GEAR: Graph-Evolving Aware Data Arranger to Enhance the Performance of Traversing Evolving Graphs on SCM

Wen-Yi Wang, Chun-Feng Wu, *Member, IEEE*, Yun-Chih Chen, *Member, IEEE*, Tei-Wei Kuo, *Fellow, IEEE*, and Yuan-Hao Chang, *Fellow, IEEE*

***Abstract*—In the era of big data, social network services continuously modify social connections, leading to dynamic and evolving graph data structures. These evolving graphs, vital for representing social relationships, pose significant memory challenges as they grow over time. To address this, storage-class-memory (SCM) emerges as a cost-effective solution alongside DRAM. However, contemporary graph evolution processes often scatter neighboring vertices across multiple pages, causing weak graph spatial locality and high-TLB misses during traversals. This article introduces SCM-Based graph-evolving aware data arranger (GEAR), a joint management middleware optimizing data arrangement on SCMs to enhance graph traversal efficiency. SCM-based GEAR comprises multilevel page allocation, locality-aware data placement, and dual-granularity wear leveling techniques. Multilevel page allocation prevents scattering of neighbor vertices relying on managing each page in a finer-granularity, while locality-aware data placement reserves space for future updates, maintaining strong graph spatial locality. The dual-granularity wear leveler evenly distributes updates across SCM pages with considering graph traversing characteristics. Evaluation results demonstrate SCM-based GEAR's superiority, achieving 23% to 70% reduction in traversal time compared to state-of-the-art frameworks.**

***Index Terms*—Checkpointing, evolving graph, graph, HW/SW Co-design, memory management, middleware, non-volatile memory, system software.**

Manuscript received 4 August 2024; accepted 4 August 2024. This work was supported in part by the National Science and Technology Council under Grant 113-2628-E-A49-021, Grant 112-2222-E-A49-002-MY2, Grant 113-2223-E-001-001, Grant 111-2221-E-001-013-MY3, Grant 113-2927-I-001-502, and Grant 111-2923-E-002-014-MY3; in part by the Academia Sinica under Grant AS-IA-111-M01; and in part by the Ministry of Education under Yushan Young Fellow Program. This article was presented in the International Conference on 2024 and appears as part of the ESWEEK-TCAD special issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding authors: Chun-Feng Wu; Yuan-Hao Chang.)*

Wen-Yi Wang is with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan (e-mail: r09922090@ntu.edu.tw).

Chun-Feng Wu is with the Department of Computer Science, National Yang Ming Chiao Tung University, Hsinchu 300, Taiwan (e-mail: cfwu417@cs.nycu.edu.tw).

Yun-Chih Chen is with the Department of Computer Science, Technische Universitat Dortmund, 44227 Dortmund, Germany (e-mail: yunchih.chen@tu-dortmund.de).

Tei-Wei Kuo is with the Delta Electronics, Department of Computer Science and Information Engineering, High Performance and Scientific Computing Center, and the Center of Data Intelligence: Technologies, Applications, and Systems, National Taiwan University, Taipei 10617, Taiwan (e-mail: ktw@csie.ntu.edu.tw).

Yuan-Hao Chang is with the Institute of Information Science, Academia Sinica, Taipei 115, Taiwan (e-mail: johnson@iis.sinica.edu.tw).

Digital Object Identifier 10.1109/TCAD.2024.3447222

I. Introduction

SOCIAL network services utilize graph data structures to manage the connections between users. The relationship is highly dynamic, with connections added, updated, or removed at all times [1], [2]. As a result, the underlying graph data structures are dynamic and change with time. These dynamic graphs are known as evolving graphs: the connection between any two users may not be static all the time [3]. As evolving graphs grow over time, so does the system's memory demand. Storage-class memory (SCM) [4], [5], [6], [7] can augment DRAM to provide larger memory space at a lower price, alleviating the need to constantly add more DRAM to meet memory demand. Recent works in graph processing propose to buffer graph updates in RAM before flushing them to SCMs in batch [8], [9]. Our investigation reveals that such batch updates can be inefficient, with vertice updates spread across multiple batches and neighboring vertices spread across multiple pages. As a result, this characteristic leads to weak graph spatial locality, which may result in high-translation lookaside buffer (TLB) misses during subsequent graph traversals. To improve graph traversing performance, this work proposes a joint management middleware that take graph spatial locality into account in the data placement policy on SCMs.

Major social network providers, such as Google [10], Meta [11], and JingDong (JD.com) [2], have adopted graph processing algorithms, such as page rank and graph neural networks, to extract information from Web pages and social networks. A distributed system is one option for storing all graph data in memory. However, building an efficient distributed system remains a challenge, especially for small companies, due to high deployment and maintenance costs, load balancing, and fault tolerance. Out-of-core systems are alternative architectures that run graph processing on a single consumer-level machine, supplementing limited memory capacity with storage devices. Graph processing system based on out-of-core architecture have gained significant attention in the community [12], [13]. GraphChi [14] proposed breaking down large graphs into small parts and storing them in storage devices. Several works (e.g., FlashGraph [15], Graphene [16], and GraphSSD [17]) have proposed to carefully manage Solid-State Drives by adopting some I/O request merging or sophisticated buffering approaches with considering graph access behaviors. In contrast to processing static graphs, these systems need a new data structure to track the most recent version of each vertex and edge in evolving graphs. Section II will provide a comprehensive overview of cutting-edge solutions and challenges.

However, we observed that state-of-the-art evolving graph frameworks have poor graph spatial locality, which makes them inefficient in executing graph traversal algorithms. We proposed a joint management middleware between graph-evolving processing and memory devices (including both DRAM and SCMs), called the SCM-Based Graph-Evolving Aware Data ArrangeR (GEAR). Our goal is to arrange and write the evolving graph data into SCMs while achieving strong graph spatial locality. The SCM-Based GEAR has three major components: 1) multilevel page allocation; 2) locality-aware data placement; and 3) a dual-granularity wear leveler.

1) The main idea behind multilevel page allocation is to prevent graph-evolving processes from scattering neighbor vertices across different pages. Technically, the system maintains multilevel size subpages and assigns a suitable-size subpage to accommodate all neighbors associated with each vertex, taking into account their number.
2) The locality-aware data placement reserves an unused area in each subpage for future graph updates to the corresponding vertex, ensuring strong graph spatial locality even as the graph evolves over time.
3) The dual-granularity wear leveler, in conjunction with our page allocation, distributes graph updates evenly across all memory pages on SCM during graph evolution. The evaluation results show that, compared to the state-of-the-art frameworks, our SCM-based GEAR can save the total execution time by 23%–70% when traversing an evolving graph.

The remainder of this article is organized as follows. Section II elaborates the graph evolving processes and shows the impact of the weak graph spatial locality on the graph traversal time. Section III provides the design concept and implementation of the SCM-based GEAR. Section IV evaluates the proposed strategy. Finally, Section V concludes this article.

II. Background, Observation, and Motivation

A. Background

1) Evolving Graphs: Graphs are commonly used to represent the relationship between data points. In general, each node in a graph represents a data point, and the edge that connects two data points (or nodes) records their relationship. A graph is considered a *evolving graph* if its layout or edge weights change over time. Social networks, for example, are constantly evolving [18], [19] as new users join and connections are established frequently. To analyze an evolving graph over time, evolving graph processing systems take snapshots on a regular basis [20]. However, systems storing multiple full snapshots[1] may waste huge memory space to accommodate redundant data. Modern graph processing frameworks, like LLAMA [21], use *delta snapshot [22]* to save memory space by storing only the updated nodes or edges in each snapshot. In other words, each delta snapshot only contains graph updates (e.g., insertion, modification, and deletion) that occurred after the previous delta snapshot, so all snapshots must be read to traverse the entire graph. With support for delta snapshots, evolving graph processing systems can not only provide version control but also efficiently analyze graphs in the time domain. In the rest of this article, "delta snapshot" and "snapshot" are used interchangeably.

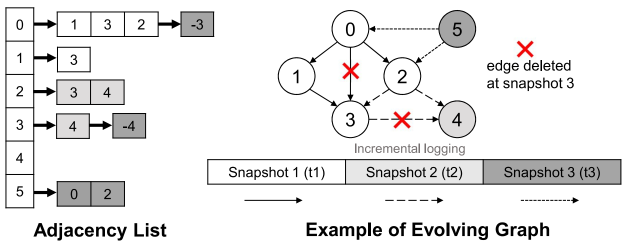


Fig. 1. Data structure for evolving graphs.

An evolving graph is typically stored in the format of an adjacency list [23], which is also applied to static graphs. An adjacency list maintains a linked list for each vertex to chain all correlated neighbors, and all updates from the same snapshot are grouped in an array [20]. Its structures enables efficient traversal all neighbors of any vertex in the adjacency list. Fig. 1 shows an evolving graph and its adjacency list format. The graph evolves to its third snapshot. The first snapshot includes four insertions (i.e., (1, 3), (0, 3), (0, 1), and (0, 2), with (1, 3) representing the newly inserted edge connecting vertex 1 and 3. The second snapshot contains three insertions, while the third snapshot has two insertions and two deletions. It is worth noting that, in the evolving graph framework, deleting an edge is typically translated into out-place updates. Rather than removing the deleted edge directly, we create a new edge with a negative sign. Out-place update not only lowers the cost of fine-grained memory modification, but it also makes it simple to go back to a previous version of the evolving graph.

2) Storage Class Memory: As the graph continues to evolve over time, the sheer volume of data within the graph structure increases proportionally, leading to more frequent and intensive data movements within systems. This growth in data size poses significant challenges for memory management and access efficiency [24]. Fortunately, recent advancements in manufacturing technologies, such as 3-D X-point [25], [26], [27], [28] and ultralow-latency NAND Flash [29], [30], [31], [32]), have paved the way for the emergence of SCM [33], [34]. These innovative memory solutions offer a hybrid approach, combining the speed and byte-addressable access of DRAM with the nonvolatility and higher density of traditional storage devices. Several products and prototypes have emerged to capitalize on these advancements, including Intel® Optane™ Persistent Memory [35] and HPE NVDIMM [6], [36].

SCM represents a new category of memory devices that combine the desirable characteristics of both DRAM and traditional storage devices. These memory devices offer

[1]A full snapshot is a snapshot, which contains the entire graph layout in the moment of taking snapshot.

byte-addressable access granularity with 64-B cacheline accesses, ensuring efficient data retrieval and manipulation. Additionally, SCMs feature nonvolatility, allowing data to be retained even when power is removed, akin to storage devices. Moreover, SCMs boast lower-unit costs (price/GB) compared to DRAM and higher-storage density, providing up to 512-GB per DIMM, making them an attractive solution for memory-intensive applications [37].

Moreover, the integration of SCM into computing systems has been further facilitated by its diverse connectivity options. In addition to occupying traditional DIMM channels, SCM can also be connected via PCIe channels, leveraging the compute express link (CXL) interconnection[2] [38], [39], [40], [41]. This flexibility in connectivity enables SCM to be seamlessly integrated into existing architectures, offering greater scalability and adaptability to evolving memory requirements. With SCM's ability to bridge the gap between DRAM and storage, computing systems can achieve enhanced performance and efficiency in handling the growing demands of evolving graph structures and other data-intensive workloads.

However, due to their slower performance and shorter lifespan relative to DRAM, SCMs are typically utilized as extensions of DRAM rather than as direct replacements [42], [43]. In this hierarchical memory architecture, frequently accessed data (such as inner nodes in tree-data structure) resides in DRAM to leverage its faster access times and lower latency [42]. Conversely, less frequently accessed or large-scale data (such as leaf nodes in tree-data structure) that exceeds DRAM capacity is stored in SCMs, allowing for efficient use of available memory resources. One notable advantage of SCMs is their direct accessibility by CPUs, enabling seamless data transfer from SCMs directly into the CPU cache in cacheline-sized chunks. This direct access capability, discussed extensively in prior research [4], [44], allows for efficient utilization of SCMs alongside DRAM, mitigating the performance impact of slower SCM access times by leveraging CPU cache mechanisms. As a result, SCM adoption offers promising opportunities for improving memory performance and scalability in modern computing systems.

B. Observation

1) Delta Snapshots Break Graph Spatial Locality: Delta snapshots can generate multiple data versions for the same graph, significantly increasing memory usage and necessitating larger memory devices. Another major problem with delta snapshots is a loss of spatial locality. As more snapshots are generated, neighboring vertices are scattered across multiple memory pages, significantly degrading graph traversal performance. In many graph processing algorithms, when a vertex is accessed, all of its 1-hop neighbor vertices are also accessed. Because these neighbors are updated at different times, they are stored on separate memory pages. Many graph processing frameworks write snapshots to pages in chronological order (i.e., by creation time). As a result, vertices physically stored together may be logically distant from one another. Consequently, compared to ideal placement, more pages have to be read in a graph traversal to access each vertex's neighbors, resulting in extra page table walks and TLB accesses.

[2]CXL is a high-speed interconnect technology that facilitates efficient communication between CPUs and accelerators, including memory devices, to enable heterogeneous computing architectures.

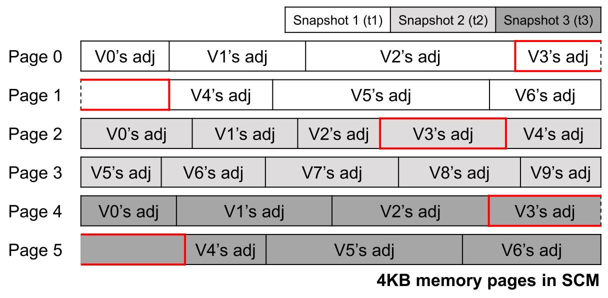


Fig. 2. Storing evolving graph on memory.

Fig. 2 shows an example to illustrate the impact of graph spatial locality. Each rectangle represents a 4-kB page, where different gray levels indicate different snapshots. For example, the darkest part implies all graph updates belonging to the third snapshot. Besides, adj stands for an adjacency list, where V0's adj means the adjacency list of V0. Assuming that neighbor vertices belonging to V3 evolves during different time period (e.g., $t1$, $t2$, and $t3$), those updated neighbor vertices are scattered across three snapshots. In this case, it requires 5 page accesses (marked by red lines) to explore V3's neighbors, where each page access might cause 1 TLB access and at most 4 memory accesses for walking page tables. Even worse, the performance degradation becomes more serious where systems shall explore most of the vertices for traversing a graph, instead of exploring only one vertex. Although some frameworks, such as LLAMA, can alleviate the performance impact by periodically merging multiple snapshots, the performance of graph traversing becomes unstable and fluctuates seriously. The reason is that, the graph traversing reaches best performance right after running snapshots merging, but it becomes worse until triggering the next merging.

Fig. 2 demonstrates the impact of graph spatial locality. Each rectangle represents a 4-kB page, and the different gray levels indicate different snapshots. For example, the darkest part denotes all graph updates from the third snapshot. Furthermore, "adj" stands for an adjacency list, and "V0's adj" denotes V0's adjacency list. Assuming that neighbor vertices in V3 evolve over different time periods (e.g., $t1$, $t2$, and $t3$), the updated neighbor vertices are distributed across three snapshots. In this case, it takes 5 page accesses (marked by red lines) to traverse V3's neighbors, with each page access potentially resulting in 1 TLB access and up to 4 memory accesses for walking page tables. Even worse, performance degradation worsens when systems must traverse all of the vertices in order to traverse a graph, rather than just one. Although some frameworks, such as LLAMA, can mitigate the performance impact by periodically merging multiple snapshots, graph traversal performance still fluctuates significantly: traversal performs best immediately after a snapshot merge, but gradually degrades thereafter.

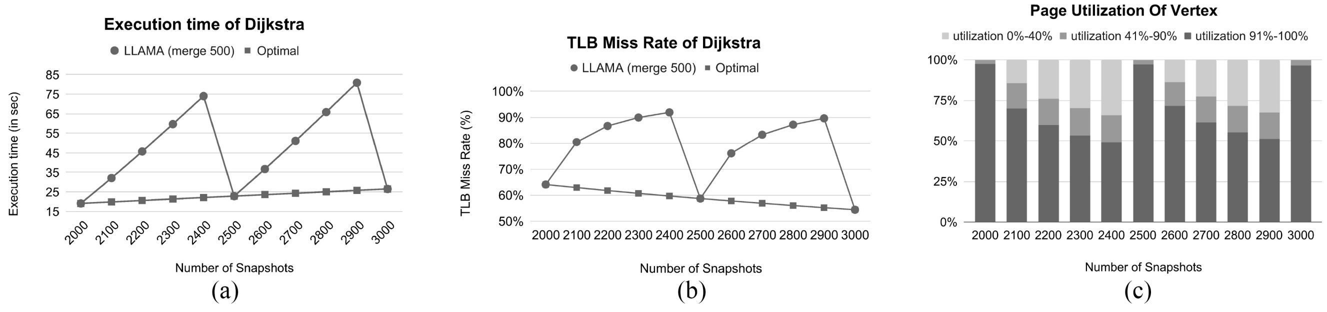


Fig. 3. Weak graph spatial locality hurts performance (Dataset: Friendster). (a) Execution time. (b) TLB miss rate. (c) Page utilization.

2) Performance Impact Under Weak Graph Spatial Locality: We conducted a series of experiments to validate our findings. Fig. 3 shows the performance results. We use LLAMA [21] as an example, which is a representative evolving graph framework. Without loss of generality, the LLAMA merge frequency is set to every 500 snapshots. To simulate graph evolution, we divide a large graph, Friendster,[3] into 10 000 snapshots, and the graph will eventually evolve (or update) 10 000 times. We evaluated graph traversal performance by running the Dijkstra algorithm every 100 snapshots using two approaches. The first approach is LLAMA, which stores snapshots in SCMs. The second approach is called "optimal." It merges all snapshots in DRAM and immediately rewrite a new graph to SCMs. This approach has the strongest spatial locality but can suffer from high-update overhead.

Fig. 3(a) and (b) show evaluation results for overall execution time and TLB miss rate, respectively. The x-axis in both figures represents the number of archived snapshots. Fig. 3(a) shows that the placement issue may significantly affect the execution time, with the system adopting LLAMA spending 5 times more execution time than the system running the optimal approach. Running LLAMA breaks graph spatial locality, causing the CPU to read extra pages, resulting in high-TLB misses and frequent page table walks, as shown in Fig. 3(b). Furthermore, it is obvious that the performance of running graph traversal is unstable when using LLAMA. This unstable performance will degrade the user experience. Even worse, frequently merging snapshots may result in frequent access to SCMs, which consumes additional energy.

The above experiment shows that weak graph spatial locality can reduce page utilization. The page utilization of each vertex is defined as the ideal memory size occupied by the vertex's neighbors divided by the memory size occupied by the vertex's neighbors. For example, the total size of V1's adjacency list (i.e., all of V1's neighbors) is less than the size of one page, requiring only one memory page to store it. In reality, V1's page utilization is less than 10% because its neighbors are scattered across 10 memory pages.

Fig. 3(c) shows page utilization for a system with varying snapshots. The x-axis shows the number of snapshots owned by the system, while the y-axis shows page utilization across all vertices. To better demonstrate the trend, we divide page utilization into three categories: 1) 0%–40%; 2) 41%–90%; and 3) 91%–100%. As the system generates more snapshots, the number of vertices with page utilization between 91% and 100% decreases significantly.

[3]The dataset is from Stanford network analysis project (SNAP) [45].

C. Motivation

This work is strongly motivated by the need to improve the traversing performance for the SCM-based evolving graph systems by keeping strong graph spatial locality for all vertices. We propose a joint management middleware that performs both memory allocations and data placements for evolving data while taking into account graph spatial locality. The major technical challenges are 1) how to maintain strong graph spatial locality while the graph evolves, and 2) how to intelligently place and rewrite data on SCMs without causing excessive energy consumption.

III. SCM-Based Graph-Evolving Aware Data Arranger

A. Overview

This section introduces our SCM-Based GEAR, designed to maintain strong graph spatial locality by consolidating all neighbors of each vertex on the SCM while minimizing energy consumption. Technically, SCM-based GEAR serves as middleware between the graph application and the SCM device, bridging the information gap between them. Implementing GEAR as middleware not only facilitates information exchange but also ensures high compatibility, avoiding the need to modify either the application or the devices. Fig. 4 provides an overview of our design, which comprises four key components: 1) multilevel page allocation; 2) locality-aware data placement; 3) dual-granularity wear leveler; and 4) graph updates accumulation.

Our multilevel page allocation component partitions and allocates SCM memory areas to store all neighbors associated with each vertex. Each vertex's degree (the number of the vertex's edges) determines the size of its allocated SCM memory area. Furthermore, our locality-aware data placement mechanism ensures that all evolved graph data (i.e., newly updated edges) related to the same source vertex are stored in the corresponding SCM memory area, thereby preserving strong graph spatial locality. Additionally, our dual-granularity wear leveler collaborates with our page allocation strategy to evenly distribute graph updates across all memory pages in SCM during graph evolution.

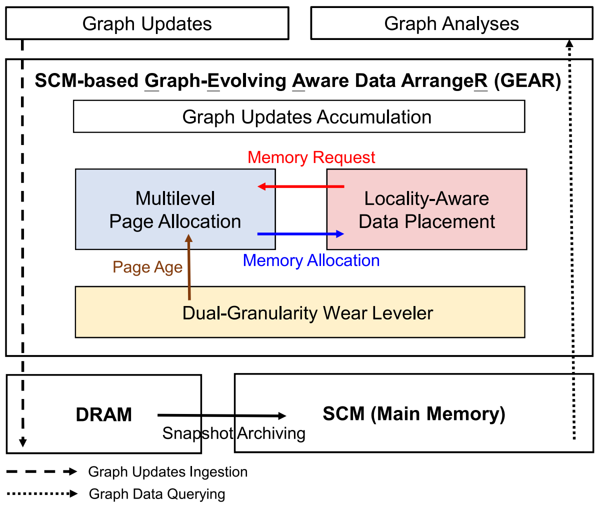

Graph Updates
Graph Analyses
SCM-based Graph-Evolving Aware Data ArrangeR (GEAR)
Graph Updates Accumulation
Memory Request
Multilevel Page Allocation
Locality-Aware Data Placement
Page Age
Memory Allocation
Dual-Granularity Wear Leveler
DRAM
SCM (Main Memory)
Snapshot Archiving
Graph Updates Ingestion
Graph Data Querying

Fig. 4. System architecture.

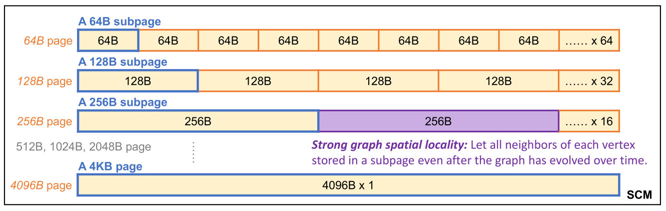


Fig. 5. Example of multilevel page allocation.

Finally, the graph updates accumulation policy buffers incoming graph updates in DRAM. It employs a data structure called an edge log array to facilitate quick querying of these new edges without traversing the entire graph. Because SCM has a higher-write latency than DRAM, our design prioritizes staging new graph updates in DRAM for quick ingestion. The buffered data are subsequently transferred to SCMs in batches, referred to as snapshots. The edge log array maintains incoming graph updates in a first-in-first-out (FIFO) manner, with each update containing three fields: 1) the source vertex; 2) the destination vertex; and 3) the edge weight between them. It is worth noting that such stage-and-flush design is widely used in many graph systems, so we will not go into specific design details.

B. Multilevel Page Allocation

SCM-based GEAR aims to maintain strong graph spatial locality by consolidating all neighbors belonging to each vertex within contiguous memory areas on the SCM. In real-world graphs, hub vertices, which are those with extremely high degrees, have significantly more neighbors than nonhub vertices. Celebrities in social networks are an excellent example of a hub vertice: their graph neighbors can be hundreds, if not thousands, of times more than regular users (nonhub vertices).

Traditionally, most systems allocate memory areas (or pages) of 4 kB. To reduce maintenance costs of graphs evolution, it is common to allocate a 4-kB page for each hub or nonhub vertex. However, this allocation results in low-page utilization. For example, assume that storing one neighbor edge requires approximately 8 bytes (including the index of the neighbor vertex and the edge weight). Then, storing a nonhub vertex with 100 neighbor edges would only require 800 bytes, well below the 4-kB capacity. Even if the combined neighbors of some hub vertices can fill a 4-kB page, the memory requirement might expand over time and no longer fit within the 4-kB memory area as the graph evolves. A simple solution would be to divide a 4-kB page into smaller sizes, but this would require significant maintenance overhead and result in severe space fragmentation.

The multilevel page allocation strategy in SCM-based GEAR relies on two fundamental principles. First, it aims to minimize maintenance overhead by organizing memory areas into sizes aligned with seven predefined levels (each a power of two in size): 64, 128, 256, 512, 1024, 2048, and 4096 B. To provide a clearer understanding of this concept, Fig. 5 visually illustrates the relationship between a 4-kB page and its potential partitioned levels. For instance, a 4-kB page can be partitioned into 64 64-B subpages, with each subpage dedicated to storing neighbors from the same vertex. Second, the allocation process chooses an appropriate memory area size from among the available options based on the vertex's degree. This adaptive approach ensures that memory allocation is tailored to the each vertex's specific characteristics, resulting in improved performance and resource utilization.

GEAR uses the `mmap` system call to obtain multiple 4-kB pages from the operating system (OS). The multilevel page allocation partitions each 4-kB page into identically sized subpages that fall into one of seven predefined levels. The required size for storing all neighbors associated with a vertex is estimated using the vertex's degree, and a subpage that meets or exceeds this requirement is allocated. This design has a low overhead for multilevel page allocation, requiring only a few extra bits per subpage to find a subpage's location within a 4-kB page. Furthermore, it mitigates the fragmentation issue of fixed-size memory areas. Additionally, the alignment of subpage sizes with the CPU cacheline[4] (64 B) ensures that unused data is not transferred from SCMs to the CPU cache, thereby optimizing data transfer efficiency.

Fig. 6(a) and (b) show the data structures used by GEAR to manage the mapping between each 4-kB page and its subpages. The page metadata table [Fig. 6(a)] stores all relevant information about each 4-kB page. The granularity flag indicates the size of the corresponding subpage, represented by a 3-bit binary number (for example, a subpage size of 2048 B is denoted as 110). The empty flag indicates whether or not the subpage has been allocated.

The available page lists [Fig. 6(b)] consist of seven arrays, which include a free page list and six size-specific available page lists. The free page list contains all 4-kB pages that have not yet been divided into subpages. Each size-specific available page list corresponds to one of the six subpage levels (64 to 2048 B), maintaining all associated 4-kB pages with unallocated subpages. This design requires only a 4-byte page index to track each page. For example, given a 1-GB SCM,

[4]A cacheline is the smallest unit of data that can be transferred between main memory and the CPU cache.

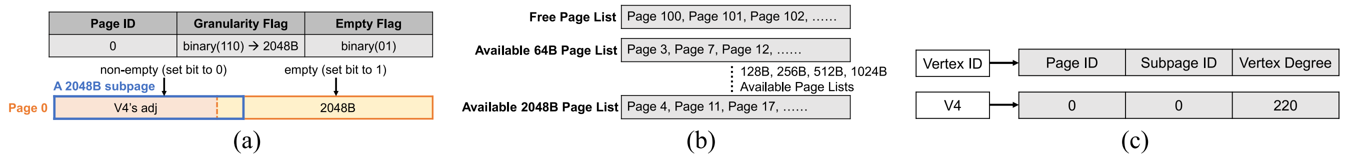


Fig. 6. Data structures for maintaining evolving graph data on SCMs. (a) Page metadata table. (b) Available page list. (c) Vertex-to-page table.

there are 262, 144 4-kB pages, and the overall seven arrays consume 1 MB (i.e., 262 and 144 pages × 4 B).

Lastly, GEAR features a vertex-to-page table [Fig. 6(c)] to track the relationship between each vertex and its associated page information. This includes the 1-byte vertex's subpage index, 4-byte page index, and 2-byte vertex degree. Based on our calculations, the combined space overhead of all three data structures (i.e., vertex-to-page table, available page list, and page metadata table) accounts for less than 5% of the total graph size.

Let us use an example to demonstrate the page allocation process. To assign a 2048-B subpage to a vertex [e.g., V0 in Fig. 6], the process starts by checking the 2048-B available page list. If it is empty, the system chooses a page from the free page list. The metadata table is then updated, with the granularity flag set to 110 for the selected page, indicating its size as 2048 B. The vertex-to-page table is then updated to associate V0 with the allocated page index, with the subpage index set to 0 and the degree recorded as 220. This allows for efficient management and retrieval of graph data during evolution and traversal.

C. Locality-Aware Data Placement

The locality-aware data placement strategy aims to maintain strong graph spatial locality while transferring accumulated graph updates from DRAM to SCMs to generate a snapshot. As part of this strategy, the multilevel page allocation ensures that each vertex's subpage is sufficiently sized[5] to accommodate all its neighbors. Consequently, each subpage typically contains unused space, known as the reserved area. This reserved area serves as a designated space for future graph updates associated with the vertex, ensuring that new updates to different vertices remain segregated, thus preserving strong graph spatial locality across all vertices.

When incorporating a new graph update into a subpage's reserved area, two scenarios may occur: 1) the reserved area of the targeted subpage is either sufficient (i.e., not full) or 2) insufficient (i.e., full). If the reserved area is sufficient, the graph update is written directly to the appropriate reserved area. In contrast, if the reserved area is insufficient, our approach requires rewriting all previous data within the subpage, including the entire adjacency list, to a larger subpage. This ensures that the most recent updates are accommodated while preserving strong graph spatial locality for each vertex. Even for node deletion, the graph system generates a new graph update, as explained in Section II-A1. That is, whenever

[5]The size of the subpage must be greater than or equal to the space currently occupied by all neighbors belonging to the vertex.

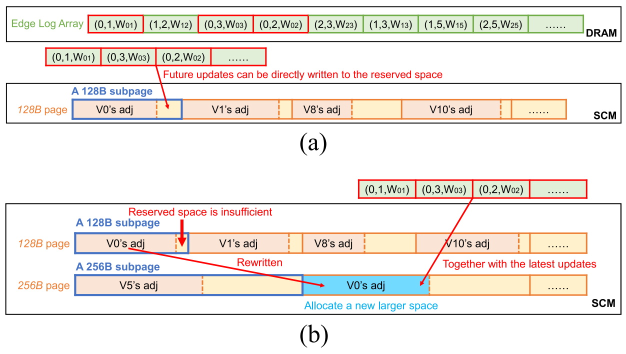


Fig. 7. Two scenarios for the reserved area. (a) Reserved area is not full. (b) Reserved area is full, rewrite data to a larger subpage.

a neighbor is removed from a vertex, a new edge with a negative value is appended to the adjacency list.

For instance, Fig. 7 shows how locality-aware data placement works when writing all graph updates associated with source vertex V0 to the SCM. The notation "(0, 1, $W_{0,1}$)" means the edge value between source vertex V0 and its neighbor vertex V1 is updated to $W_{0,1}$. There are two cases: when the corresponding reserved area in the SCM is insufficient or sufficient. In both cases, all graph updates are buffered in the edge log array in DRAM. In the case where the reserved area is sufficient, as depicted in Fig. 7(a), our policy directs the writing of all graph updates associated with vertex V0 to the corresponding subpage, which belongs to the 128-B level, in the SCM.

On the other hand, in Fig. 7(b), the reserved area of the subpage associated with vertex V0 lacks enough free space to accommodate graph updates associated with vertex V0. Given that the adjacency list of vertex V0 was originally stored in a 128-B subpage, the data placement mechanism collaborates with the multilevel page allocation to obtain an empty 256-B subpage capable of storing both the old adjacency list and all new updates for vertex V0. Subsequently, the old adjacency list of vertex V0, along with its latest updates from DRAM, is transferred and rewritten to the newly allocated 256-B subpage in the SCM.

It is important to point out that our strategy only rewrites subpages with insufficient reserved area, rather than rewriting 4096-B subpages equivalent to a normal page. Consequently, compared to the merging strategy employed by state-of-the-art frameworks, our strategy achieves strong graph spatial locality for each vertex with fewer writes.

D. Dual-Granularity Wear Leveler

Data updates on real-world graphs exhibit a high degree of skew, a phenomenon well-documented in [46], [47], and [48].

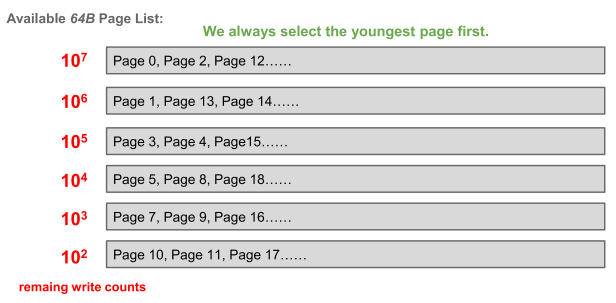

Available 64B Page List:
We always select the youngest page first.
10^7 Page 0, Page 2, Page 12……
10^6 Page 1, Page 13, Page 14……
10^5 Page 3, Page 4, Page15……
10^4 Page 5, Page 8, Page 18……
10^3 Page 7, Page 9, Page 16……
10^2 Page 10, Page 11, Page 17……
remaing write counts

Fig. 8. Interpage wear-leveling mechanism.

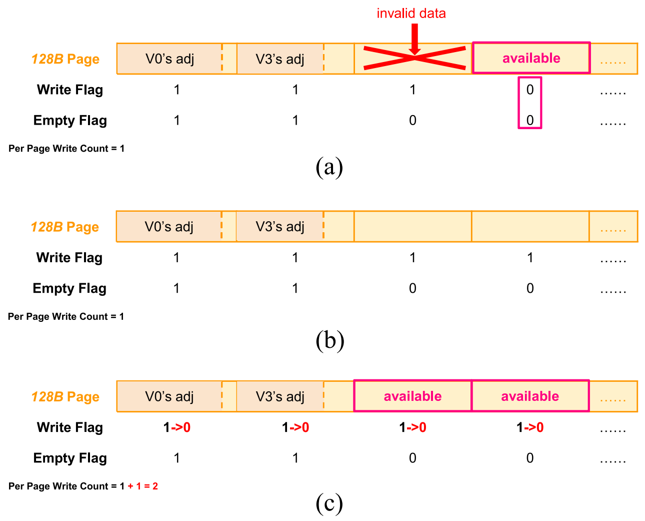


Fig. 9. Intrapage wear-leveling mechanism. (a) Available page: Write flag == 0 and empty flag == 0. (b) No available page. (c) Reset write flag and per page write count + 1.

This skew is primarily attributed to hub vertices that are densely connected to numerous neighboring vertices. Consequently, these hub vertices undergo more frequent updates compared to other vertices. Such skewed updates pose a significant challenge in the context of SCM, which has a limited lifetime. Moreover, our design's manipulation of subpage allocation introduces a further layer of complexity, potentially resulting in disparate write counts among subpages within the same 4-kB memory page. This disparity exacerbates the wear leveling issue, necessitating a comprehensive approach to address wear leveling not only across all 4-kB pages but also within each 4-kB page. To tackle this challenge comprehensively, we propose a dual-granularity wear leveler comprising both an interpage wear-leveling mechanism and an intrapage wear-leveling mechanism.

We design the interpage wear-leveling mechanism to ensure a uniform distribution of write counts across all memory pages. The main idea is to consistently select the healthier page during memory allocation. To facilitate this process, we maintain a per-page write count for each memory page in the page metadata table, as depicted in Fig. 6(a). Technically, we use a multilevel page list, where each page list bounds the minimum remaining write counts for each page within it. The minimum remaining write count associated with the highest level is determined based on the ideal lifetime of the SCM device. To reduce maintenance overhead, we categorize the minimum remaining write count for each level in an exponential manner. For example, if an SCM device can endure at most 10^8 write accesses per cell, our mechanism configures the page list into six levels: 1) 10^7; 2) 10^6; 3) 10^5; 4) 10^4; 5) 10^3; and 6) 10^2, as illustrated in Fig. 8. The number of each level denotes the minimum remaining write count. The remaining write count for each page is calculated as 10^8 minuses the page write count. For example, 10^7 indicates that the page belonging to this level can withstand at least 10^7 more write operations. This meticulous categorization ensures an even distribution of write operations across memory pages, thereby effectively mitigating wear-leveling issues at the interpage level.

As detailed in Section III-C, insufficient space in a subpage designated for storing graph updates the rewriting of all data from the original subpage to a larger subpage. Such movement of vertices between subpages within the same 4-kB page can lead to wear-unleveling issues. To address this concern, we introduce an intrapage wear-leveling mechanism, which is implemented by maintaining a 1-bit write flag and a 1-bit empty flag in the page metadata table for each subpage. The write flag records whether the subpage has been written in the current round, with 1 indicating that it has been written and 0 indicating otherwise. Additionally, the empty flag denotes whether the subpage is currently used by a vertex's adjacency list, with 1 indicating used and 0 indicating availability.

As shown in Fig. 9(a), we only allocate a subpage when both the write flag and empty flag are 0, to ensure balanced write counts across all subpages within the same 4-kB page. When none of the pages in the available page list have available subpages, it indicates most of the subpages in these pages were written during this round. Thus, we reset all the write flags of the available page list to 0 and increment the per-page write count by 1. Subsequently, all subpages become available again, as depicted in Fig. 9(b) and (c). This approach not only maximizes the utilization of available subpages but also ensures the amortization of write counts across all subpages within the same 4-kB page, effectively mitigating wear leveling issues at the intrapage level.

IV. Performance Evaluation

A. Evaluation Setup and Performance Metrics

This section evaluates the efficacy of GEAR in enhancing the performance of both graph traversal and graph evolution. We thoroughly compared SCM-based GEAR to three baseline approaches, checking their performance in a number of areas, such as execution time, TLB miss rate, CPU cache miss rate, energy use, and the number of writes to the SCM. The three baseline approaches consist of two state-of-the-art evolving graph processing frameworks: 1) LLAMA [21], configured with merge frequencies set to 100 and 500 snapshots and 2) GraphOne [23], which incorporates cache-line-sized memory allocation and hub vertex handling. GraphOne's memory allocation strategy provides a cache-line-sized (i.e., 64 bytes) area for storing nonhub vertices, while its hub vertex

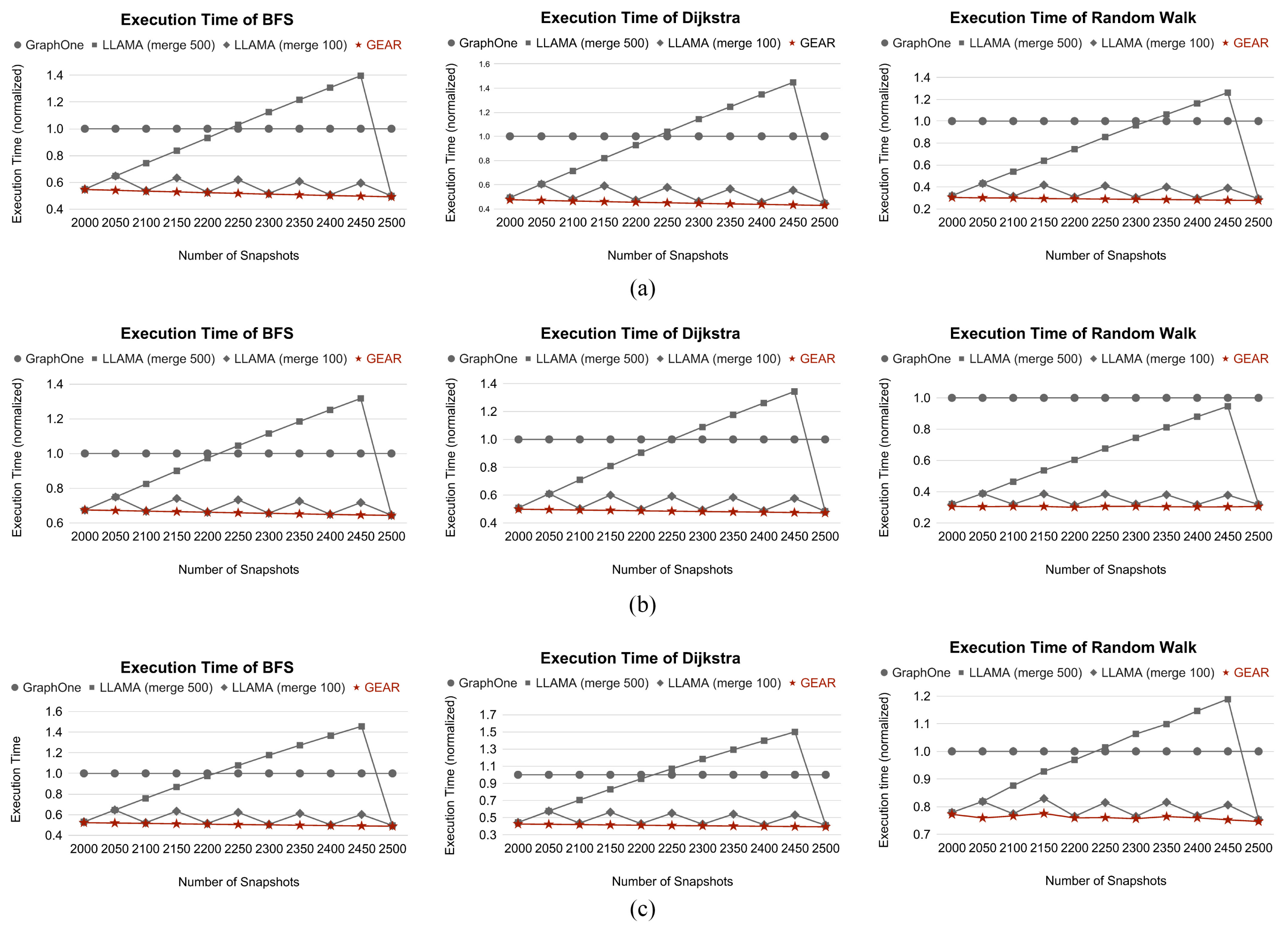


Fig. 10. Normalized execution time of running graph algorithms on evolving graphs. (a) Dataset: Orkut. (b) Dataset: Twitter-2010. (c) Dataset: Friendster.

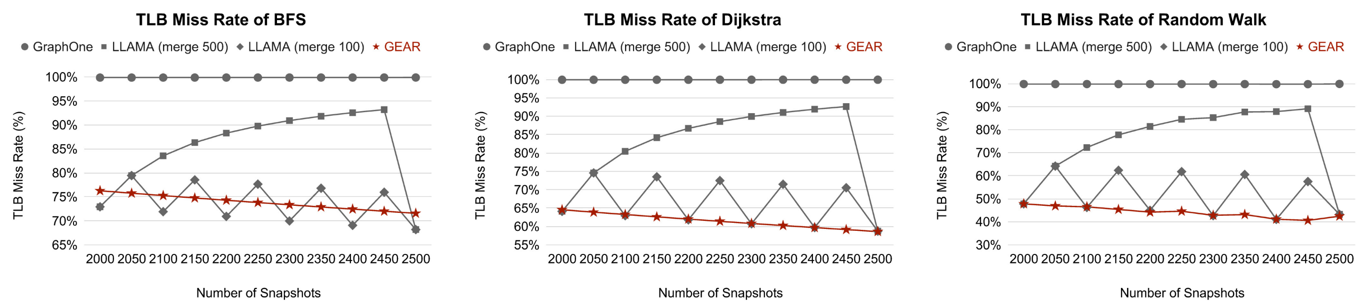
handling allocates a 4-kB page to accommodate all edges belonging to a hub vertex. It is noteworthy that, according to the literature, GraphOne includes delta checkpointing similar to LLAMA, but GraphOne does not enable checkpointing by default. Without loss of generality, all of the three baseline approaches place frequently accessed data in DRAM and less frequently accessed data in SCM.

To ensure a comprehensive evaluation, we selected three representative datasets from the SNAP [45]: 1) Orkut; 2) Twitter-2010; and 3) Friendster. The Orkut dataset encompasses 3 million vertices and 0.23 billion edges. The Twitter 2010 dataset comprises 40 million vertices and 1.5 billion edges. Lastly, the Friendster dataset includes 56 million vertices and 2.6 billion edges. We selected these datasets to offer a wide variety of graph sizes and complexities, enabling a thorough assessment of the performance of SCM-based GEAR.

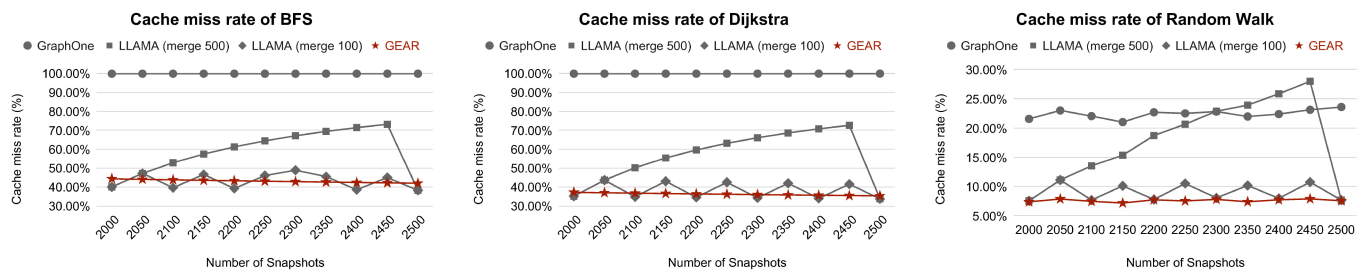
We segment each graph dataset into 10 000 snapshots to simulate the graph evolution process. We execute three widely used graph traversal algorithms—breadth-first search (BFS), Dijkstra (single source shortest path algorithm), and Random Walk algorithms—on the evolving graph at intervals of 100 snapshots. We capture memory traces during the traversal and subsequently replay them on our trace-based simulator. Our simulator simulates an Intel Skylake architecture with a fully associative TLB comprising 1536 entries and an 8 MB, 16-way associative L3 cache [49], [50]. The read/write latency for DRAM and SCMs is set to 50/50 and 120/150 ns, respectively, based on previous studies [6]. Additionally, it accounts for the energy consumption associated with writing a bit to the SCM, estimated at 16.82 pJ per bit [51]. The simulation environment is hosted on a server featuring an Intel Xeon Gold 6252n CPU, 768 GB of DRAM, and running Linux kernel version 5.4. This setup ensures a realistic emulation of the graph traversal algorithms' performance under various evolving graph scenarios, enabling a thorough evaluation of SCM-based GEAR and baseline strategies.

B. Evaluation Results

1) Performance Evaluation of Graph Traversal: In this section, we focus on demonstrating the performance of traversing an evolved graph. Unlike graph evolution, graph traversal does not require additional data writes and therefore does not impact the system's lifetime. Fig. 10(c) presents the results of the total execution time when running SCM-based GEAR against the three baseline approaches. Specifically, Fig. 10(a) and (b) depict the results obtained from executing traversal algorithms on the Twitter-2010 and Friendster datasets, respectively. The *x*-axis of each figure represents the number



(a)



(b)

Fig. 11. TLB and CPU cache miss rate of running graph traversal algorithms (DataSet: Friendster). (a) TLB miss rate. (b) CPU cache miss rate.

of archived snapshots, while the *y*-axis displays the execution time normalized to that of GraphOne. Due to the similarity in performance trends across all 10 000 archived snapshots, we only present results for the interval between 2000 and 2500 archived snapshots.

The results reveal that SCM-based GEAR achieves execution time savings ranging from 23% to 70% compared to GraphOne, 0% to 74% compared to LLAMA (merge 500), and 0% to 27% compared to LLAMA (merge 100). These savings are attributed to SCM-based GEAR's ability to maintain strong graph spatial locality, leading to fewer TLB misses when accessing pages during graph traversal. Notably, SCM-based GEAR achieves an execution time reduction comparable to LLAMA, especially when the graph algorithm executes immediately after LLAMA triggers snapshot merging. However, LLAMA's performance may exhibit instability, and frequent snapshot merging, such as every 100 snapshots, can lead to excessive energy consumption (further details are provided in Section IV-B2).

To provide a detailed breakdown evaluation, Fig. 11 showcases the TLB miss rate and CPU cache miss rate results obtained when running SCM-based GEAR against the three baseline approaches on the Friendster dataset. In each figure, the *x*-axis represents the number of archived snapshots, while the *y*-axis depicts the TLB miss rate in Fig. 11(a) and the CPU cache miss rate in Fig. 11(b). It is evident from the figures that SCM-based GEAR consistently maintains a relatively low-TLB miss rate and CPU cache miss rate across all numbers of snapshots. Conversely, the TLB miss rate and CPU cache miss rate observed in systems running GraphOne and LLAMA exhibit fluctuations, occasionally exceeding 90% (except the CPU cache miss rate caused by running a random walk), depending on the number of snapshots created. GraphOne experiences exceptionally high-TLB miss rates and CPU cache miss rates due to the absence of a snapshot merging strategy to preserve graph spatial locality during graph evolution. In contrast, LLAMA (merge 500) achieves relatively low-TLB miss rates and CPU cache miss rates every 500 snapshots when the snapshot merging strategy is executed, but the rate steadily increases to around 90%. Employing LLAMA with frequent snapshot merging, such as LLAMA (merge 100), can mitigate the occurrence of excessively high-TLB miss rates. However, the frequent merging strategy significantly prolongs the graph evolution process and, even worse, adversely affects the SCM's lifespan. More detailed evaluations of evolving time and memory endurance will be presented in the subsequent subsections.

2) Performance and Lifetime Evaluation of Graph Evolution: Fig. 12 presents a comprehensive evaluation of both the performance and lifetime aspects of graph evolution. In Fig. 12(a), the time taken to evolve the graph to a specific number of snapshots using different approaches is depicted. The *x*-axis ranges from 1000 to 6000, representing the number of snapshots, while the *y*-axis indicates the time for graph evolution normalized to GraphOne. The evaluation shows that systems running SCM-based GEAR exhibit superior evolving performance compared to LLAMA due to GEAR's lower-time complexity. Conversely, LLAMA incurs greater time consumption due to the periodic merging of snapshots, necessitating the rewriting of all snapshots. For example, LLAMA (merge 100) causes a longer graph evolution time than LLAMA (merge 500) because snapshot merging is triggered more frequently. Furthermore, SCM's high-write latency contributes to the extended time required for graph evolution.

For a more detailed analysis of the graph evolution performance, Fig. 12(b) illustrates the total edge write counts to SCM for each system at intervals of 50 snapshots between 2000 and 2500 snapshots. The *x*-axis represents the number of archived snapshots, while the *y*-axis indicates the total edge

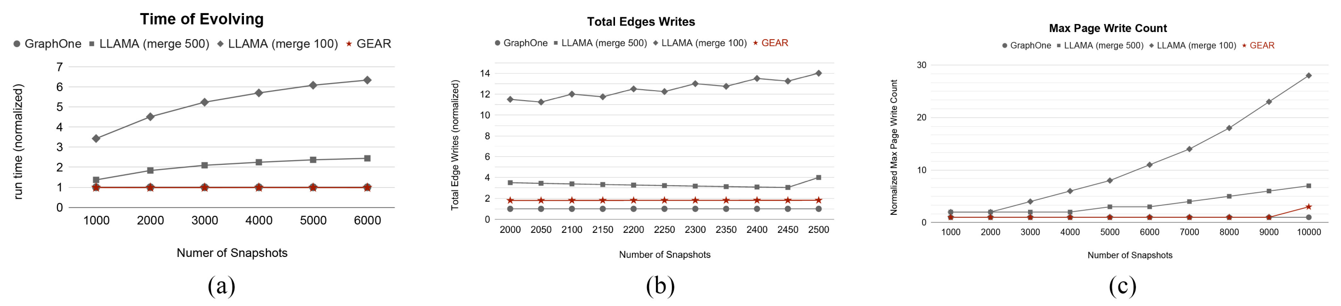


Fig. 12. Performance and lifetime evaluation on evolving a graph. (a) Time spent evolving the graph (DataSet: twitter). (b) Total number of edge writes. (c) Maximum write counts among all pages.

writes normalized to GraphOne. GraphOne, which does not incorporate snapshot merging in our experiments, does not generate extra writes. In contrast, SCM-based GEAR may produce more edge writes than GraphOne if the reserved space of a vertex is insufficient, leading to the rewriting of the original adjacency list to a newly allocated larger subpage along with the latest updates. However, because the SCM-based GEAR only rewrites vertices with inadequate reserved space and limits the maximum rewriting size to 2048 bytes, its total edge writes are only about 2.1 times higher than GraphOne's. Importantly, GEAR's total edge writes are significantly lower than those of LLAMA, which merges snapshots on a regular basis.

While LLAMA may achieve faster graph traversal by merging snapshots more frequently, this has a significant impact on SCM's lifespan. Fig. 12(c) illustrates the normalized maximum page write count as the system generates snapshots ranging from 1000 to 10 000. The x-axis denotes the number of snapshots, while the y-axis represents the maximum page write count normalized to GraphOne. The results demonstrate that our dual-granularity wear leveler is effective in mitigating the increase in maximum page write count. This effectiveness stems from the distribution of writes to a finer granularity, specifically at the subpage level. Conversely, when LLAMA frequently merges snapshots, the maximum page write count experiences a sharp escalation, as observed in the case of `LLAMA (merge 100)`.

3) Evaluation on Energy Consumption: We further evaluate the energy consumption associated with different graph evolution approaches, as depicted in Fig. 13. The x-axis represents the number of snapshots ranging from 1000 to 10 000, while the y-axis indicates the energy consumption. Each plot in the figure illustrates the cumulative energy consumption required to execute the total number of snapshots indicated on the x-axis. Fig. 13 highlights that SCM-based GEAR exhibits relatively low-energy consumption compared to LLAMA. This is primarily because the rewrite operations triggered by SCM-based GEAR result in fewer write accesses on SCMs compared to the merging operations triggered by LLAMA. Notably, GraphOne, which does not perform any snapshot merging operations, consumes the least energy among all solutions. Although SCM-based GEAR consumes more energy than GraphOne, its scalability remains intact. This is evidenced by the consistent energy consumption gap between GraphOne and SCM-based GEAR, even as the graph evolves over time.

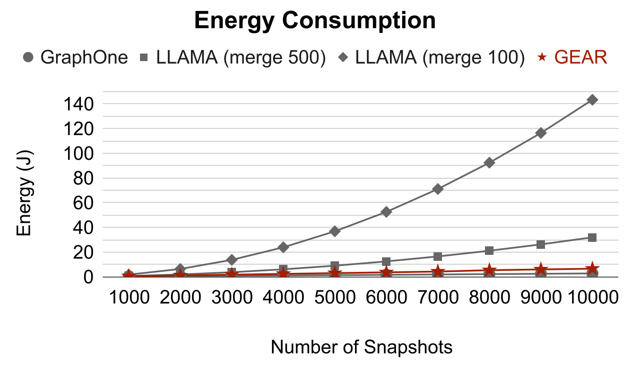


Fig. 13. Energy consumption (Dataset: Friendster).

In contrast, LLAMA's energy consumption exhibits a linear increase as the graph evolves, making it nonscalable. For instance, LLAMA (merge 100) consumes 3 and 21 times more energy than SCM-based GEAR when there are 1000 snapshots and 10 000 snapshots, respectively.

V. Conclusion

Our research addresses the challenge of weak graph spatial locality in evolving graph frameworks, which hinders efficient execution of graph traversal algorithms. To mitigate this issue, we introduce SCM-Based GEAR, a joint management middleware that optimizes the arrangement and storage of evolving graph data in both DRAM and SCMs. GEAR comprises multilevel page allocation, locality-aware data placement, and dual-granularity wear leveling components. GEAR improves graph traversal performance while maintaining strong graph spatial locality as the graph changes. It does this by allocating subpages based on vertex-neighboring relationships, keeping unused areas for future updates, and evenly spreading write operations. Our evaluation demonstrates the effectiveness of SCM-based GEAR, showing significant improvements in execution time savings ranging from 23% to 70% compared to state-of-the-art frameworks. Through meticulous management of evolving graph data across memory devices, GEAR achieves superior performance in traversing evolving graphs, addressing critical challenges posed by weak graph spatial locality.

References

[1] C.-F. Wu, C.-J. Wu, G.-Y. Wei, and D. Brooks, "A joint management middleware to improve training performance of deep recommendation systems with SSDs," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, 2022, pp. 157–162.

[2] W. Fan et al., "Graph neural networks for social recommendation," in *Proc. World Wide Web Conf.*, 2019, pp. 417–426.

[3] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Trans. Knowl. Discov. Data*, vol. 1, no. 1, p. 2, 2007.

[4] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A hybrid PRAM and DRAM main memory system," in *Proc. 46th Annu. Design Autom. Conf.*, 2009, pp. 664–469.

[5] Y.-C. Chen, C.-F. Wu, Y.-H. Chang, and T.-W. Kuo, "Exploring synchronous page fault handling," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 3791–3802, Nov. 2022.

[6] C.-F. Wu, Y.-H. Chang, M.-C. Yang, and T.-W. Kuo, "Joint management of CPU and NVDIMM for breaking down the great memory wall," *IEEE Trans. Comput.*, vol. 69, no. 5, pp. 722–733, May 2020.

[7] C. J. Xue, G. Sun, Y. Zhang, J. J. Yang, Y. Chen, and H. Li, "Emerging non-volatile memories: Opportunities and challenges," in *Proc. 7th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, 2011, pp. 325–334.

[8] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale, "DI-MMAP—A scalable memory-map runtime for out-of-core data-intensive applications," *Clust. Comput.*, vol. 18, pp. 15–28, Mar. 2015.

[9] J. Zhang et al., "Revamping storage class memory with hardware automated memory-over-storage solution," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2021, pp. 762–775.

[10] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, nos. 1–7, pp. 107–117, 1998.

[11] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at Facebook-scale," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[12] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland: A novel abstraction-based out-of-core graph processing system," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 608–621, 2018.

[13] 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 *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2017, pp. 125–137.

[14] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2012, pp. 31–46.

[15] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 45–58.

[16] H. Liu and H. H. Huang, "Graphene: Fine-grainedIO management for graph computing," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 285–300.

[17] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, "GraphSSD: Graph semantics aware SSD," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 116–128.

[18] R. Kumar, J. Novak, and A. Tomkins, "Structure and evolution of online social networks," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, 2006, pp. 611–617.

[19] T. Yang, Y. Chi, S. Zhu, Y. Gong, and R. Jin, "Detecting communities and their evolutions in dynamic social networks—A Bayesian approach," *Mach. Learn.*, vol. 82, pp. 157–189, Feb. 2011.

[20] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Proc. 4th Int. Workshop Graph Data Manag. Exp. Syst.*, 2016, pp. 1–6.

[21] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "LLAMA: Efficient graph analytics using large multiversioned arrays," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 363–374.

[22] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, 2013, pp. 997–1008.

[23] P. Kumar and H. H. Huang, "GraphOne: A data store for real-time analytics on evolving graphs," *ACM Trans. Storage (TOS)*, vol. 15, no. 4, pp. 1–40, 2020.

[24] Y. Jin, C.-F. Wu, D. Brooks, and G.-Y. Wei, "S^3: Increasing GPU utilization during generative inference for higher throughput," in *Proc. Adv. Neural Inf. Process. Syst.*, 2023, pp. 18015–18027.

[25] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3D XPoint technology," *Proc. IEEE*, vol. 105, no. 9, pp. 1822–1833, Sep. 2017.

[26] O. Krestinskaya, A. P. James, and L. O. Chua, "Neuromemristive circuits for edge computing: A review," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 31, no. 1, pp. 4–23, Jan. 2020.

[27] K. Wu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Towards an unwritten contract of Intel Optane SSD," in *Proc. 11th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2019, pp. 1–8.

[28] C.-F. Wu, M.-C. Yang, Y.-H. Chang, and T.-W. Kuo, "Hot-spot suppression for resource-constrained image recognition devices with nonvolatile memory," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2567–2577, Nov. 2018.

[29] W. Cheong et al., "A flash memory controller for 15μs ultra-low-latency SSD using high-speed 3D NAND flash with 3μs read time," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2018, pp. 338–340.

[30] J. Zhang et al., "FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI 18)*, 2018, pp. 477–492.

[31] C.-F. Wu, Y.-H. Chang, M.-C. Yang, and T.-W. Kuo, "When storage response time catches up with overall context switch overhead, what is next?" *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 4266–4277, Nov. 2020.

[32] C.-F. Wu, Y.-H. Chang, M.-C. Yang, and T.-W. Kuo, "How to steal CPU idle time when synchronous I/O mode becomes promising," in *Proc. 61th ACM/IEEE Design Autom. Conf. (DAC)*, 2024.

[33] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM J. Res. Develop.*, vol. 52, no. 4.5, pp. 439–447, Jul. 2008.

[34] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM J. Res. Develop.*, vol. 52, no. 4.5, pp. 449–464, Jul. 2008.

[35] (Intel, Santa Clara, CA, USA). *Product: Intel Optane Persistent Memory (DCPMM)*. (2020). [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html

[36] A. Sainio et al. "NVDIMM: Changes are here so what's next: Memory computer summit." 2016. [Online]. Available: https://www.snia.org/sites/default/files/SSSI/NVDIMM

[37] M. Saxena and M. M. Swift, "FlashVM: Virtual memory management on flash," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2010, pp. 1–14.

[38] S. Van Doren, "HOTI 2019: Compute express link," in *Proc. IEEE Symp. High-Perform. Interconnects (HOTI)*, 2019, p. 18.

[39] D. D. Sharma, "Compute express link (CXL): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy," *IEEE Micro*, vol. 43, no. 2, pp. 99–109, Mar./Apr. 2023.

[40] M. Jung, "Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD)," in *Proc. 14th ACM Workshop Hot Topics Storage File Syst.*, 2022, pp. 45–51.

[41] H. Li et al., "Pond: CXL-based memory pooling systems for cloud platforms," in *Proc. 28th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2023, pp. 574–587.

[42] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," in *Proc. Int. Conf. Manag. Data*, 2016, pp. 371–386.

[43] H.-Y. Cheng et al., "Future computing platform design: A cross-layer design approach," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2021, pp. 312–317.

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

[45] J. Leskovec et al., "Stanford network analysis project." 2010. [Online]. Available: https://snap.stanford.edu/

[46] T.-S. Lo, C.-F. Wu, Y.-H. Chang, T.-W. Kuo, and W.-C. Wang, "Space-efficient graph data placement to save energy of ReRAM crossbar," in *Proc. IEEE/ACM Int. Symp. Low Power Electron. Design (ISLPED)*, 2021, pp. 1–6.

[47] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Trans. Parallel Comput.*, vol. 5, no. 3, pp. 1–39, 2019.

[48] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2012, pp. 17–30.

[49] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *Proc. IEEE Symp. Security Privacy (SP)*, 2019, pp. 39–54.

[50] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 469–480, 2017.

[51] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 2–13.