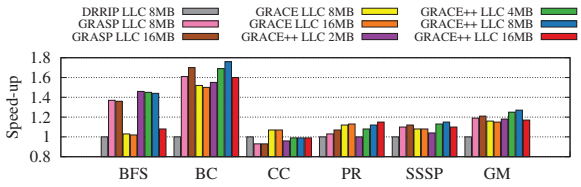

Fig. 9: LLC Sensitivity Analysis of GRACE

According to our studies (shown in Fig. 9), GRACE (with 8MB and 16MB LLC) outperforms DRRIP (with 8MB LLC) by 16% and 15% respectively. GRACE++ (with 2MB, 4MB, 8MB and 16MB LLCs) betters DRRIP (with 8MB LLC) by 18%, 25%, 27% and 17% respectively (on average). GRASP (with 8MB and 16MB LLC) outmatches DRRIP (with 8MB LLC) by 19% and 21% respectively. Our study demonstrates for GRACE++ a $(1/4)th$ sized LLC, as well as a half-sized LLC improves performance (relative to DRRIP). Therefore, a smaller LLC cache size may be employed to reduce the cache hierarchy's area and energy consumption while still offering enhanced performance benefits comparable to GRASP (with 8MB or 16MB LLC). The next section examines the orthogonality of previously proposed cache replacement policies to our proposed scheme, GRACE.

E. Cache Bypassing as a Complementary Technique

GRACE is a graph data-aware cache hierarchy management strategy that avoids edge data from L2 and LLC because edge data limits the utilization of L2 and LLC caches. GRACE outperforms DRRIP by 16% (as shown in Fig. 6), and it delivers a comparable speedup when compared to GRASP. Prior efforts such as DRRIP, SHiP, Hawkeye, GRASP, and P-OPT are cache replacement strategies attempting to select the optimal eviction candidate to extract the maximum possible locality, optimizing cache utilization. GRACE, on the other hand, is a policy that determines whether or not to insert a cache block in a specific cache level. Once the data is in the cache, the replacement policy manages the eviction of cache blocks in the event of a

cache miss. Therefore, we argue that advanced previous cache replacement policies would further boost system performance and complement our proposed scheme.

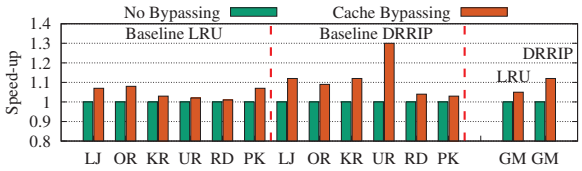

Fig. 10: GRACE as a complementary technique over LRU & DRRIP

To back up our assertion, we investigated the effect of LRU and DRRIP on the PageRank's performance with and without cache bypassing. According to Fig. 10, with cache bypassing (i.e. GRACE), we observe performance improvements of 5% and 12% for both LRU and DRRIP, respectively (compared to no-bypassing). Therefore, Fig. 10 demonstrates that DRRIP, an advanced strategy above LRU, when implemented along with *GRACE* enhances its performance further, therefore confirming our assertion.

VI. Related Work

We examine several prior approaches for optimizing the memory subsystem in CPU-based systems for graph analytics.

Cache replacement policy-based optimizations: Prior works, such as SHiP [5] and Hawkeye [6], developed replacement strategies suited for a wide range of workloads. They outperform other replacement strategies, including DRRIP [14], SBDP [15], and Leeway [16]. However, prior work [1] shows that the highly irregular access pattern of graph workloads has reduced their effectiveness for graph analytics.

Graph-aware cache replacement policies: GRASP [4] identified a few high-degree vertices due to the power-law distribution of real-world graphs, indicating that such vertices are highly reused. It proposed a cache replacement policy to avoid eviction of such vertices. It outperforms DRRIP [14], SHiP [5], and Hawkeye [6]. However, GRASP is ignorant of various graph data types. P-OPT [1] found that the transpose of a graph's adjacency matrix records subsequent references of vertices during graph's execution. It proposed a cache replacement policy to utilize above re-reference hints and improve the cache hierarchy's performance.

Prefetcher-based optimizations: DROPLET [3], HATS [17], and Prodigy [18] proposed prefetchers that can handle the irregularity of memory accesses in graph processing. While these approaches effectively reduce average memory access latency, they do not reduce main memory traffic, a key focus of our work. Moreover, as an inference from our studies, such techniques still under-utilize the cache hierarchy due to allocation of L2, LLC to edge-data.

VII. Conclusion

Modern processors' increased main memory capacity and core count have opened up the possibility of single-machine graph analytics. However, large input graph datasets seldom fit in a single-machine's cache hierarchy, making its memory subsystem performance a bottleneck. Hence, effective cache management is critical for such systems. We argue that modern cache management techniques are ineffective for graph analytics as they are oblivious to the locality demands of different graph data types. Our work finds that different levels of cache hierarchy is suitable for the locality demands of different data types. We show that edge data does not utilize the L2 and LLC effectively, and its presence in these caches adversely impacts the performance of cache hierarchy. Hence, our proposed cache management scheme, GRACE, bypasses the edge data accesses from L2 and LLC to improve the cache performance of other graph datatypes. We demonstrate that considering both cache interference and cache sensitivity of these data types is critical for high-performance graph analytics. Our comprehensive assessment shows that GRACE++ (GRACE augmented with DBG), outperforms the recent cache management scheme, GRASP, by up to 1.4× while reducing costly DRAM accesses by up to 27%. Therefore, GRACE demonstrates the benefits of bringing graph data awareness to cache management approaches for single-machine graph analytics. In the future, we will look into the energy consumption of the cache hierarchy. We would also investigate the influence of GRACE on various methods of storing graph data in order to improve the memory subsystem's efficiency.

References

[1] V. Balaji, N. Crago, A. Jaleel, and B. Lucia, "P-OPT: Practical Optimal Cache Replacement for Graph Analytics," in *HPCA*, 2021.

[2] 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/, 2017.

[3] 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 *HPCA*, 2019.

[4] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *HPCA*, 2020.

[5] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *MICRO*, 2011.

[6] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *ISCA*, 2016.

[7] "Intel®64 and IA-32 Architectures Software Developer's Manual, Vol. 3A," 2016.

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

[9] 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*, 2016, pp. 1813–1828.

[10] J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection," 2014.

[11] J. Reinders, "VTune performance analyzer essentials," *Intel Press*, 2005.

[12] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *TACO*, 2014.

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

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

[15] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *MICRO*, 2010.

[16] P. Faldu and B. Grot, "Leeway: Addressing variability in dead-block prediction for last-level caches," in *PACT*, 2017.

[17] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *MICRO*, 2018.

[18] N. Talati, K. May, A. Behroozi, Y. Yang *et al.*, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *HPCA*, 2021.