Slow is Fast: Rethinking In-Memory Graph Analysis with Persistent Memory

Hanyeoreum Bae[1], Miryeong Kwon[1], Donghyun Gouk[1], Sanghyun Han[1],
Sungjoon Koh[1], Changrim Lee[1], Dongchul Park[2], and Myoungsoo Jung[1]

Computer Architecture and Memory Systems Lab,
[1]Korea Advanced Institute of Science and Technology (KAIST), [2]Sookmyung Women's University
http://camelab.org

I. Introduction

Efficient computing for large-scale graph data is central to a broad spectrum of data-intensive applications [1]. To achieve high performance with low communication and scalability costs, many studies adopt in-memory graph systems [2]. However, without careful consideration of system-level characteristics, graph analysis with large datasets can face a severe challenges such as out-of-memory error and unreasonable long latency for its data processing.

In this paper, we explore and uncover the challenges that in-memory graph processing suffers from. Our system-level analysis includes empirical results opposite to the existing expectations of graph application users. Specifically, since raw graph data are not the same as the in-memory graph data, processing a billion-scale graph easily exhausts all system resources and makes the target system unavailable due to out-of-memory at runtime.

To address this lack of memory space problem, we configure real *persistent memory devices* (PMEMs) with different operation modes and system software. We then introduce PMEM to a representative in-memory graph system, Ligra [3], and reveal the performance behaviors of different PMEM-applied in-memory graph systems. Based on our observations, we modify Ligra to improve the performance with a solid level of data persistence. Our evaluation results reveal that our modified Ligra exhibits 4.41× and 3.01× better performance than the original Ligra running on a virtual memory expansion and conventional persistent memory, respectively.

II. Motivation for Persistent Memory

Graphs for in-memory. Figure 1 briefly shows the overall process of in-memory graph analysis. All in-memory graph frameworks first require loading the data, called *raw graph*, from the underlying storage to the working memory (❶) and convert the raw graph to in-memory graph structures (❷). These two pre-processes for the graph analysis are referred to as *graph construction*. This graph construction is generally performed to change spatial-efficient format of graph files into in-memory data structures, which contain extra information for efficient graph processing. Once the graph construction process completes, the frameworks can execute graph algorithm(s) to process the data in memory (❸). In the execution phase, the frameworks are often required to be fault-tolerant as well. This is because there are several quadratic and cubic graph algorithms taking the time longer than a mean time between failures [4]. To this end, in-memory graph systems can employ a checkpoint-restart, a method that periodically stores the copy of metadata and progress reports persistently [5].

Persistent memory. Intel Optane PMEM is recently released in the market as a byte-addressable NVM technology. The current technology that PMEM adopts offers a capacity 4× greater than DRAMs on a single server. Specifically, PMEM is a complex system-level solution, including 3D-Xpoint, DRAM, and system software to support a large, persistent memory subsystem. PMEM provides two operation modes; i) *memory mode* and ii) *app-direct mode*.

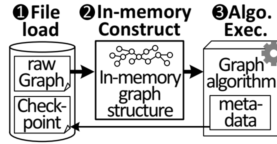


Fig. 1: Procedure of graph framework

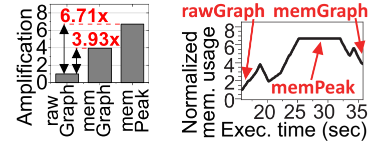


Fig. 2: Data amplification.

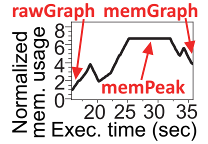


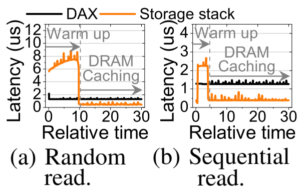
Fig. 3: Time series analysis.

If the memory mode is enabled, local-DRAM is used as a direct-mapped write-back cache for 3D-Xpoint modules. The users can thus experience DRAM-like performance and access PMEM via the conventional memory instructions (e.g., load/store). However, as the memory mode PMEM does not guarantee data persistence, it needs to utilize additional slow storage devices (e.g. SSDs) to store data persistently.

On the other hand, PMEM configured with the app-direct mode splits the memory space into a DRAM space and a 3D-Xpoint storage space. With this mode, the users can experience storage-like data persistence by accessing the 3D-Xpoint space. Unfortunately, it degrades overall performance by 73%, compared to conventional system (using DRAM). This is because PMEM in the app-direct mode cannot utilize the fast performance of local-DRAM. In addition, since app-direct mode PMEM is exposed to the users over a block device symbol, it is required to have either conventional storage stack or PMEM-optimized system software. Recently, many studies reveal that the conventional storage stack is heavy and degrades the system performance due to frequent software interventions. Therefore, recent studies insist on using *direct access* (DAX), which removes most software interventions at the storage stack side. We observe that DAX outperforms the conventional storage stack with EXT4 by 3.03× and 3.27× for read and write, respectively. However, since DAX also omits persistence management, in-memory graph systems need to "manually" control the system's data persistence.

III. Challenge: Raw graph data != Runtime Data

As the memory capacity becomes more than hundreds of GB in modern servers, many studies argue that it is affordable to accommodate most graph data in memory [2]. However, we observe that the runtime memory requirement is much higher than what the raw graph demands. Figure 2 normalizes the graph-related memory requirement to the size of the raw graph in order to show how much the runtime data is amplified for the in-memory graph analysis. The final graph data after graph construction is 3.93× greater than the raw graph (i.e., memGraph), and the memory requirement in the graph construction is 6.71× as high as the size of the raw data (i.e., memPeak). To be precise, we perform a time series analysis for the memory footprint during the graph construction, and the results are shown in Figure 3. First, right after the graph framework reads the raw graph, the memory requirement sharply increases, which is 3.85× higher than the size of the raw graph. This is because the framework needs to convert the data chunks to meaningful information such as vertex/edge numbers (i.e., long integer) (Ⓐ). The graph framework then allocates a heap to generate additional data structures to accelerate graph processing. This process introduces redundant data copies and



(a) Random read. (b) Sequential read.

Fig. 4: Latency analysis.

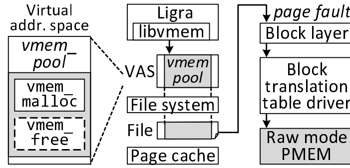
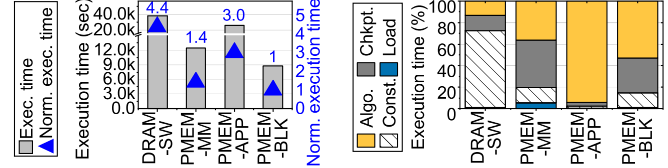


Fig. 5: Modified software stack of Ligra.



(a) Execution time. (b) Execution time breakdown.

Fig. 6: Execution time analysis (8.6B edges).

memory operations, which make the target system need 6.71× greater memory spaces than what the original graph requires (Ⓑ). Lastly, it releases the temporal data structures from the heap, which address the memory consumption by 41% (Ⓒ).

Since the actual memory requirement is unknown till the target framework tries to construct its in-memory structures, the system can fail to process data at runtime owing to out-of-memory. To address this runtime error, one may extend the memory by combining it with the underlying storage as a memory expansion. However, based on our evaluation, the execution time becomes unacceptable after the extended virtual memory begins to perform demand paging as the large amount of data is issued to the swap partition.

IV. In-Memory Graph Analysis with PMEM

To process the large graph in a reasonable time without causing the out-of-memory, we can use PMEM, which is larger than DRAM, and faster than conventional storage devices. In this section, we would like to figure out what would be the best PMEM configuration for the large-scale graph processing.

Enabling raw PMEM with slow storage stack. The best configuration should be able to maximize the performance of in-memory graph systems while minimizing the overhead imposed by persistence control. Instead of using either the DAX-enabled app-direct mode or memory mode, we here introduce raw PMEM (e.g., block storage) to in-memory graph analysis and turn back to employing a slow storage stack on the raw PMEM. Even though the storage stack is yet slower than DAX-enabled, app-direct mode PMEM (cf. Section II), it can buffer and/or cache data in VFS's page cache that naturally reaps the benefits of DRAM caching. Note that, once the page cache warms up, many graph processing operations can be served by DRAM-like performance, as shown in Figure 4. Compared to DAX-enabled PMEM, the storage stack enabled PMEM exhibits 5.36× and 1.62× longer read latency with random and sequential patterns, respectively, during the initial phases. However, after the initial phase, the storage stack's latency gets close to the latency of DRAM caching, while DAX exhibits 2.83× and 3.17× worse performance than the storage stack enabled system. In addition, the legacy file system guarantees atomicity, consistency, isolation, and durability in a natural way. Thus, it can minimize the overhead brought by the hand-worked persistence control as well.

Details of modification. As shown in Figure 5, we mount PMEM as block storage and replace the heap's DRAM space with the raw PMEM that we configured with a legacy file system (EXT4). We also modify Ligra to generate an invisible file to the mounted raw PMEM through `vmem_create()`, a POSIX-like API provided by `libvmem` library. It then generates a virtual memory pool (`vmem_pool`) by mapping the invisible file to Ligra's virtual address space using `mmap()`. Ligra accesses PMEM's `vmem_pool` as its heap memory by allocating memory through `libvmem`'s `vmem_malloc()`. When our modified Ligra accesses an allocated heap memory at the beginning, it raises a cold miss on the page cache. The system's page fault handler then goes through the conventional storage stack to handle the page fault.

V. Evaluation and Conclusion

We configure four in-memory graph systems: 1) *D-SW* extends the memory by enabling swap partition on SSDs. 2) *P-MM* and 3) *P-APP* configure PMEM in the memory mode and app-direct mode, respectively. 4) *P-BLK* uses the modified Ligra with raw PMEM and storage stack. To analyze the performance, we execute seven graph algorithms with Ligra back to back: `BC`, `BFS`, `CC`, `MIS`, `Radii`, `PR`, and `Tri`. The node that we used for the evaluations contains four 3.9GHz CPU sockets, each employing 1.5 TB 3D-Xpoint and 192GB local-DRAM. The node also utilizes four 1.8 TB NVMe SSDs for memory expansion and external storage. Metadata for checkpoint-restarts are located in the slow NVMe SSDs for D-SW and P-MM, whereas they are served from 3D-Xpoint for P-APP and P-BLK, which provide persistence.

Figures 6a and 6b show the execution time and corresponding latency decomposition, respectively. Since P-BLK takes both large storage capacity and DRAM caching, it exhibits 77% and 67% shorter execution time compared to D-SW and P-APP, respectively. Furthermore, P-BLK also outperforms P-MM, which takes the full advantage of large memory capacity and DRAM caching. This is because P-BLK can accelerate the checkpointing process using its persistent 3D-Xpoint space, while P-MM needs to access slow SSDs.

In this paper, we explore the challenges of in-memory graph processing. We also conduct comprehensive evaluations with real PMEM devices to understand behaviors of different operation modes and system software frameworks to enable PMEM. We finally modify Ligra to satisfy the requirement of graph processing performance and data persistence in memory.

VI. Original Publication

H. Bae, M. Kwon, D. Gouk, S. Han, S. Koh, C. Lee, D. Park and M. Jung. 2021. Empirical Guide to Use of Persistent Memory for Large-Scale In-Memory Graph Analysis. IEEE ICCD. *https://bit.ly/3IANLnS*

VII. Acknowledgement

This research is mainly supported by NRF 2021R1AC4001773, IITP 2021-0-00524, and IITP 2022-0-00117. The work is also supported in part by S3RC Hynix Center, SK-Hynix (G01200477), ETRI (21ZS1300), and KAIST start-up package (G01190015). D. Park is funded by NRF 2020R1F1A1048485. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies. Myoungsoo Jung is the corresponding author.

References

[1] E. Abdelhamid *et al.*, "Scalemine: Scalable parallel frequent subgraph mining in a single large graph," in *IEEE SC*, 2016.

[2] J. Chhugani *et al.*, "Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency," in *IEEE IPDPS*, 2012.

[3] J. Shun *et al.*, "Ligra: a lightweight graph processing framework for shared memory," in *ACM SIGPLAN PPoPP*, 2013.

[4] S. Hougardy, "The floyd–warshall algorithm on graphs with negative cycles," *Information Processing Letters*, 2010.

[5] J. Xue *et al.*, "Seraph: an efficient, low-cost system for concurrent graph processing," in *ACM HPDC*, 2014.