Exploring Core and Cache Hierarchy Bottlenecks in Graph Processing Workloads

Abanti Basak, Xing Hu, Shuangchen Li, Sang Min Oh, and Yuan Xie

Abstract—Graph processing is an important analysis technique for a wide range of big data problems. The ability to explicitly represent relationships between entities gives graph analytics significant performance advantage over traditional relational databases. In this paper, we perform an in-depth data-aware characterization of graph processing workloads on a simulated multi-core architecture, find bottlenecks in the core and the cache hierarchy that are not highlighted by previous characterization work, and analyze the behavior of the specific application data type causing the corresponding bottleneck. We find that load-load dependency chains involving different application data types form the primary bottleneck in achieving a high memory-level parallelism in graph processing workloads. We also observe that the private L2 cache has a negligible contribution to performance, whereas the shared L3 cache has higher performance sensitivity. In addition, we present a study on the effectiveness of several replacement policies. Finally, we study the relationship between different graph algorithms and the access volumes to the different data types. Overall, we provide useful insights and guidelines toward developing a more optimized CPU-based architecture for high performance graph processing.

Index Terms—Graph Processing, Memory-Level Parallelism, Cache Hierarchy

1 Introduction

Graph processing is widely used to solve big data problems in multiple domains such as social networks, webgraph hierarchies, protein interactions in bioinformatics, and transportation. The ubiquitousness 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).

Existing work in graph processing spans a variety of hardware platforms. Numerous software frameworks propose programming abstractions to parallelize graph processing in clusters [1]. Another category of work proposes external storage based graph processing in a single machine [2]. The goal of this paper is to improve the CPU platform where the entire graph data fits in the RAM of a single high-end server. In such a platform, the primary bottleneck is the latency gap between the on-chip caches and the DRAM, which leads to stalling as the cores wait for data to be fetched from the memory. We focus on single machine in-memory graph analytics because it has been shown to provide excellent performance while significantly reducing the programming efforts compared to distributed systems [3]. Moreover, a wide range of industry and academic graphs have been reported to fit comfortably in the memory of a single machine [3], [4].

To understand the memory subsystem bottlenecks in single-machine in-memory graph analytics, we characterize 1) the *memory-level parallelism (MLP) in an out-of-order (OoO) core* and 2) the *cache hierarchy*. Our characterization fills the gaps in prior profiling work [5], [6] in two aspects. First, we perform a data structure aware profiling which provides clearer guidelines on the management of different data types for performance optimization. Second, with the flexibility of a simulated platform, we vary the instruction window and the cache configuration design parameters to explicitly explore their performance sensitivity. Prior work does not provide this level of detail. Based on our observations from the analysis, we provide possible guidelines for a better memory hierarchy architecture. Beyond previous profiling work, our key observations and insights toward architecture design are summarized below:

- Load-load dependency chain involving specific application data types, rather than the instruction window size limitation, is 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, 3) the data type representing the graph property benefits the most from a larger shared L3 cache, and 4) any cacheline in the recency stack other than the most recently used (MRU) cacheline is a viable candidate for eviction.
- Graph algorithms can be classified into two categories on the basis of the memory access volumes to different data types. This helps identify the target data type for performance optimization in different algorithms.

2 Methodology

In this section, we describe the experiment setup and the benchmarks used. Since graph data layout is an important factor in determining memory access patterns, we also discuss the layout used by our benchmark.

2.1 Profiling Platform

The experiments have been done using SNIPER simulator [7]. Cache access timings for different cache capacities were extracted using CACTI [8]. The baseline architecture is described in Table 1. We used fewer cores than typically present in a server node because previous profiling work [5] has shown that resource utilization for parallel and single-core executions is similar. 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 the cache warm-up mode and, upon entering the ROI, collected statistics for 600 million instructions across all the cores.

2.2 Benchmark

We used the GAP benchmark [9] which consists of a set of optimized multi-threaded implementations of some of the most widely used algorithms in graph analytics. We selected GAP because it is not a framework. Hence, it allows us to rule out any framework-related overheads and extract true hardware bottlenecks. The five graph algorithms we used are Betweenness Centrality (BC), Breadth First Search (BFS), PageRank (PR), Single Source Shortest Path (SSSP), and Connected Components (CC).

A summary of the datasets used for our characterization is shown in Table 2. The datasets have been chosen such that we achieve a good coverage of input graph types both in terms of their application domains and generation methods (real/synthetic). In addition, the memory footprint of the datasets has been chosen to be small enough to achieve manageable simulation times but large enough to stress the memory system sufficiently.[1] The size on the left is that of the unweighted graph whereas that on the right is that of the weighted graph (required for SSSP).

• *The authors are with the University of California, Santa Barbara, CA 93106. E-mail: {abasak, sangminoh}@umail.ucsb.edu, {xinghu.cs, yuanxie}@gmail.com, shuangchenli@ece.ucsb.edu.*

Manuscript received 19 June 2018; revised 10 July 2018; accepted 29 July 2018. Date of publication 23 Aug. 2018; date of current version 10 Sept. 2018. (Corresponding author: Abanti Basak.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2018.2864964

1. Our conclusions still hold for larger graphs because we explain the observed architecture bottlenecks in terms of inherent data type features and algorithm characteristics which are independent of the data size.

TABLE 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 = 2.66 GHz
caches	3-level hierarchy, inclusive at all levels, writeback, LRU (Least Recently Used) replacement policy, data and tags parallel access, 64B cacheline, separate L1 data and instruction caches
L1D cache	private, 32 KB, 8-way set-associative, data access time = 4 cycles, tag access time = 1 cycle
L2 cache	private, 256 KB, 8-way set-associative, data access time = 8 cycles, tag access time = 3 cycles
L3 cache (LLC)	shared, 8 MB, 16-way set-associative, data access time = 30 cycles, tag access time = 10 cycles
DRAM	DDR3, device access latency = 45 ns, queue delay modeled

TABLE 2
Datasets

Dataset	vertices	edges	Memory Footprint	Description
kron [9]	16.8M	260M	2.1 GB/2 GB*	synthetic
urand [9]	8.4M	134M	1.1 GB/2.1 GB	synthetic
orkut [10]	3M	117M	941 MB/1.8 GB	social network
livejournal [10]	4.8M	68.5M	597 MB/1.1 GB	social network
road [9]	23.9M	57.7M	806 MB/1.3 GB	mesh network

**Weighted graph is generated from a smaller degree (hence smaller size) for a manageable simulation time.*

2.3 Graph Data Layout in the Benchmark

GAP uses the Compressed Sparse Row (CSR) representation which is the most widely used data layout for graphs because of its efficient memory space usage. As shown in Fig. 1, CSR has three main components: the offset pointer, 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. The vertex data array stores the property of each vertex and is indexed by the vertex ID. In the rest of the paper, we use the following terminology: 1) *Structure data*: the neighbor IDs array 2) *Property data*: the vertex data array 3) *Intermediate data*: any other data.

3 Observations and Analysis

In this section, we begin by analyzing the effectiveness of a larger instruction window in an OoO core in increasing the MLP. Next, we analyze the cache hierarchy to 1) find the relative usefulness of each cache level and 2) study the impact of different replacement policies. Finally, we observe how different algorithms interact differently with structure, property, and intermediate data.

3.1 Analysis of the Core and MLP

Instruction window size is not the factor impeding MLP: Previous profiling work on a real machine [5] concludes that the reorder buffer (ROB) size is the bottleneck in achieving a high MLP and a high memory bandwidth utilization for graph analytics workloads. 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. 2a, for a 4x instruction window, the average increase in memory bandwidth utilization is only 2.7 percent. Fig. 2b shows the corresponding speedups. The mean speedup is only 1.44 percent, which is very small compared to the large allotted hardware resources.

Load-load dependency chains prevent achieving high MLP: To understand why a larger ROB does not improve MLP, we track

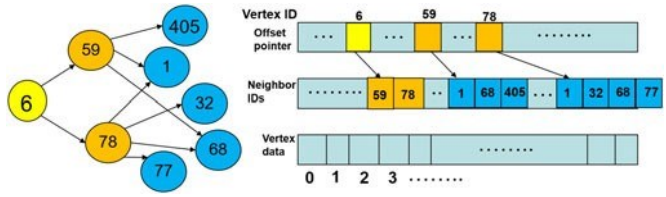

Fig. 1. CSR data layout for graphs.

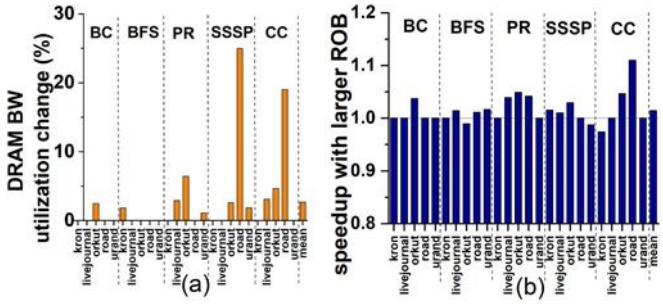

Fig. 2. (a) change in DRAM bandwidth utilization and (b) overall speedup from a 4x larger ROB.

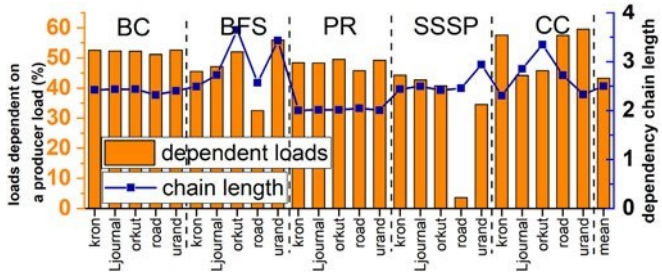

Fig. 3. Load-load dependency in ROB.

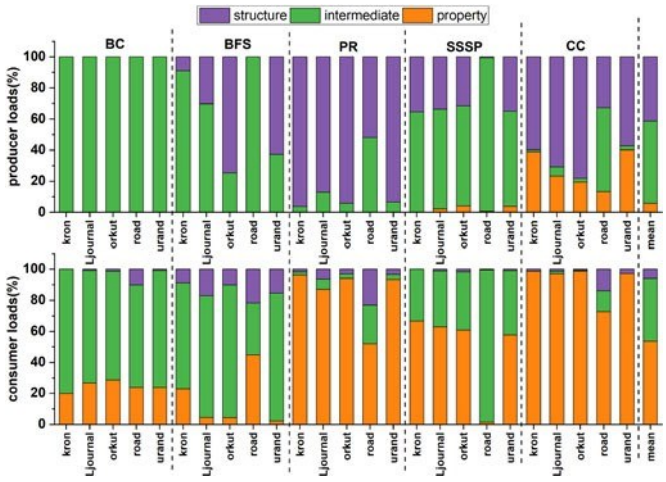

Fig. 4. Breakdown of producer and consumer loads by application data type.

dependencies of the load instructions in the ROB and find that 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 load and the younger load a consumer load. We find that these 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 shows that, on average, 43.2 percent of the loads are part of a load-load dependency chain with an average chain

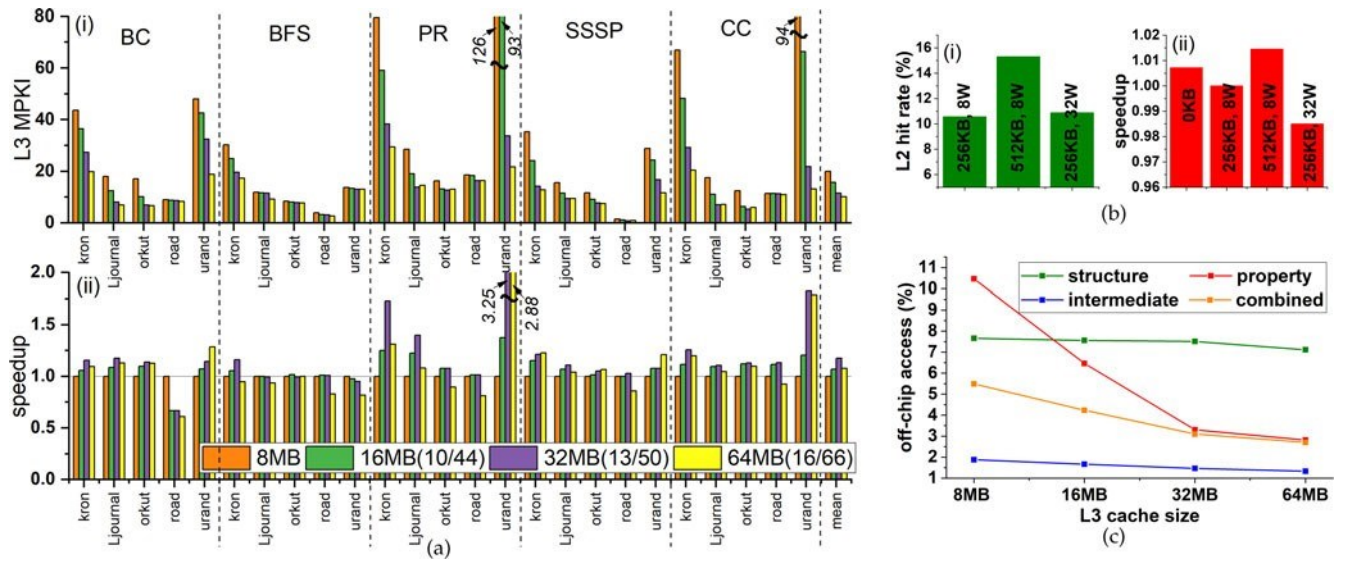

Fig. 5. (a) Sensitivity of i) L3 misses per kilo instructions (MPKI) and ii) system performance to shared L3 cache size (access times for (tags/data) in cycles); (b) Sensitivity of (i) L2 hit rate and (ii) system performance to private L2 configurations (average across all benchmarks); (c) Effect of larger L3 cache on off-chip access of different data types (average across all benchmarks).

length of 2.5, where we define the chain length as the number of instructions in the dependency chain.

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. 4. We find that the graph property data is mostly the consumer in such a chain (average 53.6 percent consumers in contrast to only 5.9 percent producers). The issuing of a graph property data load is delayed and cannot be parallelized because it has to depend on a producer load for its address calculation. Fig. 4 also shows that structure data is mostly the producer (average 41.4 percent producers as opposed to only 6 percent consumers).

3.2 Analysis of the Cache Hierarchy

3.2.1 Performance Sensitivity of L2 and L3 Caches

Private L2 cache shows negligible performance sensitivity, whereas shared LLC shows higher performance sensitivity: As shown in Fig. 5a, we vary the LLC size from 8 to 64 MB and find the optimal point of 17.4 percent (max 3.25x) performance improvement for a 4x increase in the LLC capacity. This optimal point is a balance between a reduced miss rate and a larger LLC access latency. For the private L2 cache, Fig. 5b shows that the L2 hit rate (which is already very low at 10.6 percent in the baseline) and the speedup show little sensitivity to the different L2 configurations (both capacity and set associativity). The leftmost bar in Fig. 5b(ii) represents an architecture with no private L2 caches and no slowdown compared to a 256 KB cache. Therefore, an architecture without private L2 caches is just as fine for graph processing.

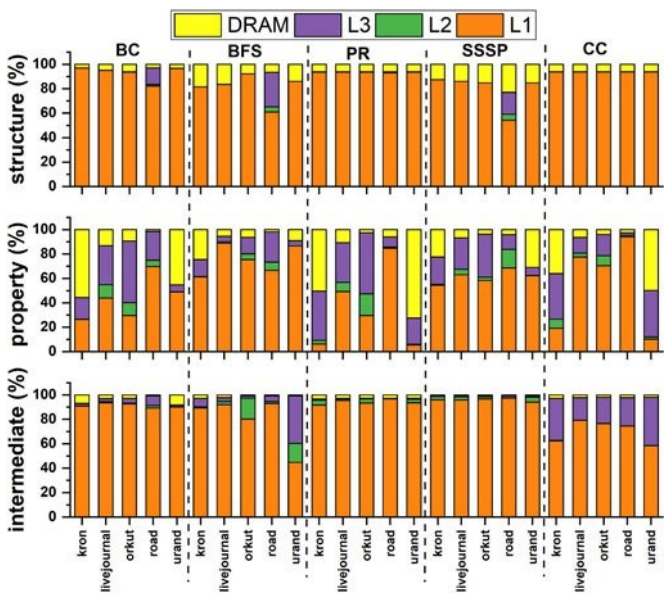

Fig. 6. Breakdown of cache and memory usage by application data type.

Property data is the primary beneficiary of LLC capacity: To understand which data type benefits from a larger LLC, Fig. 5c 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 off-chip accesses comes from the property data. Structure and intermediate data do not benefit from a higher capacity. Intermediate data is already mostly accessed in on-chip caches since only 1.9 percent of the accesses to this data type are DRAM-bound in the baseline. Structure data, on the other hand, has a higher percentage of off-chip accesses (7.5 percent) which remains mostly irresponsive to a larger LLC capacity.

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 L2 and L3, we break down the memory hierarchy usage by application data type in Fig. 6. In most benchmarks,[2] 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 the 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. 5c. 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 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. 6 provides evidence that the accesses to intermediate

2. The *road* dataset with a very large diameter [9] results in an extreme case or an exception in some experiments.

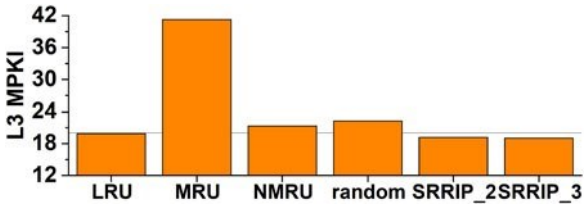

Fig. 7. LLC performance for different replacement policies (average across all benchmarks). NMRU=not most recently used, SRRIP_[2,3]=static re-reference interval prediction [11].

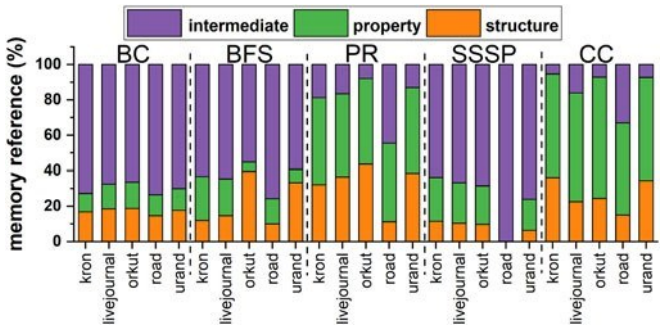

Fig. 8. Memory reference breakdown by data type.

data are mostly on-chip cache hits in the L1 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 for performance.

3.2.2 Effect of Replacement Policies on LLC

We further investigate whether there is opportunity for improvement through alternative replacement policies in the LLC. Fig. 7 reveals a skewness in the victim selection policy in the recency stack. *Any cacheline other than the MRU is a viable candidate for eviction.* Compared to the baseline LRU, MRU shows a significantly worse cache performance (higher MPKI), whereas the other policies show performance similar to LRU. This implies that a MRU cacheline has short re-reference intervals. On the other hand, the cachelines in any other recency position are not likely to be re-referenced soon. Hence, their eviction minimally affects the MPKI.

3.3 Relation Between Algorithms and Data Types

To understand how each algorithm interacts with different data types, Fig. 8 shows the breakdown of the memory reference (loads and stores) volume by structure, property, and intermediate data. We find that *for PR and CC, the data types with the largest access volumes are structure and property data. On the other hand, in BC, BFS, and SSSP, intermediate data occupies the largest share of the access volume.* This variation can be explained by how each algorithm schedules vertices for processing. In PR and CC style algorithms, vertices are scheduled sequentially. For each selected vertex, the access pattern is to first access the structure data which is then used to index the property data. In BC/BFS/SSSP, on the other hand, vertices are scheduled selectively according to multiple bookkeeping data structures such as queues, bins, or worklists, explaining their large access volume to intermediate data.[2]

4 Conclusions and Architectural Guidelines

This paper performs a data-aware characterization on a simulated multi-core architecture in order to study the bottlenecks in the core and the cache hierarchy in graph analytics. Our analysis provides opportunities and guidelines for architecture optimizations. First, to resolve dependency chain bottlenecks, techniques such as dependence graph based prefetching [12] or near-memory acceleration of dependency chains [13] could be utilized. Second, different impacts of the L2 and the L3 caches on the system performance provide multiple directions for improving the overall cache hierarchy. Private L2 caches could be reconfigured to be part of the LLC. The L2 cache could also be used as the perfect cache level for prefetching without risk of cache pollution. Third, an optimization technique targeting a specific data type should be aware of the relative importance of the data type in the spectrum of graph algorithms. A technique targeting structure or property data may benefit CC and PR significantly, whereas benefits could be reduced for BC, BFS, and SSSP-style algorithms where intermediate data absorbs the majority of the access volume.

Acknowledgments

This work was supported in part by US National Science Foundation 1730309/1719160/1500848 and by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation program sponsored by DARPA.

References

[1] 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 *Proc. Int. Conf. Manag. Data*, 2014, pp. 979–990.

[2] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 31–46.

[3] Y. Perez, R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec, "Ringo: Interactive graph analytics on big-memory machines," in *Proc. Int. Conf. Manage. Data*, 2015, pp. 1105–1110.

[4] Z. Shang, F. Li, J. X. Yu, Z. Zhang, and H. Cheng, "Graph analytics through fine-grained parallelism," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 463–478.

[5] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an Ivy bridge server," in *Proc. Int. Symp. Workload Characterization*, 2015, pp. 56–65.

[6] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, and S. Katti, "Parallel graph processing on modern multi-core servers: New findings and remaining challenges," in *Proc. Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2016, pp. 49–58.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 52:1–52:12.

[8] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0," HP Laboratories, Palo Alto, CA, USA, Tech. Rep. HPL-2009-85, 2009.

[9] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015, http://arxiv.org/abs/1508.03619

[10] J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection," Jun. 2014. [Online]. Available: http://snap.stanford.edu/data

[11] A. Jaleel, K. B. Theobald, S. C. Steely Jr , and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. Int. Symp. Comput. Archit.*, 2010, pp. 60–71.

[12] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data prefetching by dependence graph precomputation," in *Proc. Int. Symp. Comput. Archit.*, 2001, pp. 52–61.

[13] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Accelerating dependent cache misses with an enhanced memory controller," in *Proc. Int. Symp. Comput. Archit.*, 2016, pp. 444–455.