XPGraph: XPline-Friendly Persistent Memory Graph Stores for Large-Scale Evolving Graphs

Rui Wang
Zhejiang University
Hangzhou, China
rwang21@zju.edu.cn

Shuibing He*
Zhejiang University
Hangzhou, China
heshuibing@zju.edu.cn

Weixu Zong
Zhejiang University
Hangzhou, China
zorax@zju.edu.cn

Yongkun Li
University of Science and Technology of China
Hefei, China
ykli@ustc.edu.cn

Yinlong Xu
University of Science and Technology of China
Hefei, China
ylxu@ustc.edu.cn

2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00091

***Abstract*—Traditional in-memory graph storage systems have limited scalability due to the limited capacity and volatility of DRAM. Emerging persistent memory (PMEM), with large capacity and non-volatility, provides us an opportunity to realize the scalable and high-performance graph stores. However, directly moving existing DRAM-based graph storage systems to PMEM would cause serious PMEM access inefficiency issues, including high read and write amplification in PMEM and costly remote PMEM accesses across NUMA nodes, thus leading to the performance bottleneck. In this paper, we propose XPGraph, a PMEM-based graph storage system for managing large-scale evolving graphs, by developing an *XPLine-friendly graph access model* with vertex-centric graph buffering, hierarchical vertex buffer managing, and NUMA-friendly graph accessing. Experimental results show that XPGraph achieves 3.01× to 3.95× higher update performance and up to 4.46× higher query performance, compared with the state-of-the-art in-memory graph storage system implemented on a PMEM-based system.**

***Keywords*-graph processing; persistent/non-volatile memory; storage systems;**

I. Introduction

In recent years, graph storage systems have attracted wide attention [14], [32], [48], [49], [62], [63], as a well-designed graph store is essential to support high-performance graph update and graph analysis. The most commonly used in-memory storage formats for evolving graphs include: (1) edge list [60], [61], which stores each edge as a record and can support efficient data ingestion for evolving graph situations, but suffers low query performance, and (2) adjacency list [4], [22], [55], [84], which puts all neighbors of a vertex in together, thus supporting efficient graph queries, but not benefiting fast graph updates. The most recent work GraphOne [36] combines the above two complementary graph storage formats to a hybrid store, thus supporting efficient graph updates and queries.

*Shuibing He is the corresponding author.

However, as graphs get larger and larger, these in-memory graph systems face scalability issues due to the limited DRAM capacity. For example, GraphOne fails to run on the medium-sized graph Yahoo Web [80] with 6.6 billion edges by using a server with 128 GB DRAM, for the out-of-memory error. Therefore, many distributed graph systems [7], [8], [25], [42], [46], [83] and disk-based graph systems [23], [33], [38], [40], [57], [71], [74], [85] are designed to handle large graphs by placing them on clusters of machines or on disk, respectively. But these systems also face performance issues of high communication cost between cluster machines or high I/O cost of internal and external memory interaction. Furthermore, to keep crash consistency, graph updates need to be persisted to the durable disk, incurring extra I/O persistence and recovery cost.

Recently, the research for persistent memory (PMEM) has gained a great breakthrough with the release of Intel Optane Persistent Memory [50], to make the scalable PMEM with larger storage capacity finally commercially available. It provides us another way to solve the scalability problem of in-memory graph stores, and is expected to achieve high-performance graph stores for large-scale evolving graphs, and meanwhile realizing the data persistence.

However, directly moving existing DRAM-based graph systems to PMEM would suffer from a severe performance drop, for the completely different performance characteristics between DRAM and PMEM. For instance, when we use GraphOne to import the Friendster graph [24], it costs 6.37× time on PMEM than that on DRAM. We also find that, besides the hardware performance gap between PMEM and DRAM, software designs are the main causes of the performance degradation. First, DRAM-based graph systems often produce a lot of intensive small random writes, e.g., 4-byte vertex IDs, to different adjacency lists of different vertices. These random writes do not largely impact the performance of DRAM-based systems for DRAM's high random write

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00091

performance, but each 4-byte random write to PMEM may bring a 256-byte XPLine read-modify-write operation, thus causing the serious read and write amplification problem and becoming the performance bottleneck. Second, non-uniform memory access (NUMA) architecture is a necessity for providing massive bandwidth and capacity of PMEM, for the limited DIMM slots and cores in a single CPU [73]. But the latency of cross-NUMA access of PMEM is much higher than that of DRAM, especially in the case of multi-threaded accessing [81], which further exacerbates the PMEM access efficiency problem of current in-memory graph systems.

Various design efforts have been made recently to carefully manage the data stored in PMEM and improve the PMEM access efficiency, which include PMEM allocators [3], [5], [18], [43], [52], [59], PMEM indexes [9], [11], [31], [41], PMEM based key-value stores [12], [30], [77], and PMEM based file systems [10], [13], [20], [78]. However, as graph systems have different access patterns compared to the above systems, i.e., highly random accesses and poor data locality, it is inefficient to directly apply the above efforts to graph stores. Therefore, it is necessary to design a dedicated PMEM-based graph storage system for efficiently handling the large-scale evolving graphs.

An intuitive solution to the read-write amplification problem is to use DRAM as buffers to cache and merge data updates to PMEM, but its application to dynamic graphs is non-trivial for following challenges. First, edge updates for evolving graphs have poor data locality, which makes it difficult to design an efficient buffering strategy. And the data stored in DRAM buffers would be lost after a power outage, which may lead to data inconsistency issues. Second, the large performance benefits obtained by buffering strategies are often accompanied by high DRAM space requirements, which may limit system scalability. Besides, a simple buffering strategy can not reduce the high cost of remote PMEM accesses across NUMA nodes under multi-threading.

To address the challenges above and improve the graph data access efficiency, we develop XPGraph, an efficient PMEM-based graph storage system for larger-scale evolving graphs, by adopting an *XPLine-friendly graph access model*. XPGraph mainly focuses on reducing the high PMEM access costs during the graph ingestion process, which includes relieving the heavy PMEM read and write amplification, and avoiding the remote PMEM accesses across NUMA nodes. In summary, our main contributions are as follows.

- We propose a *vertex-centric graph buffering strategy*, that allocates a DRAM vertex buffer to temporarily cache some edge updates for each vertex, and flushes the entire buffer to PMEM when it is full, thus merging multiple XPLine accesses to PMEM into single XPLine access. We also introduce a *periodical flushing strategy* for fine-grained edge-level consistency guarantees.
- We design a *hierarchical vertex buffer managing scheme*, which dynamically adjusts the size of each vertex buffer according to the changes of the vertex degree, thus reducing the DRAM space overhead, while maintaining the performance benefits. We also adopt a *buddy-liked memory pool managing strategy* to reduce the memory allocation/free cost of these vertex buffers.
- We introduce a *NUMA-friendly graph accessing method*, which separately places the different graph data in different NUMA nodes and binds the processing threads to cores of the corresponding node, thus completely avoiding the remote PMEM accesses across NUMA.
- We implement the prototype XPGraph and conduct extensive experiments to demonstrate its efficiency. Results show that XPGraph achieves 3.01× to 3.95× higher ingestion rate as well as up to 4.46× higher query rate, compared with the state-of-the-art in-memory graph storage system implemented on a PMEM-based system.

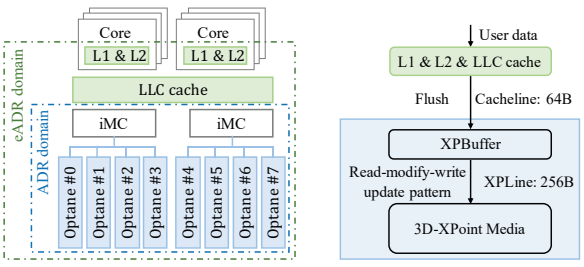
The rest of this paper is organized as follows. In §II, we first introduce characteristics of PMEM, the commonly used graph storage formats in DRAM, and analyze the limitations of directly moving DRAM-based graph stores to real PMEM. In §III and §IV, we present the design and implementation details of XPGraph, respectively. In §V, we show the evaluation results. Finally, §VI reviews related work and §VII concludes.

II. Background and Motivation

We first introduce the characteristics of the emerging persistent memory (PMEM) and how it differs from the traditional DRAM. Then we illustrate the storage format and access process of the DRAM-based graph storage strategy, by taking the state-of-the-art dynamic graph storage system GraphOne [36] as the study case. Finally, we analyze the limitations of DRAM-based graph storage strategy in supporting PMEM resident evolving graphs.

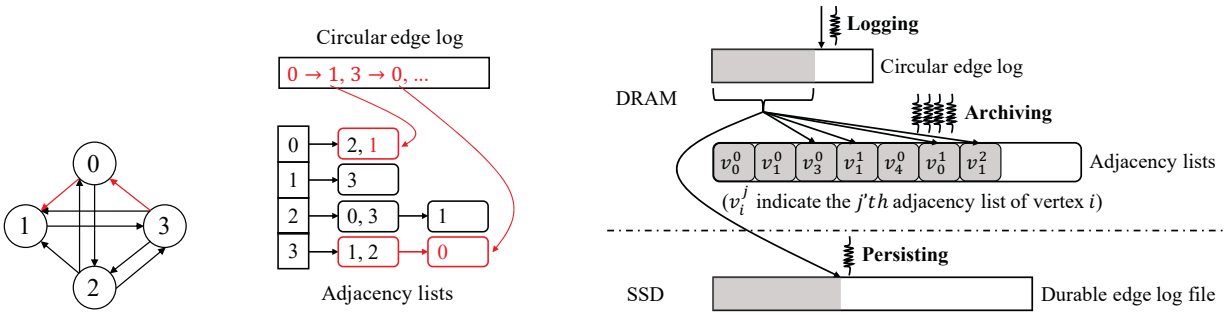
A. Persistent Memory

We take the latest PMEM product, i.e., Intel Optane Persistent Memory 200 Series (abbreviated as *Optane* in the rest of the paper), as the study case, to introduce the access process and the performance characteristics of PMEM-based systems. As shown in Fig.1(a), Optane sits on the memory bus



(a) Optane DIMM (b) Write data process

Figure 1. Intel Optane Persistent Memory 200 Series.



(a) Example graph (b) Hybrid graph storage format (c) Graph ingestion process

Figure 2. Storage and access process in GraphOne

like normal DRAM, and communicates with the processor's iMC (integrated memory controller). Data updates would first be written to the CPU caches, and then flushed to PMEM in cache line granularity. PMEM systems support one of two persistent modes: (1) Asynchronous DRAM Refresh (ADR) mode, in which the programmers must explicitly flush cache lines to ensure crash consistency. (2) Extended ADR (eADR) mode, in which the CPU caches are also included in the *power fail protected domain*, so no flushing is necessary. We focus on eADR mode since it is already supported in the 3^{rd} generation of Intel Xeon Scalable Processors [21].

However, Optane is very different from DRAM in its internal access process, as shown in Fig.1(b). (1) Optane uses the 3D-Xpoint media to store data internally, whose actual physical access granularity is 256-byte XPLine, which is larger than the 64-byte cache line size. (2) Before accessing the 3D-XPoint media, Optane manages a small XPBuffer internally to cache and merge some data updates, so a data write to the 3D-Xpoint media is actually a read-modify-write with XPLine granularity. (3) Optane has limited ability in handling accesses from multiple threads simultaneously for its limited store performance. (4) NUMA (non-uniform memory access) effects for Optane are much larger than they are for DRAM, especially for the multi-threaded cross-NUMA accesses [81]. Therefore, Optane's performance characterization is much more complicated and can be fluctuated by many factors, including access type (read vs. write), access pattern (sequential vs. random), access size and so on [81]. We need to carefully manage the data in Optane to realize a good performance.

B. DRAM-based Graph Stores

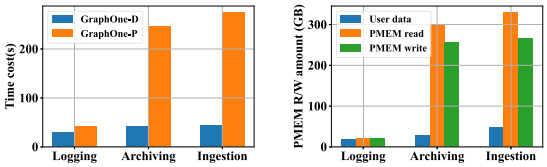
For simplicity, we take GraphOne [36], the state-of-the-art in-memory evolving graph storage system, as an example to illustrate the storage format and access process of the DRAM-based graph stores. Note that, the storage formats (edge list and adjacency list) and the vertex-centric random access pattern used in GraphOne, are all widely used in many other DRAM-based in-memory graph storage systems like [14], [22], [32], [47], [49], [62], [63].

GraphOne uses a hybrid storage format by combining two most commonly used in-memory graph storage formats, i.e., edge list and adjacency list. Specifically, GraphOne first uses a circular edge log in DRAM to store the latest graph updates in the edge list format, thus supporting efficient data ingestion for evolving graphs. Besides, GraphOne also uses many adjacency lists to store the older data, i.e., edges that are archived periodically from the edge log, thus supporting efficient graph queries. Fig.2(b) shows the hybrid graph storage of the sample graph in Fig.2(a), where the black edges are the older edges stored in adjacency lists, and the red edges are the newly updated edges stored in a circular edge log, and they would be moved to adjacency lists after an archiving phase.

Fig.2(c) further shows the graph ingestion process in GraphOne, which is managed in several phases. During the *logging phase*, the incoming updates are expressed in edge list format and appended to the tail of the circular edge log in memory by a dedicated logging thread, thus contributing to a high ingestion rate. When the number of edges reaches the predefined threshold, GraphOne comes to a parallel *archiving phase*, which converts the older edges of edge list format to adjacency list format, by putting all incremental neighbors of a vertex in together as a single adjacency list, thus supporting efficient graph queries. Specifically, GraphOne adopts a *global batched edge-centric archiving strategy*, which first counts the degree increment of each vertex before each archiving, and allocates adjacency lists for vertices according to their incremental degrees, then it traverses each edge and appends the neighbor to the corresponding adjacency list by multiple archiving threads. Besides, GraphOne also manages a *persisting phase*, which uses a separate persisting thread to periodically write the edge log data to a durable edge log file in disk to guarantee the durability to some extent.

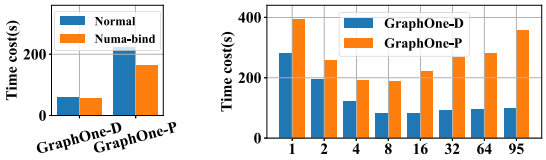
C. Limitations for PMEM Graph Stores

DRAM-based in-memory graph storage systems perform well for DRAM-resident graphs, but the graph scale it can support is limited by the DRAM capacity. For example,



(a) Time cost (b) PMEM read and write

Figure 3. Compare GraphOne-D with GraphOne-P.



(a) NUMA (b) Number of threads

Figure 4. Impact of NUMA and number of threads.

GraphOne needs at least 63.89 GB memory space to run on the small-sized graph Friendster [24] with 2.6 billion edges, and even fails to run on the medium-sized graph Yahoo Web [80] with 6.6 billion edges by using a server with 128 GB DRAM, for out-of-memory error. Please refer to §V-A for more dataset information and experiment settings. Besides, DRAM-based in-memory graph storage systems also need an extra disk to realize the data persistence, for the volatility of DRAM, which also brings extra durable cost or/and high recover cost. For example, GraphOne's periodical persistence to disk can only achieve coarse-grained consistency guarantees, and it is also time-consuming for recovery after power failure, for all archiving processes need to be redone.

To realize large-scale graph processing in memory and meanwhile achieving fine-grained consistency guarantee and fast recovery, PMEM with large capacity and non-volatility provides us a good opportunity. However, when directly moving the storage formats that are designed for DRAM to real PMEM, we may suffer from severe performance drop for the different characteristics between DRAM and PMEM. We implement a PMEM-based GraphOne (abbreviated as GraphOne-P) by migrating the circular edge log and all adjacency lists from DRAM to PMEM (using *pmem_map_file()* of PMDK [54] for allocating PMEM space), and keeping the metadata (like vertex indexes and snapshots information) and the intermediate data (like temporary edge lists) in DRAM, and then turning off additional persistence operations. We conduct experiments to compare the performance gap between GraphOne-P and the original DRAM-based GraphOne (abbreviated as GraphOne-D), in the aspect of graph updates ingestion, which consists of the parallel *logging* and *archiving* processes. We find that GraphOne-P costs near five minutes to ingest the small graph Friendster with 2.6 billion edges, which is 6.37× higher than the cost of GraphOne-D.

High read and write amplification in PMEM. We further separately test the logging and archiving performance of GraphOne-D and GraphOne-P, respectively, and show the results in Fig.3(a). It indicates that the performance of logging process with sequential writes in GraphOne-P drop a little compared with that in GraphOne-D, for the hardware bandwidth differences. The performance bottleneck of GraphOne-P mainly comes from the graph archiving process, in which a lot of small data updates (4-byte vertex ID of neighbor information) are written to the adjacency lists stored in PMEM, and each 4-byte random write may cause an XPLine (256-byte) read-modify-write to the 3D-XPoint media according to the Optane's access process, which finally causes the heavy *PMEM read-write amplification problem*. We also measure the data amount read from and written to the PMEM during the conduct of GraphOne-P, by using Intel PCM tool [53], and show the result in Fig.3(b). We can see that GraphOne-P brings 9.96× read amplification and 8.56× write amplification during the archiving process, which causes the high PMEM access costs.

Costly remote PMEM accesses across NUMA nodes. Besides, among these PMEM accesses, a large portion of them are cross-NUMA remote PMEM accesses, which further exacerbate the problem of high cost for graph data access. We also evaluate the NUMA impact of GraphOne-D and GraphOne-P by comparing the time cost of ingesting Friendster for normal execution and binding only one NUMA node, respectively, and show the results in Fig.4(a). We can see that NUMA effects are much larger for GraphOne-P than they are for GraphOne-D. In addition, we also test the impact of the number of archiving threads for GraphOne-D and GraphOne-P, and show the results in Fig.4(b). We find that GraphOne-P will suffer from the severe performance drops after setting the number of archiving threads as larger than eight, because of PMEM's limited performance in multi-threaded accesses, especially for cross-NUMA accesses.

In conclusion, directly moving existing DRAM-based in-memory graph storage systems to PMEM would suffer severe performance drop for high PMEM access costs, including high read and write amplification in PMEM and costly remote PMEM accesses across NUMA nodes. In the next section, we will present our designs to carefully solve the problems mentioned above.

III. Design of XPGraph

In this section, we first introduce the main idea and the overview of XPGraph, which is a PMEM-based graph storage system that supports large-scale evolving graphs. Then, we present the details of its key design techniques, including vertex-centric graph buffering, hierarchical vertex buffer managing, and NUMA-friendly graph accessing.

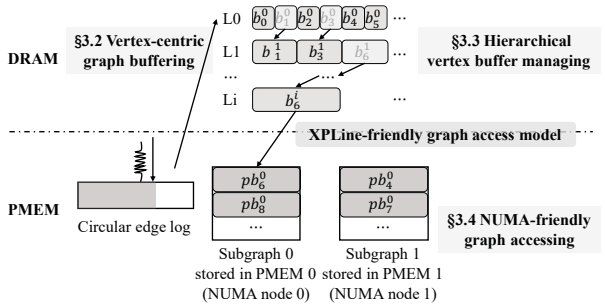


Figure 5. Overview of XPGraph.

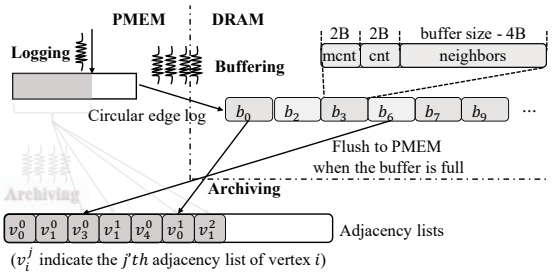


Figure 6. Vertex-centric graph buffering.

A. Overview

We target for improving the evolving graph storing performance by reducing the PMEM access cost, which includes PMEM read-write amplification and cross-NUMA remote PMEM accesses. A classic solution to address the problem is to deploy DRAM buffers to cache and coalesce some data updates to PMEM. However, its application to large-scale dynamic graph stores is non-trivial due to the following challenges.

- Edge updates in evolving graph applications are generally with poor data locality for the complex relationships between vertices, thus making it difficult to design an efficient buffering strategy for these updated graph data. Besides, since the graph data stored in DRAM buffers may be lost on power failure, it is necessary to guarantee the edge-level data consistency.
- In general, the impressive performance benefits obtained from the buffering strategies are often accompanied by high DRAM space requirements, and the poor locality of graph data further increases the DRAM demands for achieving good buffering performance, which may lead to the DRAM space pressure, limiting the system scalability. Besides, frequent memory allocation and freeing of vertex buffers may further increase the memory management costs.
- Additionally, the buffering strategy can not reduce the high cost of remote PMEM accesses across NUMA nodes, and how to reduce or avoid the cross-NUMA remote PMEM access to further improve the PMEM accessing performance?

In order to achieve our goal and handle the challenges mentioned above, we propose an *XPLine-friendly PMEM access model* with three techniques, according to the XPLine-centric access pattern of PMEM mentioned in §II-A. The designed system overview is shown in Fig.5. First, we propose a vertex-centric graph buffering strategy to amortize the PMEM access cost for each edge update, and also introduce a periodical flush technique for fine-grained edge-level consistency guarantee. Then, we design a hierarchical buffer managing scheme to limit the DRAM space demands, and also adopt a buddy-liked memory pool management to reduce the memory managing cost. Last, we introduce a NUMA-friendly graph accessing method to totally avoid the cross-NUMA PMEM access. In the following subsections, we present these three techniques in detail.

B. Vertex-Centric Graph Buffering

To address the severe read-write amplification problem brought by the edge-centric adjacency list writes in current DRAM-based graph storage systems, we introduce a vertex-centric graph buffering strategy, to cache some edge updates in DRAM, thereby reducing actual writes to PMEM. In this subsection, we first introduce the process of the vertex-centric buffering strategy, and then present the periodical flush strategy for realizing the fine-grained edge-level consistency guarantees. Finally, we analyze the DRAM space requirements of this vertex-centric buffering strategy.

Vertex-centric buffering strategy. As shown in Fig.6, we allocate a temporary adjacency list stored in DRAM (stated as vertex buffer in the rest of the paper) for each vertex with edge updates, to temporarily cache some edge updates of the same vertex together. Each vertex buffer has a 4-byte *header* to store the maximum count (*mcnt*) and current count (*cnt*) of neighbors stored in this vertex buffer, and the rest space is used to store the vertex ID of these neighbors. For example, when we set the vertex buffer size as 16 bytes, then the maximum count of neighbors this buffer can store equals $(16-4)/4=3$. When a DRAM vertex buffer is full, we flush all neighbors in this vertex buffer to the corresponding PMEM adjacency list by only one XPLine access, then clear this vertex buffer for caching subsequent edge updates. With this vertex-centric graph buffering strategy, we can merge multiple XPLine accesses to one XPLine access, and amortize the PMEM write cost for each edge update. Note that, the buffers can also serve as caches to reduce the reads from PMEM, improving graph query performance (§V-C).

Periodical flushing for consistency guarantees. Since data stored in DRAM vertex buffers may be lost on power failure, we also design a *periodical flushing strategy* by redesigning the circular edge log structure, which are stored in PMEM, for fine-grained edge-level consistency guarantees.

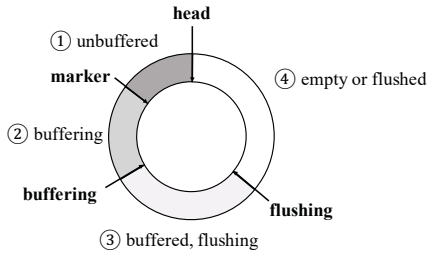


Figure 7. Consistency guaranteed circular edge log.

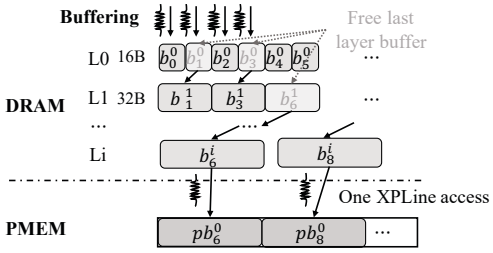


Figure 8. Adaptive hierarchical buffer size adjusting.

As shown in Fig.7, new edge updates are always appended to the *head* position of the circular edge log, sequentially in clockwise order. When the number of non-buffered edges in the edge log reaches a predefined threshold, we start a *buffering phase* to buffer these edges to their corresponding DRAM vertex buffers. During the buffering phase, when a vertex buffer is full, we flush all neighbors in this DRAM vertex buffer to PMEM. The edges between the *marker* position and the *buffering* position are currently being buffered, and the edges after the *buffering* position are already buffered. Considering the volatile of DRAM vertex buffers, we can not overwrite these buffered edges in the circular edge log. Hence, when the number of buffered edges in the edge log reaches a predefined threshold, we start a *flushing phase* to flush all DRAM vertex buffers, including out-neighbors and in-neighbors of all vertices, to PMEM for data persistence. We introduce a *flushing* pointer to indicate that the edges after the *flushing* position are already flushed to PMEM, so they can be overwritten by new coming edge updates.

Once system crashes, we can recover the lost DRAM vertex buffers by traversing the edges between the *marker* and *flushing* positions of this circular edge log. Note that, some of these edges have already been flushed to PMEM when their corresponding vertex buffers were full. To avoid data redundancy after recovery, before writing a neighbor to its DRAM vertex buffer, we need to check whether it is already stored in the corresponding PMEM adjacency list.

DRAM space requirements for vertex buffers. Note that, the performance benefit gained with this vertex-centric graph buffering strategy is traded off with the per-vertex buffer size setting, i.e., a larger per-vertex buffer size setting brings a better performance, but at the cost of more DRAM space requirements. We also conduct experiments to evaluate the impact of per-vertex buffer size setting for Yahoo Web [80] graph in §V-E, and we find that the larger the per-vertex buffer size is, the more neighbors can be cached in DRAM for each vertex, then generally brings the less time cost for completing the graph ingestion process. However, we need to pay for this increased benefit with a larger demand of DRAM space, which heavily limits the system's scalability. For example, we need over 50 GB DRAM for supporting this medium-sized graph Yahoo Web when we set the per-vertex buffer size as 256-bytes, and we even can not afford the larger graph Kron30 [27] under this setting, in our server with 128 GB DRAM. Therefore, in the next subsection, we will struggle to reduce the DRAM space requirements while maintaining the performance benefit.

C. Hierarchical Vertex Buffer Managing

In this subsection, we first introduce the power-law vertex degree distribution of real-world graphs, then propose an adaptively hierarchical buffer size adjusting scheme, according to this characterization of real-world graphs, to reduce the DRAM space requirements for storing these vertex buffers. Finally, we also adopt a buddy-liked memory pool management to reduce the time cost for managing the frequent allocating/freeing of these vertex buffers with different sizes under multi-threading.

Power-law degree distribution of real-world graphs. In real-world graphs, vertex degrees, i.e., numbers of neighbors, of different vertices may vary greatly, and they generally have a power-law distribution [8], [25]. For example, in our evaluated four real-world graphs (refer to §V-A), vertices with a degree of 1 or 2 can account for more than 40% and even a half for many other real-world graphs [82], and vertices with a degree ranging from 4 to 7 account for around another 20%, while only a small part of vertices have degrees larger than 64. Therefore, if we allocate large buffers for these low-degree vertices, it will cause a serious waste of DRAM space, and if we allocate small buffers for these high-degree vertices, it will seriously affect the benefits of the vertex-centric buffering. Namely, the fixed-size setting for all vertex buffers is hard to realize a balanced trade-off for all vertices. Based on this founding, we target to realize differentiated vertex buffer size settings for different vertices with different degrees. However, it is also challenging as we can not predict the final degree for each vertex in the evolving graph situations.

Adaptive hierarchical buffer size adjusting. To solve the above challenge, we propose an adaptively hierarchical vertex buffering scheme, which adaptively adjusts the vertex buffer sizes according to the changes of vertex degrees, as shown in Fig.8. Specifically, we create an initial buffer sized as 16 bytes for the vertex with edge updates, and state it as layer 0

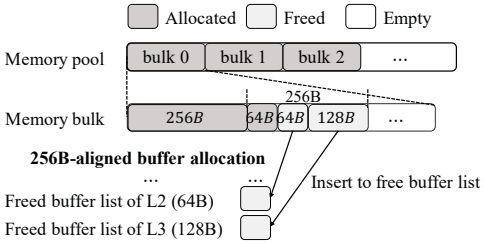


Figure 9. Memory pool based vertex buffer management.

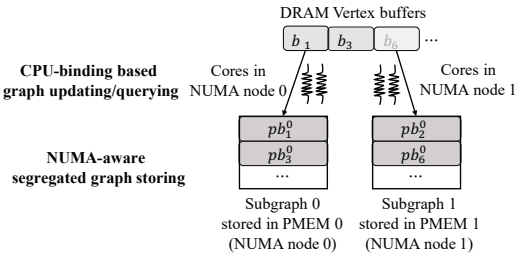


Figure 10. NUMA-friendly graph accessing.

(L0) buffer. The new coming edge update is first written to its corresponding L0 buffer, which can hold at most 3 neighbors. When a vertex's L0 buffer is full, we create a double-sized (32-byte) layer 1 (L1) buffer, and move all buffered neighbors to its L1 buffer, and then free the L0 buffer. After that, the new coming edge updates of this vertex would be directly written to its L1 buffer, which can hold at most 7 neighbors. When a L1 buffer is full, we create a double-sized (64-byte) layer 2 (L2) buffer, and repeat the process above, until the buffer size reaches the predefined maximum buffer size. For example, if we specify the maximum buffer size as 256 bytes, we will totally manage five DRAM layer-buffers at most. When a L4 buffer is full, we flush all neighbors in it to the corresponding persistent adjacency list stored in PMEM by only one XPLine access and then clear the L4 buffer. With this adaptive hierarchical buffer size adjusting strategy, we can reduce the DRAM space requirements with low-layer buffers for low-degree vertices, while maintaining the performance benefit gained from the high-layer buffers for the high-degree vertices.

Note that, this hierarchical buffer size adjusting strategy incurs extra data movement overhead across different layers, but this overhead is acceptable for two reasons. First, the data copy operation does not happen frequently on each vertex, because we buffer the edge updates in batches (see §IV-A). If a vertex's edge count reaches a higher level in current batch, we skip the allocation of lower layer buffers. Second, the performance bottleneck mainly lies in the PMEM accesses, thus the limited DRAM data movement cost can be hidden. We study the efficiency of this strategy in §V-E and the comparison of Fig.16(a) (fixed buffers) and 16(b) (hierarchical buffers) also validates that the data movement cost is acceptable.

Buddy-liked memory pool management. To achieve the adaptively hierarchical buffer managing, we need to frequently allocate and free plenty of small DRAM pieces of different sizes, e.g., range from 16-byte to 256-byte, which brings high memory managing cost, especially for the high concurrency multi-threads buffering scenarios, since it may bring frequent system user mode/kernel mode switch, lock contention, and memory fragmentation. To reduce the memory allocation/free cost of these vertex buffers, we adopt memory pool management, which directly allocates a large of memory space in advance to manage the buffer allocation/free by itself, as shown in Fig.9. First, to avoid the access conflict of multiple threads, we cut the memory pool into many smaller memory bulks, each with the size of 16 MB by default. Each thread acquires a separate memory bulk to satisfy the current thread's buffer allocation, and applies for a new memory bulk if it's runs out. For each thread with a contiguous memory bulk, we adopt a buddy-liked buffer allocation/free strategy, which considers the vertex buffers with the same size and adjacent memory space as *buddy buffers*, and manages a free buffer list to recycle the freed space for each buffer size. When a new buffer allocation comes, we first find useable memory space in the free buffer lists, and if not found, then we fetch from the memory bulk. We also use an aligned memory allocation strategy for efficient freed vertex buffer space recycling. To further reduce the DRAM space cost, we also recycle the freed buffer space for meeting the subsequent buffer allocations. Specifically, when a vertex buffer is freed, we add it into the corresponding free buffer list according to its size.

With this memory pool based DRAM space managing strategy, we can efficiently reduce the management cost of numerous small DRAM segments. Furthermore, considering scalability issues, we can also limit the DRAM usage of these vertex buffers by setting a threshold for the memory pool size. If the memory pool is going to be exhausted soon, we flush all vertex buffers to PMEM to recycle all memory pool space. We also evaluate the impact of the memory pool sizes in §V-F to show the performance sensitivity.

D. NUMA-Friendly Graph Accessing

As we discussed in §II-C, NUMA effects for PMEM are much larger than they are for DRAM [81], which largely degrade the data access performance of graph updates and queries. To completely avoid remote PMEM accesses across NUMA nodes, we propose a novel NUMA-friendly graph accessing strategy, as shown in Fig.10. It first stores different parts of the graph data in different NUMA nodes, then binds the graph updating/querying threads to the CPU cores of the corresponding NUMA nodes.

NUMA-aware segregated graph storing. We partition the graph data into P parts for a P-socket system, and store

each part of the graph data in the PMEM(s) of a separate NUMA node. For example, a simple implementation based on a two-socket system is to store the out-graph data in PMEM 0 of NUMA node 0, and store the in-graph data in PMEM 1 of NUMA node 1. We can also easily extend this technique with sub-graph based implementations, to make it suitable for more general situations, e.g., undirected graph formats or systems equipped with more NUMA nodes. That is, we can divide the whole graph into P sub-graphs, by existing graph partition strategies [2], [15], [58], [64], [67], and each sub-graph contains the neighbor information for a subset of all vertices. For example, we default to use the commonly-used hash-based graph partitioning strategy [16], [35], [37], which simply distributes vertex v to sub-graph $v\%P$, to make the numbers of vertices and edges be balanced in each sub-graph. Then, we place each sub-graph in the PMEM(s) of a separate NUMA node. We default to use the sub-graph based implementations if not specified, and we also compare the two kinds of implementations in §V-E.

CPU-binding based graph updating. During the graph buffering and flushing phases, when we update the graph data stored in PMEM(s) of NUMA node p, we can bind the buffering threads to CPU cores of NUMA node p, by a thread-class CPU binding function *pthread_setaffinity_np()* of Linux pthread [56]. For example, based on the out/in-graph segregated storing, when we flush the adjacency lists of vertices' out-neighbors, we bind the threads to cores of NUMA node 0 and when we flush the adjacency lists of vertices' in-neighbors, we bind the threads to cores of NUMA node 1. On the basis of sub-graph segregated storing, when we flush the adjacency lists of sub-graph p, we will bind the threads to cores of NUMA node p. Thus, we can completely avoid remote PMEM writes across NUMA nodes to all PMEM resident adjacency lists, since all corresponding flushing threads are bound to the local cores.

CPU-binding based graph querying. To further avoid remote PMEM reads across NUMA during graph queries, we can also bind the querying thread to the CPU cores of the corresponding NUMA node, when querying the corresponding adjacency lists of a vertex. However, the per-vertex CPU-binding strategy incurs a high cost of frequent thread migrations, which can be more than ten times the cost of remote PMEM access, in our experiments. Therefore, to avoid the high cost of frequent thread migrations, as well as the remote PMEM reads across NUMA during graph queries, we classify the vertex queries according to their belonged NUMA parts at the beginning of each computing iteration, and bind the querying threads to the corresponding CPU cores, respectively, before computing.

IV. Implementation

In this section, we first introduce three key data managing phases of XPGraph, then integrate some graph view interfaces according to these data managing phases. Finally, we implement three prototype systems to accommodate different system settings based on the above graph view interfaces.

A. Data Management Phases

For a better understanding, we describe how graph data is managed in DRAM and PMEM, which may go through three phases in XPGraph: logging, buffering and flushing.

Logging phase. A logging thread is called when edge updates are received from the client. The logging process sequentially writes the edge updates to the PMEM resident circular edge log (see §III-B), and will sleep waiting for available space in the circular edge log to avoid overwriting the non-flushed edge updates for data persistence. If the number of non-buffered edge updates in the circular edge log reaches a threshold, it will inform the system to start a buffering phase.

Buffering phase. A buffering phase is executed by multiple buffering threads to move a batch of non-buffered edges from the edge log to the DRAM vertex buffers. We adopt the *edge sharding based approach* used in GraphOne [36] to achieve load-balanced multi-threaded buffering, while maintaining the integrity of data ordering. Specifically, it shards the batched edges to multiple temporary ranged edge lists based on the ranges of their source vertex IDs, each of which can then be buffered in parallel without any atomic instructions. To handle the load imbalance issues, it creates a larger number of ranged edge lists than the available threads, and assigns different numbers of ranged edge lists to each thread, so that each buffering thread gets an approximately equal number of edges. For better performance, we store the temporary ranged edge lists in DRAM.

Flushing phase. During the buffering phase, when a vertex's highest layer buffer is full, the current buffering thread switches to a brief flushing phase for that vertex, moving its neighbors stored in DRAM vertex buffer to PMEM adjacency list. At the end of a buffering phase, if the number of non-flushed edges in the edge log reaches a threshold, which means we may overwrite the non-flushed edges soon, so it will notify the system to start a flushing phase for all vertices, i.e., all DRAM vertex buffers would be moved to PMEM adjacency list to ensure the data consistency. Additionally, if the vertex buffer pool's usage reaches a threshold, i.e., the pool is about to fill up, we also inform the system to start a flushing phase for all vertices, thereby emptying the vertex buffer pool for the next buffering phases.

In the eADR platform, no explicit cache line flush instruction is necessary for crash consistency (see §II-A), and data cached in CPU caches would be flushed to PMEM in an unexpected order of cache line granularity by system's cache replacement strategy, which may affect the PMEM access performance [9]. Hence, we also implement a *proactively flush strategy* to flush the adjacency lists with sizes equal or larger than XPLine (256 bytes) to PMEM by *clwb* instruction.

Table I
GRAPH VIEW APIS.

Graph Updating Interfaces	
status	add_edge(src, dst)
count	add_edges(buf, size)
count	buffer_edges(buf ,size)
status	del_edge(src, dst)
Graph Querying Interfaces	
count	get_nebrs_{out/in}(vid)
count	get_nebrs_log_{out/in}(vid)
count	get_nebrs_buf_{out/in}(vid)
count	get_nebrs_flush_{out/in}(vid)
count	get_logged_edges()
Graph Arranging Interfaces	
status	buffer_all_edges()
status	flush_all_vbufs()
status	compact_adjs(vid)
status	compact_all_adjs(vid)

It ensures that the graph data within the same XPLine would be flushed to PMEM in only one XPLine access.

B. Graph View Interfaces

We also provide simple data access APIs for user applications, by hiding the complex data management mentioned above. Table I shows the common-used graph view interfaces. The most critical interface for graph updating is the basic *add_edge(src, dst)* to receive a new edge, and we also implement several variants for receiving a batch of edges. We also realize the *del_edge(src, dst)* to delete an existing edge. As for graph querying interfaces, we provide the basic *get_nebrs_{out/in}(vid)* for querying all out/in neighbors of a vertex, and we also implement several variants to querying out/in neighbors in each data structure, respectively. We also provide *get_logged_edges()* for querying all non-buffered edges stored in the circular edge log. Besides, we provide graph arranging interfaces for transforming the graph data status between the three data management phases described above, which includes buffering all non-buffered edges to DRAM vertex buffers, and flushing all vertex buffers to PMEM's adjacency lists. We also implement the *compact_adjs(vid)* to merge a vertex's all adjacency lists stored in DRAM and PMEM to one large adjacency list, which is then stored in PMEM, thus supporting more efficient graph queries.

C. Prototype System

We implement all designs and optimizations mentioned above in around 8300 lines of C++ code, and encapsulate all the above interfaces into a library called *libxpgraph*. We then implement a prototype system XPGraph by calling these encapsulated interfaces. XPGraph is a graph storage framework running on a DRAM-PMEM hybrid memory system, that supports efficient large-scale evolving graph processing, while maintaining fine-grained data consistency for each edge update.

Besides, considering the fact that battery-backed DRAM is fairly cheap and is being mass-produced recently [19], [34], we also implement a variant system XPGraph-B to accommodate these battery-backed systems. Specifically, we modify the design of circular edge log to allow the logging process to overwrite the buffered edges, since DRAM vertex buffers are also included in the power-failure protection domain in battery-backed systems.

In addition, considering the situations that the system DRAM capacity is large enough and there is no crash consistent requirement, we also implement another variant system XPGraph-D to accommodate DRAM-only systems, which stores all data structures in DRAM, and sets the per-vertex buffer size as fixed 64 bytes to avoid frequent data movement. Note that we can simply switch between these three variant systems of XPGraph by just setting different parameters for different system configurations.

V. EVALUATION

A. Experiment Settings

Test bed. All experiments are performed on a server with two 2.10GHz Intel(R) Xeon(R) Gold 5318Y processors, each with 24 physical cores with hyper-threading enabled (48 logical cores). Each physical core has a private 32 KB L1 and 1 MB L2 caches, while all cores within a socket share a 36 MB L3 cache (LLC). For memory, it equips with 8 × 16 GB (128 GB) DRAM and 8 × 128 GB (1 TB) Intel Optane Persistent Memory 200 Series, which are interleaved inserted to the memory slot. It also equips with 3.84 TB Intel NVMe SSD for storage. The server runs Ubuntu 18.04.6 LTS with Linux kernel of 5.10.0.

Comparison systems. We take GraphOne, the state-of-the-art single-machine in-memory graph storage system, as a baseline for overall performance comparison. We compared three versions of GraphOne, i.e., (1) the original GraphOne on DRAM (**GraphOne-D**) that stores all data on DRAM. (2) GraphOne on PMEM (**GraphOne-P**), which stores the edge log and adjacency lists on PMEM by the *mmap*-based implementation, on the default Ext4-DAX file system, and keeps the meta-data (like vertex indexes and snapshots information) and the intermediate data (like temporary edge lists) on DRAM. (3) GraphOne on PMEM using the state-of-the-art PMEM file system NOVA [78] (**GraphOne-N**), which only stores the adjacency lists on PMEM and keeps other data on DRAM. To utilize the NOVA's optimizations, we implement a *file-I/O based GraphOne*, that only changes the adjacency list related memory interfaces based operations to file-I/O based operations. For a fair comparison, we mount NOVA in the *relaxed mode* that relaxes atomicity constraints on file data and metadata to achieve the best performance of NOVA [79]. We also performed the *file-I/O*-based GraphOne on the default Ext4-DAX file system, and observed about 4× the time cost on NOVA, which roughly matches the results

Table II
STATISTICS OF DATASETS.

Dataset	$\|V\|$	$\|E\|$	Bin Size	CSR Size
Twitter (TT)	61.6M	1.5B	12GB	12.4GB
Friendster (FS)	68.3M	2.6B	20.8GB	21.4GB
UKdomain (UK)	101.7M	3.1B	24.8GB	26.4GB
YahooWeb (YW)	1.4B	6.6B	52.8GB	75.2GB
Kron28 (K28)	256M	4B	32GB	36GB
Kron29 (K29)	512M	8B	64GB	72GB
Kron30 (K30)	1B	16B	128GB	144GB

reported in [79]. We provide implementations and detailed configurations of these three comparison systems, in our artifact (refer to §A). Note that, we exclude GraphOne on PMEM using NOVA by the *mmap*-based implementation to remove redundant comparison, as this system has almost the same performance as GraphOne-P, since NOVA does not optimize the *mmap* process in its latest code version [39].

Graph datasets. We use four real-world graphs Twitter [68], Friendster [24], UKdomain [69] and Yahoo Web [80], as well as three synthetic Kronecker graphs, i.e., Kron-28, Kron-29 and Kron-30, which are generated by graph500 generator [27], for our evaluation. These graphs are all widely used in graph system evaluations. Table II lists the graph information. They are all direct graphs, and Bin Size indicates the size of the dataset stored in binary edge list format, and CSR Size indicates the size of storing graphs in CSR format for both in-graphs and out-graphs.

Note that, CSR is the most compact storage format for static graphs. In the case of dynamic graphs, more memory space is usually required to store the evolving adjacency lists for all vertices. For example, GraphOne costs more than 40 GB for just storing the adjacency lists after ingesting the Friendster dataset (only 21.4GB in CSR format), and it also costs the other 23 GB memory space to store the metadata and intermediate data. Hence, for the larger graphs like Yahoo Web, Kron29 and Kron30, DRAM-based systems (GraphOne-D and XPGraph-D) fails to complete the ingestion process, as they require more memory space than the 128 GB DRAM capacity of our test bed.

Evaluation metrics. We evaluate XPGraph and its comparison systems in the following three aspects: (1) Graph ingesting performance, which includes the time cost and the PMEM read/write amount for ingesting above seven graphs. (2) Graph query performance, which includes the simple one-hop neighbor query, i.e., accesses the neighbors of random 2^{24} non-zero degree vertices, as well as three common graph computing algorithms BFS (traverses the connected sub-graphs of random three roots), PageRank (runs for ten iterations), and CC (finds the connected components in the graphs). (3) Graph recovery performance. Each experiment was performed ten times and the average completion time was calculated for a more accurate display of time cost.

B. Graph Ingestion Performance

Graph ingestion process imports the graph data by batching edges one by one from an edge buffer stored in the binary edge list format, which contains parallel single-thread logging process and multi-thread archiving process. For convenience, we also state XPGraph's buffering process and the flushing process as the archiving process in the rest of the paper. For a fair comparison, we use a unified 16 archiving threads for all tested systems to test their overall ingestion performance, as 16-thread setting shows relatively good performance for all of them. Note that, more archiving threads settings may bring performance improvements for XPGraph, which cause performance drop for GraphOne-P, and we also study the impact of the number of archiving threads in §V-F. We set the archiving threshold, i.e., the number of non-flushed edges stored in the circular edge log to trigger an archive process, as 2^{16} by default as well as GraphOne does.

Ingestion time cost for non-volatile systems. We first compare the time cost for ingesting the above seven graphs, by GraphOne-P, GraphOne-N and our XPGraph, XPGraph-B, respectively, as all of them can achieve the fine-grained edge-level consistency guarantee (XPGraph-B guarantees the consistency with a system-level battery supported). As Fig.11 shows, GraphOne-N is always an order of magnitude slower than other three systems. Because in the *file-I/O* based implementation of GraphOne, the virtual file system (VFS) cost, metadata management cost, and log data management cost may account for a large portion of the total time cost (refer to Figure.10 in [79]), thus leading to the poor performance. As far as we think, the *mmap* based implementation, in which the user application takes the responsibility to manage its own address space of PMEM, is more suitable for graph processing situations for the light-weighted management cost.

For the other graph systems, compared with GraphOne-P, XPGraph is consistently faster and achieves 3.01× to 3.95× speedup for different graphs, as XPGraph greatly reduces the PMEM access cost by our proposed three techniques. Besides, with a system-level battery supported, XPGraph-B can further improve the performance by up to 23% on top of XPGraph, meaning that the optimizations of XPGraph also apply to battery-backed systems.

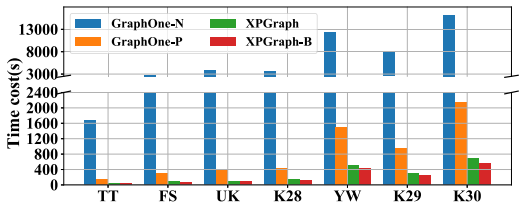


Figure 11. Graph ingestion time cost for non-volatile systems, where Optane is set in application-direct mode.

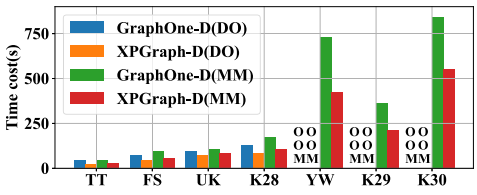
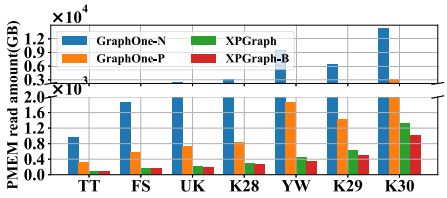


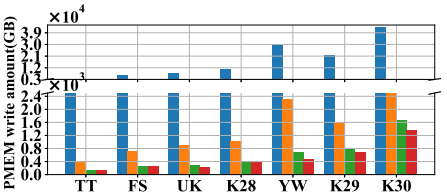
Figure 12. Graph ingestion time cost for volatile systems, where DO indicates running on a DRAM-only system, and MM indicates running on a PMEM-based system with Optane set in memory mode.

Ingestion time cost for volatile systems. We also study the performance on different hardware settings to show the generality of our work. Specifically, we compare the performance of GraphOne-D and XPGraph-D (see §IV-C), running on a DRAM-only system, and a PMEM-based system with Optane in memory mode, respectively. Note that, these systems assume no crash consistent requirements. We show the results in Fig.12. Both of GraphOne-D and XPGraph-D fail to handle the three larger graphs when they run on the DRAM-only setting, because of the limited capacity of DRAM as they store all data in DRAM. For the other tested results, XPGraph-D always performs faster than GraphOne-D: the speedup is up to 73% for DRAM-only systems and 76% for PMEM-based systems with Optane in memory mode.

PMEM read and write data amount. We further measure the data amount read from and written to the PMEM during the ingestion process of GraphOne-P, GraphOne-N and our XPGraph, XPGraph-B, respectively, by using the Intel PCM tool [53], to validate the greatly reduced PMEM access cost of XPGraph. We show the result in Fig.13. We can see that GraphOne-N is still always an order of magnitude worse compared with other three systems, as explained before. Compared with GraphOne-P, XPGraph greatly reduced the amount of PMEM read data by 2.29× to 4.17× and PMEM write data by 2.02× to 3.44× for different graphs, which mainly benefits from the vertex-centric local batched graph archiving (refer to §III-B), and this benefit finally achieves a great contribution to the overall performance introduced in Fig.11. XPGraph-B further reduces the PMEM read data amount by up to 31% and PMEM write data amount by up to 47% on top of XPGraph.



(a) PMEM read amount.

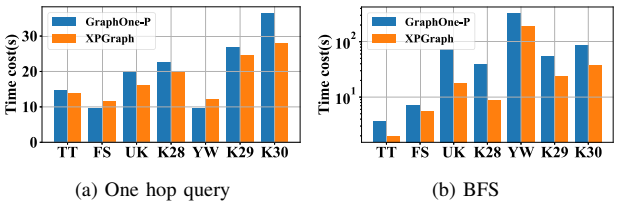


(b) PMEM write amount.

Figure 13. PMEM read and write data amount.

C. Graph Query Performance

Fig.14 shows the graph query performance of GraphOne-P and XPGraph with all available threads of our server, i.e., 96 threads. The four sub-figures show the time cost of completing the four graph algorithms introduced in §V-A. We can see that for the simple one-hop neighbor query, performance behaviors of GraphOne-P and XPGraph vary from graph to graph. For some cases XPGraph runs faster, and for other cases GraphOne-P runs faster, but in most cases, they need comparable time cost to finish the queries with the performance gap limited in 30%. As for the graph analytic algorithms BFS, PageRank, and CC, XPGraph often provides better support, and achieves up to 4.46×, 3.57×, and 4.23× speedup for BFS, PageRank, and CC respectively. This performance improvement comes from two aspects: First, GraphOne-P stores all neighbor information in PMEM, while XPGraph buffers some recent neighbor information in DRAM vertex buffers, thus reducing the data amount needed to fetch from the persistent adjacency lists stored in PMEM. Second, XPGraph adopts the NUMA-friendly graph access strategy (see §III-D), which not only evenly distributes the PMEM queries to different sockets by the sub-graph based NUMA-aware segregated graph storing, but also avoids the remote PMEM reads across NUMA by the CPU-binding based graph querying, thus improving the PMEM data read efficiency and graph query performance.



(a) One hop query (b) BFS

(c) PageRank (d) CC

Figure 14. Graph query performance (BFS, PageRank and CC are shown in log scale).

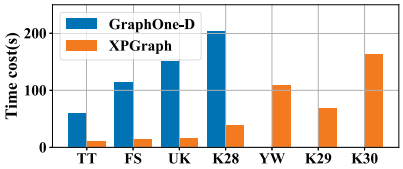
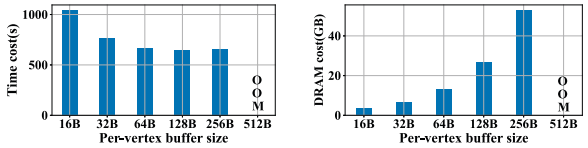


Figure 15. Graph recovery performance.



(a) Time cost (b) DRAM demand

Figure 16. Impact of vertex-centric graph buffering with different per-vertex buffer sizes, where OMM indicates the out-of-memory error.

D. Graph Recovery Performance

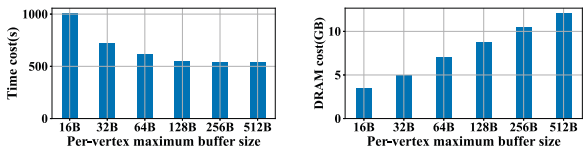
PMEM is also capable of data persistence, which provides us an opportunity to realize the fast graph recovery after a power failure. Therefore, we also implement a simple recovery scheme to reload the graph data from persistent adjacency lists stored in PMEM, and update the pointer links between multiple adjacency lists of the same vertex, thus making it capable of ingesting new edge updates and querying graph data as usual. We also compare the recovery performance with GraphOne. Note that, GraphOne's recovery process is to re-building the data structure, by just running the archiving process worked on bulk of data [36]. Therefore, we test GraphOne's recovery performance by testing the graph archiving performance with the archiving threshold set as 2^{27} edges, as recommended in its paper to achieve the best performance.

Fig.15 shows the graph recovery time costs, which include the time cost of loading data from PMEM and recovering the adjacency list structure for all vertices, of XPGraph and GraphOne for different datasets. XPGraph always provides better support for recovery performance, as it does not need to re-build the data structure for all edges. In summary, XPGraph achieves 5.20× to 9.47× higher recovery performance, compared with GraphOne for the four relatively small graphs. For the larger three graphs that can not run out on GraphOne-D, XPGraph can also realize the recovery in a reasonable time. For example, XPGraph costs only 163.66 seconds to recover the largest dataset Kron30, while GraphOne takes 204.45 seconds to recover the much smaller graph Kron28 with only 1/16 edges of Kron30.

E. Discussion of Design Choices

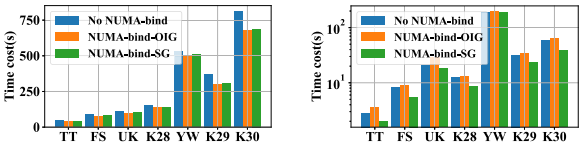
Efficiency of vertex-centric graph buffering. First, we measure the benefit of the vertex-centric graph buffering strategy (see §III-B). We implement this strategy with a predefined per-vertex buffer size. When each vertex buffer size is set as 8 bytes, we can cache 1 neighbor per vertex. When the per vertex buffer size is set as 256 bytes, up to 63 neighbors can be cached for each vertex. We record the time cost and DRAM space requirements of these vertex buffers, for importing the Yahoo Web dataset under different buffer size settings, and show the results in Fig.16(a) and Fig.16(b), respectively. They show that by buffering some edge updates in DRAM, we can greatly reduce the time cost of ingesting the dynamic graph. And we found that the larger, the buffer size is set, the more neighbors each vertex can cache in DRAM, then the less total number of writes to PMEM, and thus contributing to the lower the overall time cost. However, at the same time, it also requires more DRAM space for larger buffers. When the buffer size of each vertex is set to 256 bytes, the overall DRAM space demand is even more than 52 GB for Yahoo Web, and when the buffer size is set to 512 bytes, it even fails to run out for out-of-memory error. This severely limits the scalability for large-scale graph processing. And the high memory allocation cost for these vertex buffers may degrade the performance on the contrary, thus causing the slight performance drop when increasing the per-vertex buffer size setting from 128 bytes to 256 bytes.

Efficiency of adaptive hierarchical buffering. We further propose a hierarchical vertex buffer managing strategy (see §III-C). We also measure its benefit by the same settings above and show the results in Fig.17. We can see that after adopting the hierarchical vertex buffer managing strategy, we can maintain the same performance gains as we allocate the maximum buffers for all vertices, but make the DRAM space requirement be largely reduced. For example, when we set a fixed buffer sized 128-bytes for all vertices, we cost around 645.42 seconds for importing the Yahoo Web dataset, and require around 26.54 GB DRAM space for storing all vertex buffers. Note that, it is the best performance for fixed buffer size setting as shown in Fig.16. While when we set a hierarchical buffer sized from 16-bytes to 256-bytes according to the vertex degree changes, we cost only 544.72 seconds, which is even faster than before because of the lower time overhead of DRAM allocation, and the DRAM space requirement is reduced to around 10.49 GB, which is less than a half of before. Besides, we can also adjust the DRAM space requirement by limiting the size of the maximum buffer in the limited DRAM capacity situations.



(a) Time cost (b) DRAM demand

Figure 17. Efficiency of hierarchical vertex buffer managing with different per-vertex maximum buffer sizes.



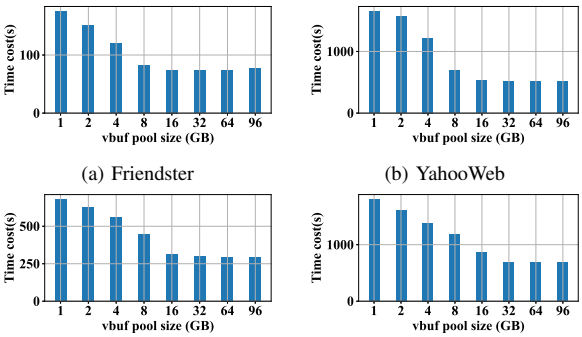
(a) Graph ingestion time (b) BFS time

Figure 18. Efficiency of NUMA-friendly graph accessing (BFS time is shown in log scale), where NUMA-bind-OIG indicates the out/in-graph-based NUMA-binding implementation, and NUMA-bind-SG indicates the subgraph-based NUMA-binding implementation.

Efficiency of NUMA-friendly graph accessing. Fig.18 shows the graph ingesting and BFS computing times under three settings: no NUMA-binding, out/in-graph based NUMA binding, and subgraph based NUMA binding. With the NUMA-friendly graph accessing technique enabled, we can further improve the graph ingestion performance by 5% to 23% on top of the performance gains of the above two techniques, and this improvement is also getting larger for the larger graphs for it can avoid more cross-NUMA PMEM accesses. The two versions of implementations performs similar for graph ingestion. As for graph query performance, the out/in-graph based NUMA binding implementation may cause the performance drop by 3% to 29% on top of the baseline, because of the load-imbalance issues. While the subgraph based NUMA binding implementation can improve the BFS performance by up to 54%, because of the avoided remote PMEM accesses across NUMA and the load-balanced sub-graph partitioning between different NUMA nodes.

F. Impact of system configurations

Impact of vertex buffer memory pool size. We also conduct experiments to evaluate the impact of vertex buffer memory pool sizes, to show the performance sensitivity of our XPGraph. We evaluate the graph ingestion cost under different memory pool size settings, for Friendster, YahooWeb, Kron29 and Kron30, respectively, and show the results in Fig.19. We can see that, as the memory pool size



(a) Friendster (b) YahooWeb

(c) Kron29 (d) Kron30

Figure 19. Impact of vertex buffer memory pool size.

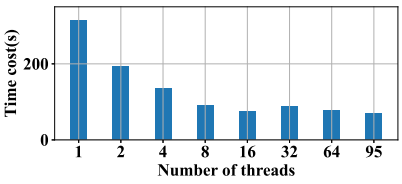


Figure 20. Impact of number of archiving threads.

set larger from 1 GB to 16 GB, the overall time cost decreases significantly, because more memory pool space provides more chance to cache and merge more PMEM accesses. When the memory pool size goes from 32 GB to 96 GB, the time cost changes slightly, even for the three largest graphs. Because 32 GB is generally enough to hold most vertex buffers for these graphs. Besides, extra space is not actually allocated for the copy-on-write technique used in Linux system, thus does not impact the performance. Therefore, we recommend a larger memory pool size setting for better performance.

Impact of the number of archiving threads. Next, we observe the impact of the number of archiving threads, i.e., number of threads used for converting the graph data from edge list format to the adjacency list format, for XPGraph. Fig.20 shows the ingestion time costs of Friendster for different number of archiving threads settings. We can see that in general, XPGraph's ingesting performance increases with the thread number increases, and achieves the peak performance by setting the number of archiving threads as 95, which is the maximum available number of threads in our server (the other one thread is used for edge update logging). Compared with the result for GraphOne-P shown in Fig.4(b), XPGraph achieves a much better scalability for multi-threading.

Scalability. To analyze the scalability of our design, we also show the memory usage breakdown during the ingest process of the tested seven graphs in Table III. Meta indicates the DRAM usage for storing the meta-data and intermediate data, i.e., vertex indexes, snapshots information and temporary edge lists. Note that this part of DRAM usage is inherited from GraphOne [36], and it is similar to that of GraphOne. These meta-data and intermediate data may consume a lot of DRAM space and limit the scalability of XPGraph, and we will consider moving them to PMEM to further improve the scalability. Vbuf indicates the DRAM usage for storing the vertex buffers, which are managed by the vertex buffer memory pool. This part of DRAM usage can be further limited by the memory pool size setting, refer to Fig.19. Input indicates the PMEM usage for storing the input graph data, which is stored in the binary edge list format. Elog indicates the PMEM usage for storing the circular edge log, which is set as 8 GB by default. Pblk indicates the PMEM usage for storing the persistent adjacency lists, which contain the key graph information of XPGraph.

Table III
MEMORY USAGE OF XPGRAPH (GB).

Dataset	DRAM		PMEM		
	Meta	Vbuf	Input	Elog	Pblk
TT	6.62	5.11	10.94	8.00	13.61
FS	12.11	6.86	19.27	8.00	30.31
UK	16.60	7.08	24.60	8.00	37.19
YW	55.93	10.59	49.57	8.00	116.70
K28	15.72	8.91	32.00	8.00	41.97
K29	27.65	16.89	64.00	8.00	82.67
K30	49.54	28.22	128.00	8.00	165.95

We can see that, XPGraph can efficiently handle large graphs that fit in PMEM capacity with a limited and tunable DRAM usage. For example, the largest test graph Kron30, with billions of vertices and tens billions of edges, consumes about 80 GB DRAM space and 300 GB PMEM space in total, which can be supported in most medium-sized servers. In most enterprise-scale servers with larger memory space, such as terabytes of PMEM per socket [51], XPGraph can support a large portion of large-scale graph processing applications. For larger graphs that can not fit in PMEM, we will consider extending the SSD-supported XPGraph and distributed XPGraph in future work.

VI. RELATED WORK

Large-scale graph processing. Traditional in-memory graph processing systems usually use the adjacency list liked formats, for efficient graph queries [29], [49], [63]. However, their scalabilities are limited by the DRAM capacity. To support efficient graph processing for large-scale graphs that can not reside on a single machine's memory, many distributed graph systems have been proposed to process graph computation on a cluster of machines [7], [8], [25], [26], [35], [42], [46], [66], [83], [84]. On the other hand, disk-resident single machine graph processing also receives a lot of attention by storing graphs on external storage devices [1], [17], [23], [33], [38], [40], [44], [57], [71], [72], [85]. However, these systems are usually designed for static graphs and require efficient graph partitioning. Besides, they often suffer from performance drops due to the high cost of communication between machines or I/O from disk. Therefore, our XPGraph first introduces a PMEM-based graph store, which can realize both good scalability and high performance graph processing.

Dynamic graph stores. Dynamic graph processing is an important research field, which can execute graph analysis concurrently with graph updates, and many dynamic graph frameworks have been developed in recent years, such as STINGER [22], GraphIn [61], EvoGraph [60], Hornet [4], GraphOne [36], and so on [14], [28], [32], [47], [62], [75], [76]. The majority of these frameworks usually combine the adjacency list or the edge list that consists of chunks, thus enabling a trade-off between the locality of accesses and time to perform updates [4], [22], [36], [60], [61]. And some frameworks use some form of batching the updates to increase the parallelism and ultimately the throughput of graph accesses [4], [6], [22], [45]. However, these frameworks are usually designed for DRAM-based dynamic graph stores, which often brings many small random accesses during the process of updating the adjacency lists for different vertices, thus causing access efficiency issues when moving these frameworks to a PMEM-based system. Therefore, our XPGraph proposes a PMEM-friendly graph accessing model, to support high performance dynamic graph stores for large-scale graphs.

PMEM-based storage optimizations. Persistent memory has been well studied in the past decade [20], [65], [70], even before the release of real industrial products. Because of the different characteristics between DRAM and PMEM, e.g., Intel Optane Persistent Memory's performance characterization is complicated and fluctuated by many factors, including access type (read vs. write), access pattern (sequential vs. random), access size, and so on [81], many recent persistent memory works are proposed to carefully manage the data in PMEM to realize good performance, including PMEM allocators [3], [5], [18], [43], [52], [59], PMEM indexes [9], [11], [31], [41], PMEM based key-value stores [12], [30], [77], PMEM based file systems [10], [13], [20], [78], and so on. While graph structure data and graph processing applications are totally different from the above situations, which pushes us to design a dedicated PMEM based graph storage management for efficient large-scale graph processing.

VII. CONCLUSION

In this paper, we proposed XPGraph, which is an XPLine-friendly PMEM-based graph storage system, targeting to realize the high-performance stores for large-scale evolving graphs. XPGraph carefully manages the processes of flushing graph data to PMEM, by vertex-centric graph buffering, hierarchical vertex buffer managing, and NUMA-friendly graph accessing. Our experimental results show that XPGraph can largely reduce the PMEM access cost for graph data, and improves the graph update performance, graph query performance, as well as graph recover performance. We also provide data access APIs encapsulated into a library, for easy use of XPGraph by user applications.

ACKNOWLEDGMENT

We would like to thank the anonymous shepherd and reviewers for their comments. We also thank Xuechen Zhang and Jian Xu for their help to improve this work. This work is supported in part by the National Key Research and Development Program of China (2021ZD0110700), the National Science Foundation of China (62172361), the Program of Zhejiang Province Science and Technology (2022C01044), the Zhejiang Lab Research Project (2020KC0AC01), and the China Postdoctoral Science Foundation (2022M712755).

APPENDIX

A. Abstract

The artifact contains the prototype implementations of XPGraph, as well as its two variants, i.e., XPGraph-B and XPGraph-D. We open-source XPGraph and provide some graph view interfaces to allow other researchers and developers to use and improve it in their own work. We also provide the implementations of the comparison baselines, i.e., GraphOne-D, GraphOne-P, and GraphOne-N. This artifact should enable others to reproduce a subset of our results and conduct their own studies on PMEM-based graph processing.

B. Artifact check-list (meta-information)

- **Program:** Linux kernel of 5.10.0., ndctl tool, PMDK libpmem library, XPGraph, GraphOne.
- **Compilation:** GCC 10.2.0.
- **Data set:** Four real-world graphs Twitter, Friendster, UKdomain and Yahoo Web, as well as three synthetic Kronecker graphs, i.e., Kron-28, Kron-29 and Kron-30, which are generated by graph500 generator.
- **Run-time environment:** Ubuntu 18.04.
- **Hardware:** A server with two processors configured in a non-uniform memory access (NUMA) architecture, and with Intel Optane Persistent Memory 200 Series equipped.
- **Execution:** Automated by shell scripts.
- **Metrics:** Graph ingesting, query and recovery performance.
- **Output:** The key results would be recorded to the directory *results/*, and detailed run-time state would be printed to the command window.
- **Experiments:** Graph ingest time cost for GraponeOne-P, XPGraph and XPGraph-B (Fig.11). Graph ingest time cost for GraphOne-D, XPGraph-D (Fig.12). Graph query performance for GraphOne-P and XPGraph (Fig.14). Graph recovery performance for GraphOne-D and XPGraph (Fig.15).
- **How much disk space required (approximately)?:** 1 TB.
- **How much time is needed to prepare workflow (approximately)?:** 48 hours for downloading and preprocessing all graph datasets.
- **How much time is needed to complete experiments (approximately)?:** 48 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Version v1.0.4
- **Workflow framework used?:** No, but scripts are provided to automate the measurements.
- **Archived (provide DOI)?:** 10.5281/zenodo.6997642

C. Description

1) How to access: The source code and scripts are host on Zenodo https://zenodo.org/record/6997642#.Yvtnky-KFpQ.

2) Hardware dependencies: This artifact runs on a server with two processors configured in a non-uniform memory access (NUMA) architecture. Each processor has 24 physical cores with hyper-threading enabled (48 logical cores). For memory, it equips with 8 $\times$ 16 GB (128 GB) DRAM and 8 $\times$ 128 GB (1 TB) Intel Optane Persistent Memory 200 Series, which are interleaved inserted to the memory slot.

3) Software dependencies: This artifact runs on Ubuntu 18.04.6 LTS with Linux kernel of 5.10.0, and we use GCC 10.2.0 with -O3 optimization for evaluation. Users also need to install ndctl tool for managing the PMEM mode in the Linux kernel, and PMDK libpmem library for allocating PMEM space.

4) Data sets: We use four real-world graphs Twitter, Friendster, UKdomain and Yahoo Web, as well as three synthetic Kronecker graphs, i.e., Kron-28, Kron-29 and Kron-30, which are generated by graph500 generator, for our evaluation. Data sets download links:

- **Twitter:** http://an.kaist.ac.kr/traces/WWW2010.html
- **Friendster:** http://konect.uni-koblenz.de/networks/friendster
- **UKdomain:** http://konect.cc/networks/dimacs10-uk-2007-05
- **Yahoo Web:** http://webscope.sandbox.yahoo.com

Graph500 generator link and Kronecker graph generation commands:

- **Generator link:** https://github.com/rwang067/graph500-3.0
- **Make genetator:** cd graph500-3.0/src && make graph500_reference_bfs
- **Generate Kron28:** ./graph500_reference_bfs 28 16 kron28_16.txt
- **Generate Kron29:** ./graph500_reference_bfs 29 16 kron29_16.txt
- **Generate Kron30:** ./graph500_reference_bfs 30 16 kron30_16.txt

D. Installation

Users need to download the source code and scripts from Zenodo to the server. The following is the directory structure of the source code, scripts, and instructions:

- **README.md:** This file contains a detailed step-by-step "Try out XPGraph" guide.
- **src/:** This directory contains the core source code of XPGraph implementations.
- **apps/:** This directory contains the graph query algorithms implemented on top of XPGraph.
- **baselines/:** This directory contains the zipped source codes of the comparison baselines, i.e., GraphOne-D, GraphOne-P, and GraphOne-N.
- **scripts/:** This directory has scripts to run experiments.
- **preprocess/:** This directory has scripts to preprocess the graph datasets.

After downloading the source code and scripts, users need to compile XPGraph and prepare graph datasets. Please see 'README.md' for detailed guide. To evaluate the comparison baselines, users also need to unzip and compile them, please see their corresponding README.md for detailed guide.

E. Experiment workflow

The suggested workflow is organized in *scripts/*, which contains the guide for configuring Optane PMEM (see *nvdimm.md*) and the shell scripts. Users can use the command 'cd XPGraph && bash scripts/run.sh' for running all default experiments. The running progress and the expected completing time of each experiment would be printed to the file *scripts/progress.txt* automatically. Note that we run each experiment 10 times and calculate the average values for a more accurate display of time cost. Users can modify it in the shell scripts.

F. Evaluation and expected results

The evaluation results would be generated to the directory *results/*, which would classified by figures in our paper. Users can reproduce the results in Fig.11, Fig.12, Fig.14 and Fig.15, which should roughly match the respective figures from the paper. Note that, the results may be not the exact ones presented in the paper. Because in our subsequent experiments we found that different states of the PMEMs may also impact the performance.

G. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng, "Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o," in *USENIX ATC*, 2017, pp. 125–137.

[2] Y. Akhremtsev, P. Sanders, and C. Schulz, "High-quality shared-memory graph partitioning," *IEEE TPDS*, vol. 31, no. 11, pp. 2710–2722, 2020.

[3] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 677–694, 2016.

[4] F. Busato, O. Green, N. Bombieri, and D. A. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus," in *IEEE HPEC*, 2018, pp. 1–7.

[5] W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati, and M. L. Scott, "Understanding and optimizing persistent memory allocation," in *ACM SIGPLAN ISMM*, 2020, pp. 60–73.

[6] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE TCDE*, vol. 36, no. 4, 2015.

[7] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng, "G-miner: an efficient task-oriented graph mining system," in *ACM EuroSys*, 2018, pp. 1–12.

[8] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *ACM EuroSys*, 2015, pp. 1–39.

[9] S. Chen and Q. Jin, "Persistent b+-trees in non-nolatile main memory," *PVLDB*, vol. 8, no. 7, pp. 786–797, 2015.

[10] Y. Chen, Y. Lu, P. Chen, and J. Shu, "Efficient and consistent nvmm cache for ssd-based file system," *IEEE TC*, vol. 68, no. 8, pp. 1147–1158, 2018.

[11] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu, "Utree: a persistent b+-tree with low tail latency," *PVLDB*, vol. 13, no. 12, pp. 2634–2648, 2020.

[12] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *ACM ASPLOS*, 2020, pp. 1077–1091.

[13] Y. Chen, J. Shu, J. Ou, and Y. Lu, "Hinfs: A persistent memory file system with both buffering and direct-access," *ACM ToS*, vol. 14, no. 1, pp. 1–30, 2018.

[14] 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 *ACM EuroSys*, 2012, pp. 85–98.

[15] C. Chevalier and F. Pellegrini, "Pt-scotch: A tool for efficient parallel graph ordering," *Parallel computing*, vol. 34, no. 6-8, pp. 318–331, 2008.

[16] A. Ching, "Giraph: Production-grade graph processing infrastructure for trillion edge graphs," *ATPESC*, vol. 14, 2014.

[17] D. M. Da Zheng, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," in *USENIX FAST*, 2015, pp. 45–58.

[18] Z. Dang, S. He, P. Hong, Z. Li, X. Zhang, X.-H. Sun, and G. Chen, "Nvalloc: Rethinking heap metadata management in persistent memory allocators," in *ACM ASPLOS*, 2022, pp. 115–127.

[19] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: Distributed transactions with consistency, availability, and performance," in *ACM SOSP*, 2015, pp. 54–70.

[20] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *ACM EuroSys*, 2014, pp. 1–15.

[21] "eadr: New opportunities for persistent memory applications," https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[22] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *IEEE HPEC*, 2012, pp. 1–5.

[23] N. Elyasi, C. Choi, and A. Sivasubramaniam, "Large-scale graph processing on emerging storage devices," in *USENIX FAST*, 2019, pp. 309–316.

[24] "Friendster," http://konect.uni-koblenz.de/networks/friendster.

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *USENIX OSDI*, 2012, pp. 17–30.

[26] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *USENIX OSDI*, 2014, pp. 599–613.

[27] "Graph500," https://graph500.org/.

[28] O. Green and D. A. Bader, "custinger: Supporting dynamic graph algorithms for gpus," in *IEEE HPEC*, 2016, pp. 1–6.

[29] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: a dsl for easy and efficient graph analysis," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 349–362, 2012.

[30] Y. Huang, M. Pavlovic, V. Marathe, M. Seltzer, T. Harris, and S. Byan, "Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs," in *USENIX ATC*, 2018, pp. 967–979.

[31] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," in *USENIX FAST*, 2018, pp. 187–200.

[32] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *ACM GRADES*, 2016, pp. 1–6.

[33] S.-W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind, "Grafboost: Using accelerated flash storage for external graph analytics," in *IEEE ISCA*, 2018, pp. 411–424.

[34] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger, "Viyojit: Decoupling battery and dram capacities for battery-backed dram," in *IEEE ISCA*, 2017, pp. 613–626.

[35] A. Khan, G. Segovia, and D. Kossmann, "On smart query routing: for distributed graph querying with decoupled storage," in *USENIX ATC*, 2018, pp. 401–412.

[36] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," in *USENIX FAST*, 2019, pp. 249–263.

[37] A. Kyrola, "Drunkardmob: Billions of random walks on just a pc," in *ACM RecSys*, 2013, pp. 257–264.

[38] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *USENIX OSDI*, 2012, pp. 31–46.

[39] "Linux-nova," https://github.com/NVSL/linux-nova.

[40] H. Liu and H. H. Huang, "Graphene: Fine-grained io management for graph computing," in *USENIX FAST*, 2017, pp. 285–300.

[41] J. Liu, S. Chen, and L. Wang, "Lb+ trees: Optimizing persistent index performance on 3dxpoint memory," *PVLDB*, vol. 13, no. 7, pp. 1078–1090, 2020.

[42] 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," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[43] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu, "Roart: Range-query optimized persistent art," in *USENIX FAST*, 2021, pp. 1–16.

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

[45] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *IEEE ICDE*, 2015, pp. 363–374.

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

[47] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *ACM SOSP*, 2013, pp. 439–455.

[48] "Neo4j inc," https://neo4j.com/.

[49] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *ACM SOSP*, 2013, pp. 456–471.

[50] "Intel optane persistent memory," https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[51] "Achieve greater insight from your data with intel optane persistent memory," https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html.

[52] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes, "Memory management techniques for large-scale persistent-main-memory systems," *PVLDB*, vol. 10, no. 11, pp. 1166–1177, 2017.

[53] "Intel. processor counter monitor (pcm)," https://github.com/opcm/pcm.

[54] "Persistent memory development kit (pmdk)," https://github.com/opcm/pcm.

[55] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haradasan, "Managing large graphs on multi-cores with graph awareness," in *USENIX ATC*, 2012, pp. 41–52.

[56] "pthread_setaffinity_np(3) — linux manual page," https://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html.

[57] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *ACM SOSP*, 2013, pp. 472–488.

[58] P. Sanders and C. Schulz, "Engineering multilevel graph partitioning algorithms," in *ACM ESA*. Springer, 2011, pp. 469–480.

[59] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "Nvm malloc: Memory allocation for nvram," *VLDB*, vol. 15, pp. 61–72, 2015.

[60] D. Sengupta and S. L. Song, "Evograph: On-the-fly efficient mining of evolving graphs on gpu," in *ACM ISC*, 2017, pp. 97–119.

[61] 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 *Euro-Par*, 2016, pp. 319–333.

[62] X. Shi, B. Cui, Y. Shao, and Y. Tong, "Tornado: A system for real-time iterative analysis over evolving data," in *ACM SIGMOD*, 2016, pp. 417–430.

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

[64] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *ACM SIGKDD*, 2012, pp. 1222–1230.

[65] Y. Tai, J. Yang, X. Liu, and C. Xu, "Memnet: A persistent memory network for image restoration," in *IEEE ICCV*, 2017, pp. 4539–4547.

[66] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: A system for distributed graph mining," in *ACM SOSP*, 2015, pp. 425–440.

[67] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *ACM WSDM*, 2014.

[68] "Twitter," http://an.kaist.ac.kr/traces/WWW2010.html.

[69] "Uk domain (2007) network dataset – KONECT," Jan. 2018. [Online]. Available: http://konect.cc/networks/dimacs10-uk-2007-05

[70] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH CAN*, vol. 39, no. 1, pp. 91–104, 2011.

[71] K. Vora, "Lumos: Dependency-driven disk-based graph processing," in *USENIX ATC*, 2019, pp. 429–442.

[72] K. Vora, G. H. Xu, and R. Gupta, "Load the edges you need: A generic i/o optimization for disk-based graph processing," in *USENIX ATC*, 2016, pp. 507–522.

[73] Q. Wang, Y. Lu, J. Li, and J. Shu, "Nap: A black-box approach to numa-aware persistent memory indexes," in *USENIX OSDI*, 2021, pp. 93–111.

[74] R. Wang, Y. Li, H. Xie, Y. Xu, and J. C. Lui, "Graphwalker: An i/o-efficient and resource-friendly graph analytic system for fast and scalable random walks," in *USENIX ATC*, 2020, pp. 559–571.

[75] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "Faimgraph: High performance management of fully-dynamic graphs under tight memory constraints on the gpu," in *IEEE SC*, 2018, pp. 754–766.

[76] M. Winter, R. Zayer, and M. Steinberger, "Autonomous, independent management of dynamic graphs on gpus," in *IEEE HPEC*, 2017, pp. 1–7.

[77] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *USENIX ATC*, 2017, pp. 349–362.

[78] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *USENIX FAST*, 2016, pp. 323–338.

[79] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *ACM SOSP*, 2017, pp. 478–496.

[80] "Yahoo webscope. yahoo! altavista web page hyperlink connectivity graph," http://webscope.sandbox.yahoo.com.

[81] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *USENIX FAST*, 2020, pp. 169–182.

[82] K. Yang, X. Ma, S. Thirumuruganathan, K. Chen, and Y. Wu, "Random walks on huge graphs at cache efficiency," in *ACM SOSP*, 2021, pp. 311–326.

[83] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *USENIX OSDI*, 2016, pp. 301–316.

[84] X. Zhu, G. Feng, S. Marco, X. Ma, J. Yu, L. Xie, A. Ashraf, and W. Chen, "Livegraph: A transactional graph storage system with purely sequential adjacency list scans," *PVLDB*, vol. 13, no. 7, pp. 1020–1034, 2020.

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