Zou M, Zhang MZ, Wang RJ *et al*. Skyway: Accelerate graph applications with a dual-path architecture and fine-grained data management. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 39(4): 871-894 July 2024. DOI: 10.1007/s11390-023-2939-x

Skyway: Accelerate Graph Applications with a Dual-Path Architecture and Fine-Grained Data Management

Mo Zou[1, 2] (邹　沫), *Student Member, CCF*, Ming-Zhe Zhang[3] (张明喆), *Member, CCF*
Ru-Jia Wang[4] (王茹嘉), *Member, IEEE*, Xian-He Sun[4] (孙贤和), *Fellow, IEEE*
Xiao-Chun Ye[1] (叶笑春), *Member, CCF*, Dong-Rui Fan[1] (范东睿), *Senior Member, IEEE*
and Zhi-Min Tang[1, 2] (唐志敏), *Member, CCF*

[1] *Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*
[2] *University of Chinese Academy of Sciences, Beijing 100049, China*
[3] *Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100045, China*
[4] *Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, U.S.A.*

E-mail: zoumo@ict.ac.cn; zhangmingzhe@iie.ac.cn; rwang67@iit.edu; sun@iit.edu; yexiaochun@ict.ac.cn; fandr@ict.ac.cn tang@ict.ac.cn

Received October 28, 2022; accepted October 14, 2023.

Abstract Graph processing is a vital component of many AI and big data applications. However, due to its poor locality and complex data access patterns, graph processing is also a known performance killer of AI and big data applications. In this work, we propose to enhance graph processing applications by leveraging fine-grained memory access patterns with a dual-path architecture on top of existing software-based graph optimizations. We first identify that memory accesses to the offset, edge, and state array have distinct locality and impact on performance. We then introduce the Skyway architecture, which consists of two primary components: 1) a dedicated direct data path between the core and memory to transfer state array elements efficiently, and 2) a data-type aware fine-grained memory-side row buffer hardware for both the newly designed direct data path and the regular memory hierarchy data path. The proposed Skyway architecture is able to improve the overall performance by reducing the memory access interference and improving data access efficiency with a minimal overhead. We evaluate Skyway on a set of diverse algorithms using large real-world graphs. On a simulated four-core system, Skyway improves the performance by 23% on average over the best-performing graph-specialized hardware optimizations.

Keywords graph application, computer architecture, memory hierarchy

1 Introduction

Graph processing is a critical component of many application domains, such as social network analysis[1–3], computational biology[4, 5], and machine learning[6]. However, graph processing is also known for its irregular memory access patterns and poor locality, especially for large graphs which contain millions of vertices and edges. Current on-chip caches can hardly store graphs at such a scale. In addition, graph processing cannot fully benefit from existing memory hierarchies due to poor locality. On the cache side, the random vertex traversal, which usually reads a sparsely distributed vertex (4 bytes or 8 bytes) less than every 10 instructions[7], causes excessive cache misses. On the main memory side, the random accesses also lead to frequent row buffer conflicts.

State-of-the-art acceleration techniques for graph processing mainly focus on improving temporal and spatial locality. For example, the software-based ap-

Regular Paper

This work was supported in part by the U.S. National Science Foundation under Grant Nos. CCF-2008907 and CCF-2029014, the Chinese Academy of Sciences Project for Young Scientists in Basic Research under Grant No. YSBR-029, and the Chinese Academy of Sciences Project for Youth Innovation Promotion Association.

proaches[8–14] relocate and package vertices with higher access probabilities (hot vertices) in successive memory blocks for a given graph, making the data accesses more cache-friendly. Unfortunately, for real-world graphs, the scale of hot vertices often exceeds cache capacity, limiting the benefits of software-based optimizations. Most hardware-based schemes[7, 15–22] focus on improving locality and reducing memory traffic. For example, GRASP[16] and P-OPT[15] evict the cachelines in LLC with a lower reuse possibility. PHI[17] and GraphPulse[18] coalesce multiple state updates if they target the same vertex.

Unlike prior work, we notice that a poor performance often comes from the interference of different data access patterns in graph applications, making them hard to benefit from existing memory hierarchies.

We believe that current memory hierarchies can be further enhanced to accelerate graph processing applications. We have collected extensive experimental data on a simulated multi-core system and analyzed the root causes of performance bottlenecks. Our observations are summarized as follows.

• *Poor Locality in Specific Data Arrays*. Data arrays encoding a graph have different memory access patterns and can interfere with each other in a shared memory hierarchy.

• *Low Data Reuse in the Cacheline*. Data access to the specific array (i.e., state array) is random. A conventional cacheline may not help and could be counterproductive.

• *Under-Utilized Memory System*. Although the graph applications are memory-intensive, we find that the memory bandwidth is far from fully utilized under current memory hierarchies.

Based on the observations above, we propose a novel architectural support, named Skyway, to accelerate graph processing by improving the efficiency of the system datapath. Skyway optimizes both the cache hierarchy and main memory system through integrated designs. At the cache side, we modify the conventional cache hierarchy to include a direct path with a small property buffer (PBuf), which supports fine-grained random memory accesses. At the main memory side, we revisit the memory array and row buffer design to include the duplication row (DRow) to mitigate row buffer conflicts. PBuf and DRow can work together seamlessly to improve the utilization of the overall memory system bandwidth without breaking the data locality. Although Skyway is motivated by graph applications, the key idea behind the design that discovering multiple access patterns and processing them separately based on behaviors can be extended to any applications with distinct access patterns. Overall, Skyway provides an opportunity to reduce interference and data movement according to data locality.

We evaluate the proposed Skyway using detailed micro-architectural simulation and receive consistent great performance improvement across diverse algorithms and datasets. Our experimental results show that Skyway improves the DRAM bandwidth utilization by 2.13x on average and up to 5.87x in the best case. Also, Skyway improves the performance by 29% on average and up to 86% in the best case over the baseline without any optimizations. Compared with the state-of-the-art GRASP[16], Skyway provides an average performance improvement of 23% with a 2.19x higher DRAM bandwidth utilization. Skyway adds a marginal storage overhead of 2.6% to LLC and 0.02% to DRAM.

The paper is organized as follows. Section 2 provides background information on graph representation and memory hierarchy organization. Section 3 characterizes the methodology and the benchmark for our evaluation. Section 4 investigates the reason of poor locality and low memory bandwidth utilization, and then presents several opportunities to solve the problem. Section 5 illustrates the scheme of the newly proposed Skyway. Section 6 analyzes the performance results. Finally, we introduce related work in Section 7 and conclude our work in Section 8.

2 Background

2.1 Graph Data Layout

The Compressed Sparse Row (CSR) format is a widely used storage-efficient technique to represent graph structures[7, 16, 19, 23, 24]. As Fig.1 shows, there are three arrays used to encode a graph: the offset, edge, and state. These arrays are allocated in memory at the beginning of the graph loading phase. Each offset entry is an 8-byte pointer pointing to its first neighbor in the edge array, which represents an edge from the source to the destination. The edge array maintains all edges in the form of vertex IDs, and edge-weighted values are stored here as well for weighted graphs. The state array holds the current state of each vertex, which is usually 4- or 8-byte long. The graph algorithm updates state array iteratively until convergence.

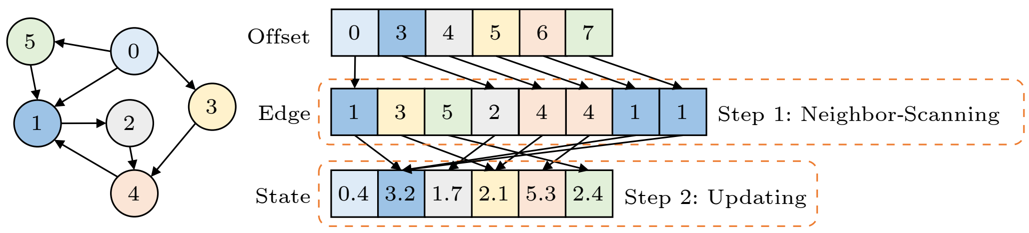


Fig.1. Example graph and its CSR representation encoding push-based approach.

Most graph applications adopt pull- or push-based computation models[24–27] to traverse a graph and update vertex states. In pull-based models, each vertex gathers new states from its incoming neighbors and updates its state upon the accumulated influence. In push-based models, one source vertex broadcasts its new state and modifies all outgoing neighbor states, as shown in Fig.1. In general, there are two steps in graph applications, regardless of pull- or push-based approaches. In step 1, the graph algorithm reads destination IDs in the edge array pointed by one source vertex. In step 2, the graph algorithm updates elements in the state array indexed by destination IDs. In the rest of the paper, we call the first step the neighbor-scanning phase, and the second step the updating phase.

2.2 Memory Hierarchy

The modern memory hierarchy includes the cache and main memory subsystems. Fig.2 shows a typical dual-core system with a three-level cache hierarchy and a connected off-chip DRAM module. The L1 and L2 caches are private, and the LLC is shared. A memory request from the core searches L1, L2, and LLC and fetches data if the request is a hit. Otherwise, the memory request needs to find the data from the DRAM.

Internally, a DRAM module is organized hierarchically into channels, ranks, and banks, where each bank is a 2D array, accessed by the assigned row ID and column ID. Upon receiving a request from the LLC, the memory controller decomposes the physical address into (channel, rank, bank, row, column) and buffers that in the read or write queue. To serve a request, the memory controller sends an activate command to an idle bank first, which loads the desired row into the row buffer. Then a column-level command reads/writes specific data from/to the row buffer. In an open-row policy, the row buffer maintains the last accessed row until further instructed. If the following request visits the open row in the row buffer, the bank does not need to be activated again, which is called a bank hit. On the other hand, if a subsequent request visits a different row within the same bank, the memory controller sends a precharge

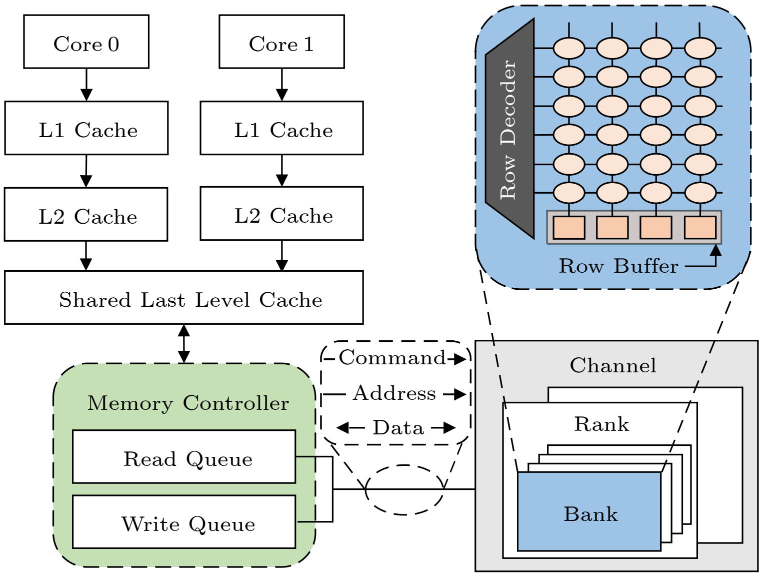


Fig.2. Typical cache hierarchy and main memory in the dual-core system.

command, closing the open row in the row buffer and preparing for the next activate command, which is called a bank miss. Because the memory controller exploits a series of tight timing constraints between command executions, the DRAM access latency in a bank miss is much longer than that in a bank hit.

Both the cache and DRAM row buffer are beneficial when the application exhibits a good locality. However, when data access patterns are irregular and multiple data access patterns co-exist, these locality-based hardware components may not work well to enhance the overall performance.

2.3 Existing Graph Processing Optimizations

Graph Reordering. Graph reordering[8–14] is a commonly used optimization scheme, which relies on preprocessing graph data layout to improve the accessing locality without hardware modification. According to the skewed power-law distribution[9, 10, 13, 28, 29], in real-world graphs, a small portion of vertices occupy most connections (hot vertices), while the rest of vertices own relatively few edges (cold vertices). By relocating the hot vertices consecutively in the memory space, most of the graph applications show better performance since the locality of data accesses is improved. However, the effectiveness of graph reordering is affected by several factors (e.g., the hot vertices identification and the scale of input graphs), which significantly increases the difficulty of performance improvement for various algorithms and graphs[30]. Moreover, the scale of the hot vertices has exceeded the cache capacity for most real-world graphs[14], which limits the efficiency of reordering.

Architectural Optimizations on Graph Locality. GRASP[16] and P-OPT[15] are popular cache managements optimizing irregular data accesses. GRASP reorders the state array based on vertex degrees, which places hot vertices at the beginning of the state array. During the execution phase, GRASP guarantees the cachelines from the hot vertices region to stay longer in cache. P-OPT scans the CSR format to analyze the graph structure and builds a re-reference matrix for dynamic irregular memory accesses. P-OPT evicts cachelines whose next reuse is further in the future. GRASP and P-OPT reduce cache misses to improve the application performance. DepGraph[21] dispatches different dependency chains to different cores, allowing efficient asynchronous vertex state updates on multi-core processors. In this way, DepGraph improves the locality in private cache. PHI[17] coalesces multiple state updates in cache if they target the same vertex and applies the merged state value to the memory controller together. GraphPulse[18] proposes a graph-specialized accelerator to coalesce updates in a FIFO queue. PHI and GraphPulse exploit temporal locality and reduce memory traffic through coalescence.

All of these approaches try to improve the locality in graph applications. In contrast, we observe that the different graph data arrays have distinct locality, and sharing a single datapath for all types of data could bring significant interference and performance degradation. Our proposed Skyway is able to handle complex access patterns more efficiently with a specialized dual datapath design.

3 Experimental Setup

3.1 Profiling Platform

We use Zsim[31], an execution-driven simulator, to measure performance. The simulator has four Out-of-Order (OoO) cores clocked at 4 GHz and an 8 MB shared LLC. To simulate memory behaviors accurately, we extend Zsim with DRAMsim3[32], which simulates a detailed and cycle-accurate memory model supporting DDR4 protocol. Prior work[33] has proved that a larger ROB will not benefit graph application performance, and thus we use 128-entry ROB here. Table 1 lists more configuration parameters. We fast forward the graph loading phase and run 100 million

Table 1. System Configurations

Hardware	Configuration
Core	Four OoO cores, 4 GHz clock frequency, 128-entry ROB, 4-wide issue width, 16 MSHRs per core
L1-I/D cache	Private, 8-way 32 KB per core, 64 B cache line, 4-cycle access latency
L2 cache	Private, 8-way 256 KB per core, 64 B cache line, 12-cycle access latency
LLC	Shared, 32-way 8 MB, 64 B cache line, 32-cycle access latency
Memory controller	64-entry read/write queue, FR-FCFS[34] scheduling policy, Open-Page, address interleaving: rochrababgco
DRAM	Four channels, 2 ranks/channel, 4 bankgroups/rank, 4 banks/bankgroup, 16 Gb DDR4-2400 x8 chips, 8 KB row buffer size[35], tRCD/tRAS/tWR 17/39/18 cycles, peak bandwidth 76.8 GB/s

instructions to warm up cache. Then we mark the region of interest (ROI) in the code covering only pull- and push-based iterations. We collect status in ROI for 600 million instructions across all cores.

3.2 Applications

We use five classic graph applications, Breadth-First Search (BFS)[36], Betweenness Centrality (BC)[37], Connected Component (CC)[38], PageRank (PR), and Single-Source Shortest Path (SSSP)[39], covering both push- and pull-based computation models, from the widely used GAP[25] benchmark for our evaluation. Table 2 gives a detailed description of the five applications. All applications update one state array except the BC application, which requires two state arrays in the execution phase. To avoid two irregular state array accesses in the BC application, we merge the two arrays together and use the optimized implementation as the baseline. For the other four applications, we use the implementation in the GAP benchmark as the baseline.

3.3 Datasets

For our profiling, we use seven real-world graph datasets detailed in Table 3. These graph datasets vary in size and degree distributions but all exceed the LLC capacity. As inputs to the graph applications, all graph datasets are encoded in CSR format and pre-processed by a state-of-the-art reordering technique, DBG①[10], to exploit locality. For graph applications traversing the graph dataset in the push-based model, we reorder the input based on in-degree. For graph applications traversing the graph dataset in the pull-based model, the input is reordered based on out-degree. We combine the seven real-world graph datasets with the five classic applications and produce 35 workloads in all evaluations mentioned in this research.

4 Observations and Design Motivation

4.1 Diverse Data Access Patterns

As introduced in Subsection 2.1, if a graph is stored in the CSR format, the memory access patterns of various data arrays are distinct. For the edge array, the accesses appear to exhibit a high spatial locality. For the state array, the accesses are much more random and suffer from poor locality in the cache hierarchy; the access pattern to the state array is sensitive to the algorithms and graph inputs, significantly increasing the difficulty of performance optimization. The memory requests from the offset array are much fewer than those from the other two arrays and do not incur significant performance overhead.

To quantify cache behaviors in three arrays utilized by CSR format, we analyze the requests classification before and after cache hierarchy and cacheline reuse rates. Since all graph applications exhibit similar cache behaviors on either pull- or push-based model, taking the PR application as an example, we show the detailed statistics on all reordered graphs listed in Table 3. We have the following key observations.

• Cache hierarchy is less effective for the state array accesses. As shown in Fig.3, for most of the datasets, the accesses to the state array are about 31%–40% of the total requests in the cache hierarchy. On the other hand, 73%–88% of the total requests to the DRAM are from the state array, which is much higher than any other data arrays. Two exceptions, Web and UK-2002, show a cluster feature that a small portion of vertices are visited repeatedly within a short time window, leading to a better locality of state array accesses.

• Cachelines from the state array have a low reuse rate. As shown in Fig.4, for most datasets, the cacheline reuse rate of the state array is only around 7%, which is much lower than the offset array (98%) and the edge array (99%). Even for the best scenarios (Web and UK-2002), the reuse rate of the state array

Table 2. Graph Applications

Application	Brief Description	Model
BFS[36]	Traversing a graph from one root vertex until all neighbors are accessed and returning a distance array	Push
BC[37]	Scoring the centrality of every vertex to find the center	Push
CC[38]	Labeling vertices into disjoint subsets to calculate the number of components	Both
PR	Ranking all vertices based on incoming neighbors until convergence or reaching the iteration limitation	Pull
SSSP[39]	Finding the shortest paths from one source vertex to all the other vertices in a weighted graph	Push

①https://github.com/faldupriyank/dbg, Jul. 2024.

Table 3. Scale of the Graph Datasets

Graph Dataset	$\lvert V\rvert$ ($\times 10^6$)	$\lvert E\rvert$ ($\times 10^6$)
Orkut[②]	9	327
DBpedia[40]	18	136
PLD[41]	43	623
Web[42]	51	1 930
MPI[43]	53	1 963
Twitter[2]	62	1 468
UK-2002[44]	134	261

cachelines is just around 34%, which is still low in terms of temporal locality.

Ideally, we want a cache hierarchy to improve the performance of all types of data accesses. However, the data with poor locality can hardly take advantage of the current cache capacity and cacheline size. On the contrary, a multi-layer cache hierarchy may increase data accessing delay for data with poor locality due to the anticipated extensive cache miss.

4.2 Asymmetric Locality in DRAM

Since graph applications suffer from frequent LLC misses, it is essential to have a better understanding of the DRAM behavior under the locality influence of graph applications.

Our evaluations find that the locality variation of various data arrays extends to DRAM as well. Fig.5 shows the row buffer hit rates of the three arrays in the PR application, indicating a significant locality asymmetry in DRAM. According to Fig.5, averaging across all graph datasets, the bank hit rates of the offset, edge, and state requests are 75%, 89%, and 40%, respectively, which illustrates that the irregular state access pattern has an extremely low opportunity of

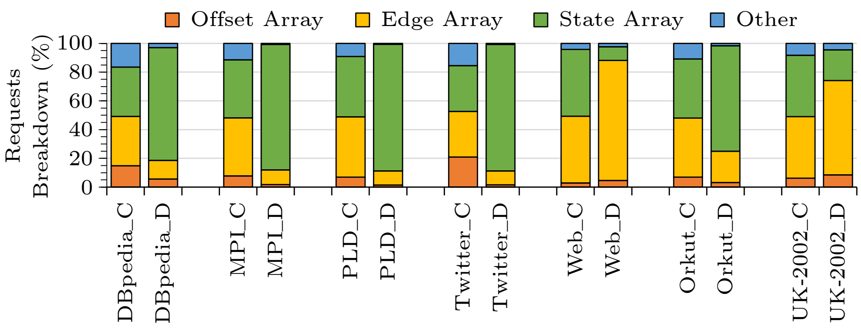


Fig.3. Breakdown of memory accesses to the different data arrays in the cache hierarchy (_C) and the DRAM (_D), taking PR as an example.

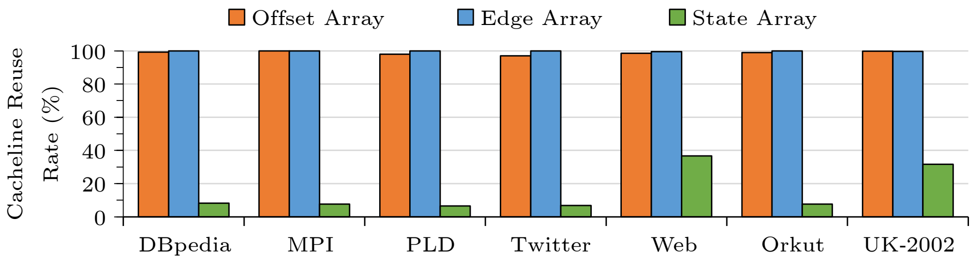


Fig.4. Comparison of the cacheline reuse rates for different arrays, taking PR as an example.

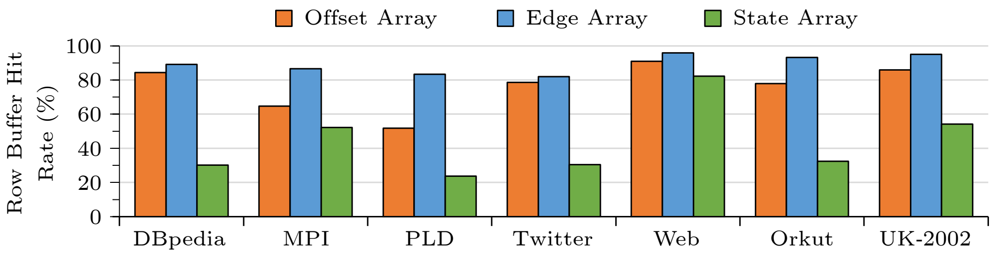


Fig.5. Row buffer hit rates of different arrays, taking PR as an example. Note that each rate is separately calculated as the ratio of the hit number over the total access to the same array.

②http://snap.stanford.edu/data, Jul. 2024.

row buffer reuse compared with the other two arrays. Moreover, the sparsely distributed state addresses will also interfere and affect the locality of offset and edge arrays.

To further understand the performance impact of different access patterns on DRAM, we devise a set of experiments with several ideal assumptions. In each experiment, DRAM ideally serves the requests to one array without changing the content in the row buffer. These experiments help us to understand the interference of data accesses on row buffer pollution.

As shown in Fig.6, the average performance increments are about 3%, 17%, and 32% when eliminating the offset, edge, and state requests to DRAM, respectively. The results indicate that removing state requests improves the performance the most, which is caused by a combination of frequent accesses and a low bank hit rate. For Web and UK-2002, we observe that removing all edge requests gains the highest performance because most of the DRAM requests fall into the edge array in these two graphs, as explained in Subsection 4.1.

Ideally, the row buffer in DRAM is beneficial when consecutive requests access the same row, which avoids the time-costly bank conflicts. However, the complex access patterns interfere with each other in graph applications, leading to a huge performance degradation.

4.3 Low Memory Channel Bandwidth Utilization

Generally, a higher cache miss ratio will lead to a higher memory bandwidth. However, due to the inefficiency of cache and DRAM, we observe that memory channel bandwidth utilization of graph applications is low even with a high LLC MPKI (misses per kilo instructions). The bandwidth utilization is calculated as the ratio of requested bytes over the peak memory bandwidth. Fig.7 reports the profiling results and Table 4 summarizes LLC MPKI on graph workloads. GM is the geometric mean across datasets.

As shown in Fig.7 and Table 4, the average bandwidth utilization in BC (3%) is lower than that in other applications due to a lower MPKI (3). However, even with a high average MPKI (18) across applications BFS, CC, PR, and SSSP, the average DRAM bandwidth utilization is only 18%, indicating that graph applications cannot make good use of DRAM channel bandwidth.

There are two reasons why these applications deliver such a low DRAM bandwidth utilization. First, all memory requests follow a single path along a three-level cache hierarchy regardless of their cache hit rate. The poor-locality data accesses (e.g., state array data accesses) cannot benefit from locality-oriented hardware resources, like ROB and MSHR, block the whole execution pipeline during the cache

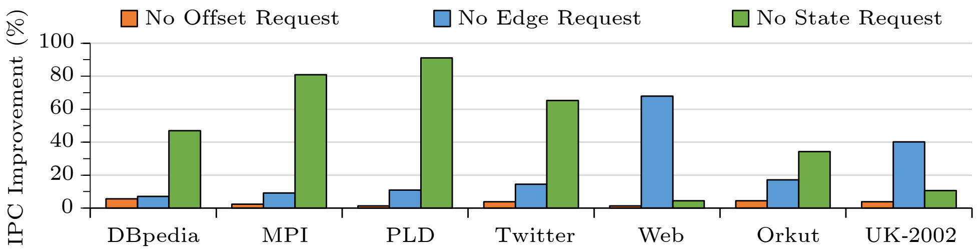


Fig.6. Performance improvements when removing the impact of one type of DRAM requests, taking PR as an example. IPC means instruction per cycle.

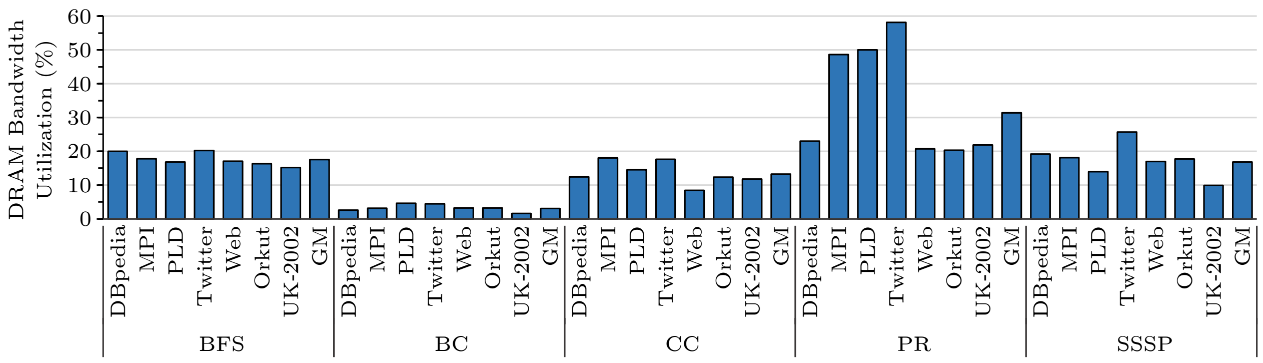


Fig.7. DRAM bandwidth utilization across various workloads.

Table 4. MPKI in Various Workloads Including Seven Real-World Graph Datasets and Five Graph Applications

Graph Dataset	BFS	BC	CC	PR	SSSP
DBpedia	16	3	17	41	21
MPI	20	3	28	60	20
PLD	35	5	19	64	28
Twitter	53	3	16	39	19
Web	8	2	4	14	12
Orkut	15	3	11	27	17
UK-2002	8	1	6	14	10
GM	18	3	12	32	17

search procedure, and still fall into DRAM most of the time. Second, in the DRAM, highly irregular data accesses lead to frequent bank conflicts and interfere with other types of requests, which significantly impacts the overall DRAM performance.

4.4 Design Opportunities

Based on our observations and analysis above, we have identified the following architectural design opportunities to accelerate graph applications.

• Various access patterns in graphs share one single datapath in the current memory hierarchy. Among them, requests to the edge array have a good locality, while requests to the state array have a very poor locality. Processing requests with distinct locality separately will avoid interference in both cache and DRAM.

• On the cache side, the data requests with distinct access patterns can be handled separately for better performance. The conventional multi-layered and 64-byte granularity cache hierarchy is suitable for data accesses with a good locality, such as the edge array requests. For the requests with low locality, we can have a specialized buffer unit to store them on-chip, with a finer data access granularity, for example, for the state array requests, instead of the multi-layered cache hierarchy. In this way, random memory requests do not need to go through a multi-layered cache hierarchy and can access the DRAM directly once missed in the small buffer.

• On the DRAM side, we observe that when focusing on the memory space with smaller granularity than the row buffer, accesses to the hot elements in the reordered state array can still perform moderate or even high temporal locality while their spatial locality is low. Therefore, providing an extra buffer space with fine-grained management support for the data may improve the performance. In addition, an extra buffer space supporting edge array will reduce the interference in the row buffer caused by the irregular accesses. The fine-grained management will allow several hot elements to share one duplication row and reduce time-costly row buffer update operations.

• By integrating our customer-designed direct data access path and the underlying cache hierarchy, we can accelerate graph processing by improving the whole memory efficiency, reducing data access interference, and utilizing the memory channel bandwidth. This integrated design is not limited to graph processing. It can be extended to alleviate the inefficiency of memory systems for a wide range of applications.

5 Skyway Architecture

We propose the Skyway architecture, which is motivated by the following two challenges. First, specific graph structures, i.e., the state array, cannot utilize multi-level cache due to poor locality. Second, frequent row buffer conflicts caused by irregular requests further degrade the performance. To overcome both challenges, Skyway proposes an optional direct datapath from a core to main memory, which enables fast and direct data fetch for requests with low locality. Also, Skyway proposes to add extra buffers in DRAM to support fine-grained duplications, which reduces the row buffer misses and time-costly DRAM updates. Skyway improves the system performance in two ways: 1) providing a highly efficient datapath for poor-locality accesses and 2) minimizing the interference between different access patterns.

5.1 Skyway Overview

Fig.8 presents the overall architecture of Skyway. There are three architecture components added to the direct datapath.

• *Property Buffer* (*PBuf*). PBuf sits between the cores and main memory, shared by all cores. It temporarily stores state data on-chip only (see Subsection 5.3).

• *Duplication Row* (*DRow*). DRow is a group of extra buffers attached to the row buffer in each DRAM bank. It backs up specific data from the row buffer when being triggered (see Subsection 5.4). Especially, the DRow is managed in the granularity of segment (i.e., 1 KB) to enhance the performance.

• *Duplication Row Monitor* (*DRowM*). DRowM is a small table maintained in the memory controller, which tracks the information of the segments in DRow for each bank (see Subsection 5.4).

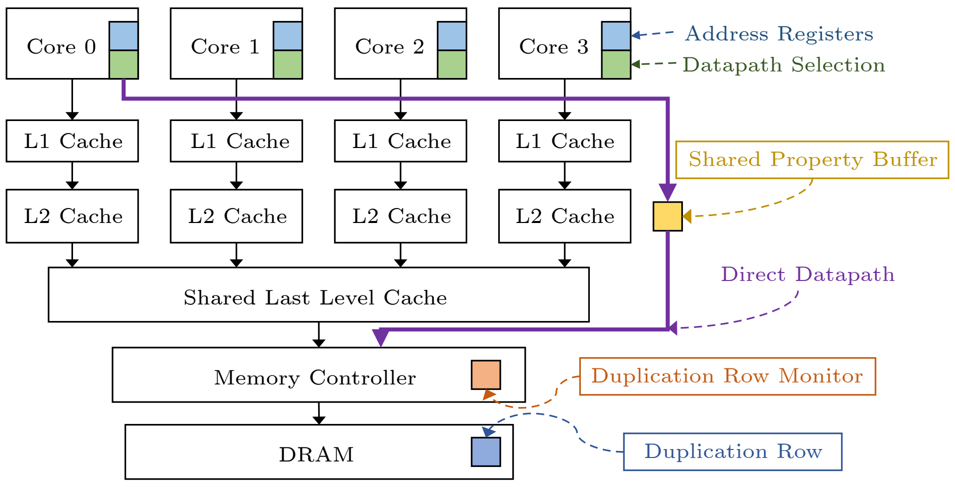


Fig.8. Overview of Skyway hardware structure integrated with a four-core system. The Skyway components are shown in color.

Additionally, Skyway makes minor modifications in the core and memory controller. In the core, pairs of address registers are initialized to guide datapath selection logic (see Subsection 5.2). Besides, each core contains a new port and selectively sends the memory requests to the L1 cache or the PBuf. The memory controller also needs to support returning data to two on-chip buffers (LLC or PBuf). The fine-grained direct datapath is shown in Fig.8 with the purple line.

5.2 Datapath Selection

As described in Subsection 5.1, Skyway enables the core to send memory access requests to either the L1 cache or the PBuf. To support such a feature, Skyway uses six 64-bit registers to track the start and end addresses of the three arrays and one 64-bit register to record the end address of hot vertices in the state array[3]. These registers are initialized when the application allocates memory space for a graph.

To classify a memory request, Skyway adds three fields to a normal memory request as shown in Fig.9.

	2 Bits	1 Bit	64 Bits
Original Request	Array Type	Hot Bit	Vertex ID

Fig.9. Extended memory request format in Skyway.

• *Array Type*, which indicates the type of accessing array;

• *Hot Bit*, which indicates whether the target element is a hot vertex (for state array access only);

• *Vertex ID*, which indicates the ID of accessing vertex (for state array access only).

For each memory request to be issued, the core classifies it as the offset, edge, state or other type by comparing the request address with address registers, and fills the Array Type field. Especially, for the state type access, the core also calculates the target vertex ID and fills the Vertex ID field as follows:

$$vertexID = \frac{Addr_{\text{request}} - Addr_{\text{state\_start}}}{Size_{\text{state\_element}}},$$

where $Addr_{\text{request}}$ is the request address, $Addr_{\text{state\_start}}$ is the start address of the state array, and $Size_{\text{state\_element}}$ is the size of one state array element, which is usually 4-byte or 8-byte. If the request address is within the range of hot vertices, its Hot Bit is set to 1. Then the core sends state type requests to PBuf and the other types of requests to the L1 cache as normal.

5.3 Property Buffer

PBuf is a shared hardware component motivated by the fact that the three-level cache hierarchy is inefficient for irregular state requests. In general, PBuf improves application performance in two ways: 1) it provides a single-level cache structure and allows the requests with irregular access patterns to arrive at the memory controller quickly; 2) it manages the entries in the granularity of state array element to hold more elements and allow more accurate updates.

5.3.1 Hardware Design

As shown in Fig.10, PBuf consists of the follow-

[3]In this work, the total hot vertices occupy no more than the size of LLC; therefore, the end address of hot vertex states is 8 MB + the start address of the state array.

ing three components.

• *Property Data Cache* (*ProCache*). Just like a small-sized cache, ProCache is managed in a set-associative way and divided into two arrays: 1) tag array, which stores the vertex ID, dirty and valid bits, plus position bits (see Subsection 5.3.3); 2) data array, which stores state array elements.

• *Backup Cacheline Buffer* (*LineBuf*). LineBuf is a buffer that stores a small number of cachelines from the memory controller (16 cachelines per core in our work). Since ProCache uses fine-grained space management and may generate a large number of requests to the DRAM, LineBuf provides a lightweight second-hit-chance for ProCache to find the data.

• *Granularity Match* (*GraMatch*). GraMatch matches the different request granularities between LineBuf and ProCache. When reading data from LineBuf, GraMatch checks the requested state array elements in the LineBuf entries. Once hit in LineBuf, GraMatch selects and loads the fine-grained state array data to ProCache. When evicting a ProCache entry, GraMatch generates the write-back memory address based on the vertex ID of the evicted data.

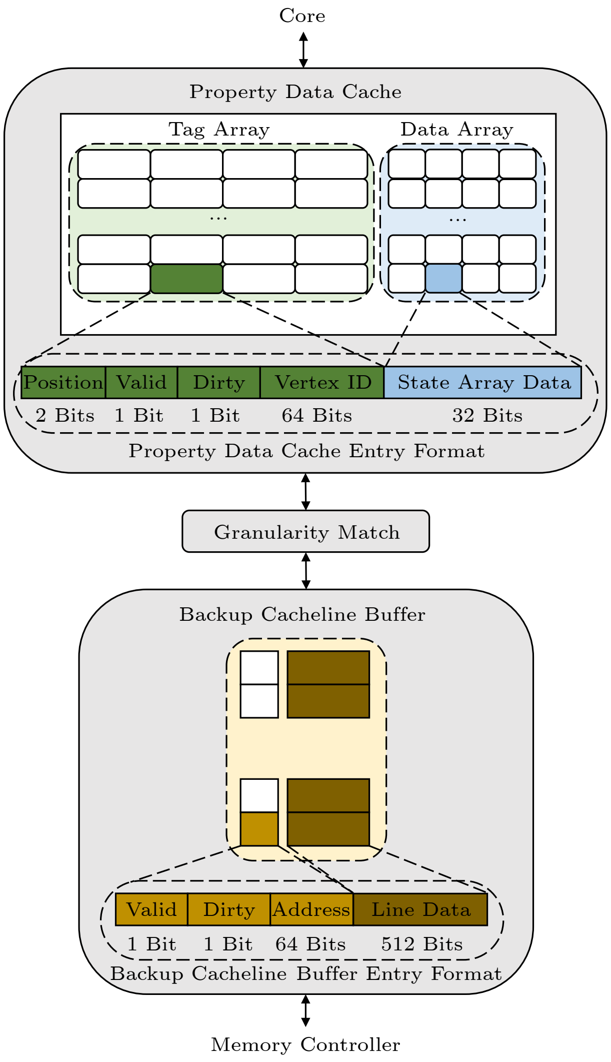


Fig.10. Property buffer hardware design.

5.3.2 Workflow

Fig.11 summarizes the workflow of PBuf. In general, when receiving a request, PBuf first checks if the request can be served by ProCache with the vertex ID in the request. For a missed request, PBuf searches LineBuf, seeking a second-hit-chance. Finally, the request missed in LineBuf will be sent to DRAM.

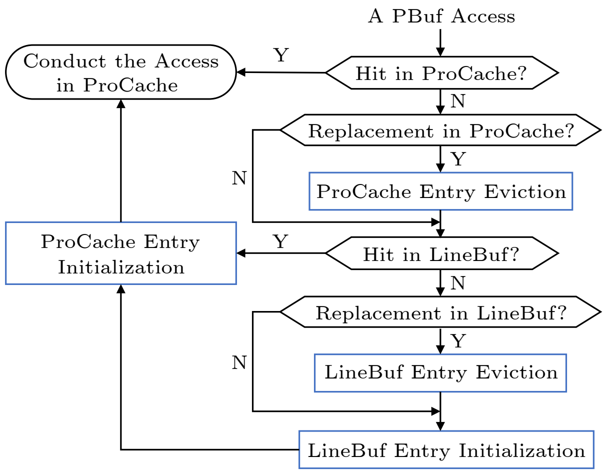


Fig.11. Property buffer workflow.

The above procedure includes the following four main operations (marked with the blue boxes in Fig.11).

• *ProCache Entry Eviction*. ProCache evicts an entry if there is no free entry for initialization. To this end, ProCache conducts the following operations. 1) ProCache chooses the LRU entry in the located set to make room for the new entry. 2) ProCache writes back the evicted entry if the dirty bit is set. In this case, GraMatch first calculates the cacheline address using the simple arithmetic as $Addr_{\rm request} = (Addr_{\rm state\_start} + Size_{\rm state\_element} \times VertexID) << 6$. In order to reduce expensive DRAM write operations, PBuf first tries to write the corresponding cacheline back to LineBuf. Once failed, the write-back request will be sent to the memory controller. 3) ProCache invalidates the evicted entry, setting the valid bit to 0.

• *LineBuf Entry Eviction*. LineBuf evicts the LRU entry when there is no available entry. For a dirty cacheline, LineBuf generates a memory write request and sends it to the memory controller; otherwise, the data in the evicting entry is directly dropped.

• *LineBuf Entry Initialization.* When initializing a new entry, LineBuf sets its dirty bit to 0 and its valid bit to 1, and then loads cacheline from DRAM.

• *ProCache Entry Initialization.* To initialize a new entry, ProCache first sets its dirty bit to 0 and its valid bit to 1. Then, ProCache selectively loads the required data from the LineBuf entry through GraMatch.

5.3.3 *Large-Size State Array Element Support*

In the evaluated application BC, two state arrays store different state values for each vertex. We merge the two arrays into one to exploit better access locality. In this case, the state array element size is larger (i.e., 16 bytes) than in other applications (usually 8 bytes). To accommodate large-size state elements, ProCache uses multiple continuous entries within one set to record one large state array element. As shown in Fig.10, each ProCache tag entry contains a 2-bit position field to record the relative position in a state array element. When receiving a request for a large-size element, ProCache selects all entries that match the requested vertex ID as a response. Then ProCache forwards these entries to the core one by one in the order based on the position field. For ProCache replacement, all entries with the same vertex ID will be evicted and loaded together. In the current version, we employ a 2-bit field to support the 16-byte array element. However, this mechanism enables PBuf to be extended to any larger access granularity. In extreme cases, PBuf may use a 4-bit field to support a 64-byte cacheline. In that case, PBuf is accessed as a normal cache.

5.4 Duplication Row and Monitor

As discussed in Subsection 4.2, various access patterns co-exist in graph applications and may cause serious performance interference at the row buffer. In the meantime, we also observe that locality in the memory blocks with small granularity does exist, which can be utilized for better performance. To reduce the interference and exploit the locality, we propose Duplication Row (DRow), a specialized optimization scheme to manage the row buffer for graph applications. The key idea of DRow is to preserve the data with moderate locality in an extra buffer space with fine-grained data management support. In this way, data accesses with moderate locality can benefit from the interference reduction. Note that DRow does not modify DRAM management. Data transmissions between disk and DRAM are as usual when required data is not found in DRAM.

Compared with prior DRAM optimizations[35, 45, 46], our scheme detects the unique features in graph applications without any historical records or future predictions. With the help of several registers, we clarify accesses accurately and process them separately. We will give a more detailed discussion in Section 7.

In our design, we select the hot state array elements and the edge array elements to be preserved in DRow. There are two reasons for choosing the two data types for DRow. First, according to skewed power-law distribution, hot vertices occupy most connections and exhibit high reuse probability in smaller granularity memory blocks, indicating that maintaining hot state array elements in DRow leads to more DRow hit opportunities. Second, duplication of the edge array helps to prevent data from being flushed by other irregular accesses and provides a better locality. Overall, DRow reduces the time-costly row buffer update operations and allows the DRAM to serve more requests with a shortened latency. We choose the duplication granularity based on the sensitivity study given in Subsection 6.8. We believe that the performance could be enhanced further through an adaptive granularity selection process in DRow, which is left to future work.

5.4.1 *Hardware Design*

As shown in Fig.12, DRow allocates extra buffers in each bank affiliated with the row buffer. In our design, DRow has the same width with conventional rows (i.e., 8 KB) but each row is segmented (i.e., 1 KB) to improve DRow utilization. To track which segments are currently duplicated in DRow, the memory controller maintains a DRowM vector for each bank. Each entry in the DRowM has a row tag to identify the duplication source row, a segment tag to record segment ID, a dirty bit, and a valid bit. There are four DRows in each bank, and each DRow contains up to eight segments, requiring 32 DRowM entries for each bank in the memory controller.

5.4.2 *Workflow*

Fig.13 summarizes the workflow of DRow. First,

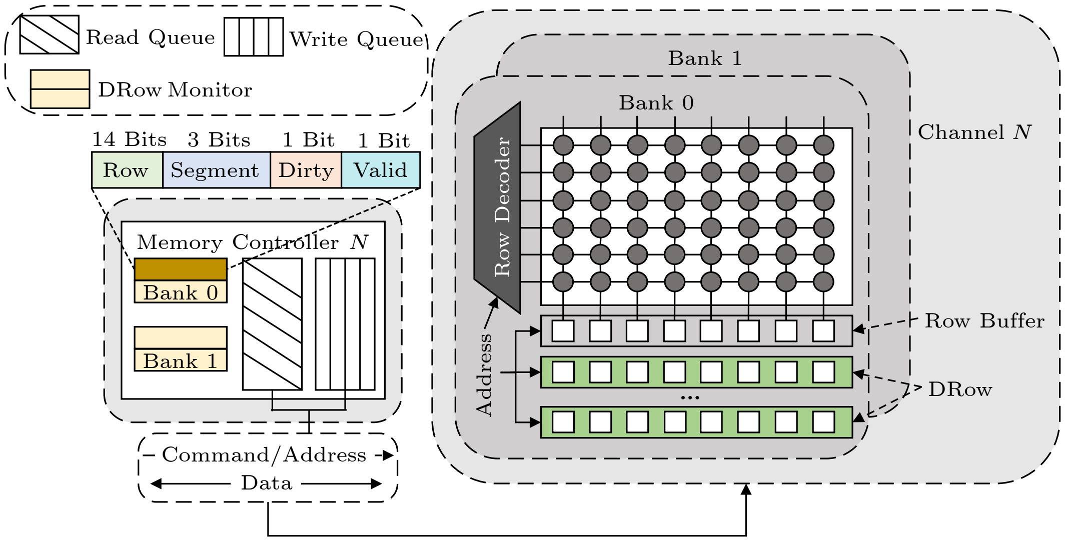


Fig.12. Duplication row hardware design. The added components are shown in color.

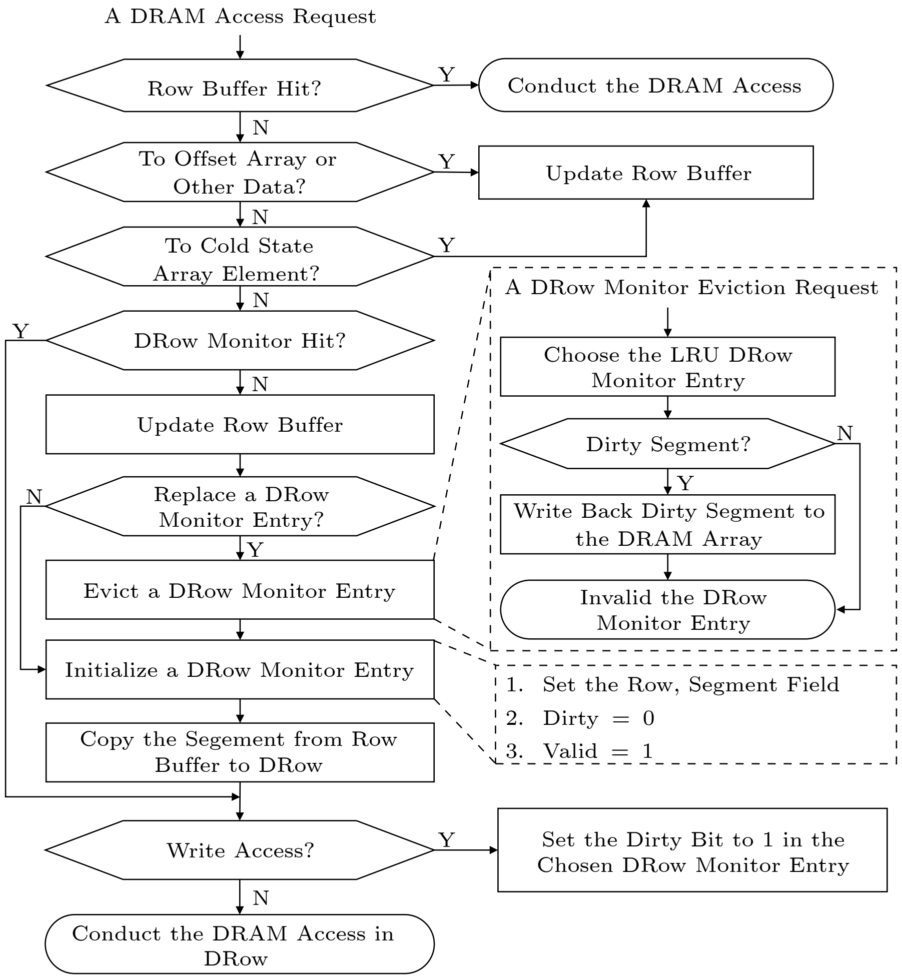

A DRAM Access Request
Row Buffer Hit?
Y
Conduct the DRAM Access
N
To Offset Array or Other Data?
Y
Update Row Buffer
N
To Cold State Array Element?
Y
N
DRow Monitor Hit?
Y
N
Update Row Buffer
Replace a DRow Monitor Entry?
N
Y
Evict a DRow Monitor Entry
Initialize a DRow Monitor Entry
Copy the Segement from Row Buffer to DRow
Write Access?
Y
Set the Dirty Bit to 1 in the Chosen DRow Monitor Entry
N
Conduct the DRAM Access in DRow
A DRow Monitor Eviction Request
Choose the LRU DRow Monitor Entry
Dirty Segment?
N
Y
Write Back Dirty Segment to the DRAM Array
Invalid the DRow Monitor Entry
1. Set the Row, Segment Field
2. Dirty = 0
3. Valid = 1

Fig.13. Duplication row workflow.

the memory controller reads the appended fields of each request and identifies requests to the hot state array elements and edge array elements. These identified requests are called trigger requests. For a trigger request, DRowM checks the contained records to see if the requested data is a hit in the DRow. If it is, the request is done at DRow. Otherwise, DRowM first conducts the command to DRAM, waiting for the requested row to be loaded into the row buffer. Then memory controller copies the target segment from the row buffer to DRow and records the information of the segment in the newly allocated DRowM entry, waiting for the next access. For a DRow write, the dirty bit in the corresponding DRowM entry will be set to 1. For a non-trigger request, DRAM processes it following the conventional workflow.

Copying a Row Buffer Segment to DRow. To efficiently copy a segment from the row buffer to DRow, we introduce Target-copy (T-copy), a new DRAM command with duplication source and destination knowledge. The T-copy command refers to CROW[35], working with the following procedures: 1) the memory controller sends the source segment ID in the row buffer and the destination segment ID in DRow to DRAM with the issued T-copy command; 2) DRAM selects the source segment in the row buffer, then reads the segment and writes it to the destination segment in DRow. The process of reading and copying the segment is just like a regular read from the row buffer. At the end of the T-copy command procedure, both the row buffer and DRow hold a duplication of the target segment.

Evicting a DRow Segment. If DRowM is full, a DRowM eviction request will be conducted before new entry allocation. To fulfill such a function, we implement the Target-precharge (T-pre) command. T-pre is similar to the regular precharge command and extended for holding the source segment ID in DRow and the target row, and the target segment ID in the DRAM array. When evicting a dirty DRowM entry, the memory controller sends the T-pre command to DRAM. Then DRAM activates the target row and latches the bitlines of the target segment. After this, the data in the source segment is written to the target row. Finally, the DRowM entry is invalidated and free to be allocated.

5.5 Implementation Overhead

We estimate the additional overhead for Skyway with the configuration as shown in Table 5.

Overall, the proposed Skyway requires additional 141.5 KB on-chip storage and extra 4 MB for DRow at the DRAM side. We use CACTI 6.5[47] to evaluate the area overhead. The results show that PBuf adds only 2.6% of the area consumed by the 8 MB LLC and DRow introduces only 0.02% additional area overhead of a 16 GB DRAM. The storage cost of Skyway is minimum compared with its performance improvement.

6 Evaluation

In order to evaluate the effectiveness of Skyway, we first show the experimental results of PBuf, DRow, and Skyway individually for better comparison. We compare our design with existing state-of-the-art hardware optimizations[16, 48] and cache bypassing schemes[49, 50]. The baseline is without any optimizations. The evaluation metrics include performance and bandwidth utilization. Then, we quantitatively analyze how Skyway effectively improves the performance. At last, we provide a series of sensitivity studies of different Skyway design choices.

6.1 Evaluation Setup

The simulation platform, graph applications, and datasets used in the evaluation are introduced in Section 3. Besides, Table 5 gives the detailed configurations of Skyway. Note that in our experiments, we assume that the first 8 MB data in the state array, with the same capacity as the LLC, is the hot vertices[16].

We evaluate Skyway and compare it with the state-of-the-art schemes described below.

• DRRIP[④][48] focuses on SPEC benchmarks, which initializes re-reference bits of cachelines based on Set

Table 5. Skyway Configurations and Hardware Overhead

Hardware	Configuration	Overhead
PBuf	ProCache: 32 KB per core, shared, 4-way associated, 4 B-entry, 4-cycle latency; LineBuf: 1 KB per core, shared, 1-way associated, 64 B-entry, 2-cycle latency	132 KB
DRow	8 KB per extra buffer, eight segments in one buffer, four buffers per bank, tCCD five cycles, LRU replacement policy	4 MB
DRowM	32 entries per bank, 19 bits per entry	9.5 KB
Register	One 64-bit register to record the end address of hot vertices in state array, six 64-bit registers to record array address range (start and end)	56 B

[④]https://github.com/ChampSim/ChampSim/blob/master/replacement/drrip.llc_repl, Jul. 2023.

Dueling and chooses the best replacement policy for different benchmarks.

• GRASP⑤[16] is proposed for graph applications, which classifies cachelines into hot and cold regions based on vertex degrees and guarantees hot cachelines to stay longer in cache.

• Core-DRAM, L1-DRAM, and L2-DRAM[49, 50] are three idealized cache bypass schemes without any buffer and looking up latency in the bypass path. The bypass path is used to forward state array data accesses to DRAM from Core, L1, and L2, respectively.

• Double-L1 doubles the L1 capacity in the baseline (i.e., 64 KB L1, which is the sum of L1 and PBuf). Double-L1 classifies that the performance of Skyway is not from additional hardware resources.

6.2 Performance

We use the instructions per cycle (IPC) to denote the system performance. Fig.14 summarizes the normalized performance improvement of DRRIP, GRASP, Core-DRAM, L1-DRAM, L2-DRAM, Double-L1, PBuf, DRow, and Skyway over the baseline.

As shown in Fig.14, using PBuf alone outperforms the baseline with speedups for BFS, BC, CC, PR, and SSSP of 17.6%, 13.4%, 15.6%, 3.7%, and 36.9%, respectively, averaging across all graph datasets. Overall, PBuf yields 17% average speedup and up to 78% in the best case on SSSP-PLD (for convenience, we abbreviate the specific workload as application-dataset in the rest of the paper) over the

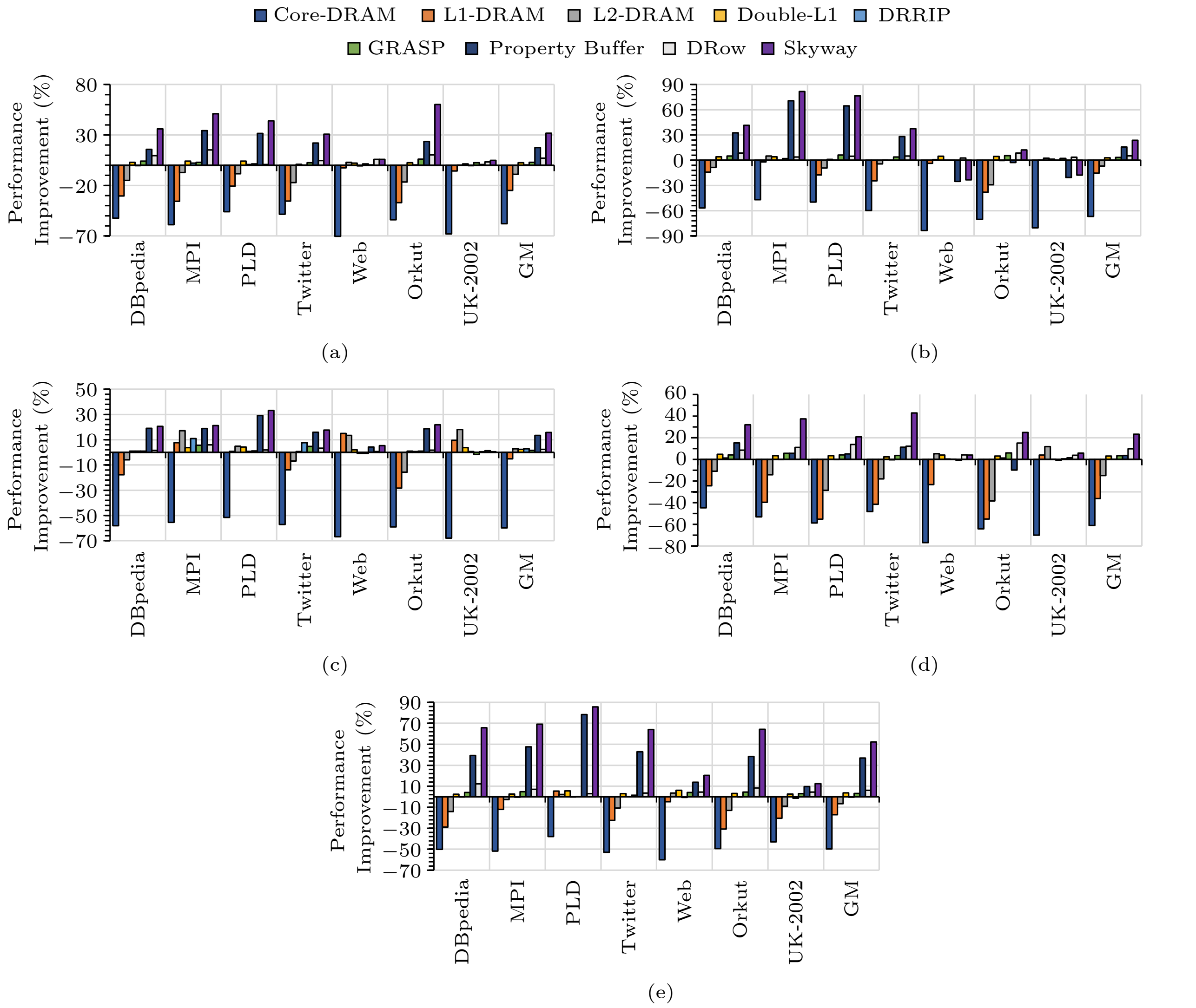


Fig.14. Performance improvements of (a) BFS, (b) BC, (c) CC, (d) PR, and (e) SSSP over the baseline.

⑤https://github.com/faldupriyank/grasp, Jul. 2024.

baseline. These improvements come from the more efficient direct datapath working on irregular requests. Also, using DRow alone provides an average speedup of 8.4% for BFS, 2.3% for BC, 5.2% for CC, 9.8% for PR, and 6.1% for SSSP. Among all the 35 workloads, DRow yields 5.7% speedup on average and up to 15% in the best case (on PR-Orkut) over the baseline. Finally, Skyway with integrated PBuf and DRow achieves the performance improvement of 32% for BFS, 16% for BC, 24% for CC, 23% for PR, and 52% for SSSP. Benefiting from the two optimizations, Skyway yields an average speedup of 29% and up to 86% over the baseline. As for prior techniques, DRRIP only slightly improves performance by 2%, and GRASP yields an average speedup of 5% over the baseline. In comparison, without buffers in the direct path, Core-DRAM, L1-DRAM, and L2-DRAM yield an average speedup of –59%, –20%, and –7%, respectively, over the baseline on the reordered datasets. On average, Double-L1 only improves the performance by 2.8% over the baseline. Because the state array accesses in graph applications are very irregular, simply increasing cache capacity is not an effective optimization.

6.3 Bandwidth Utilization

Fig.15 presents the normalized DRAM bandwidth utilization of different schemes. Compared with the baseline, PBuf improves the bandwidth utilization by 1.91x on average. Besides, DRow helps to improve the bandwidth by 7.8% on average and up to 17% in the best case (on BFS-Orkut). Finally, Skyway achieves the improvement of bandwidth utilization by 2.13x on average and up to 5.87x in the application PR with the dataset Orkut.

The reasons for the improvement of bandwidth utilization can be summarized as follows.

• PBuf reduces the latency for the requests to ar-

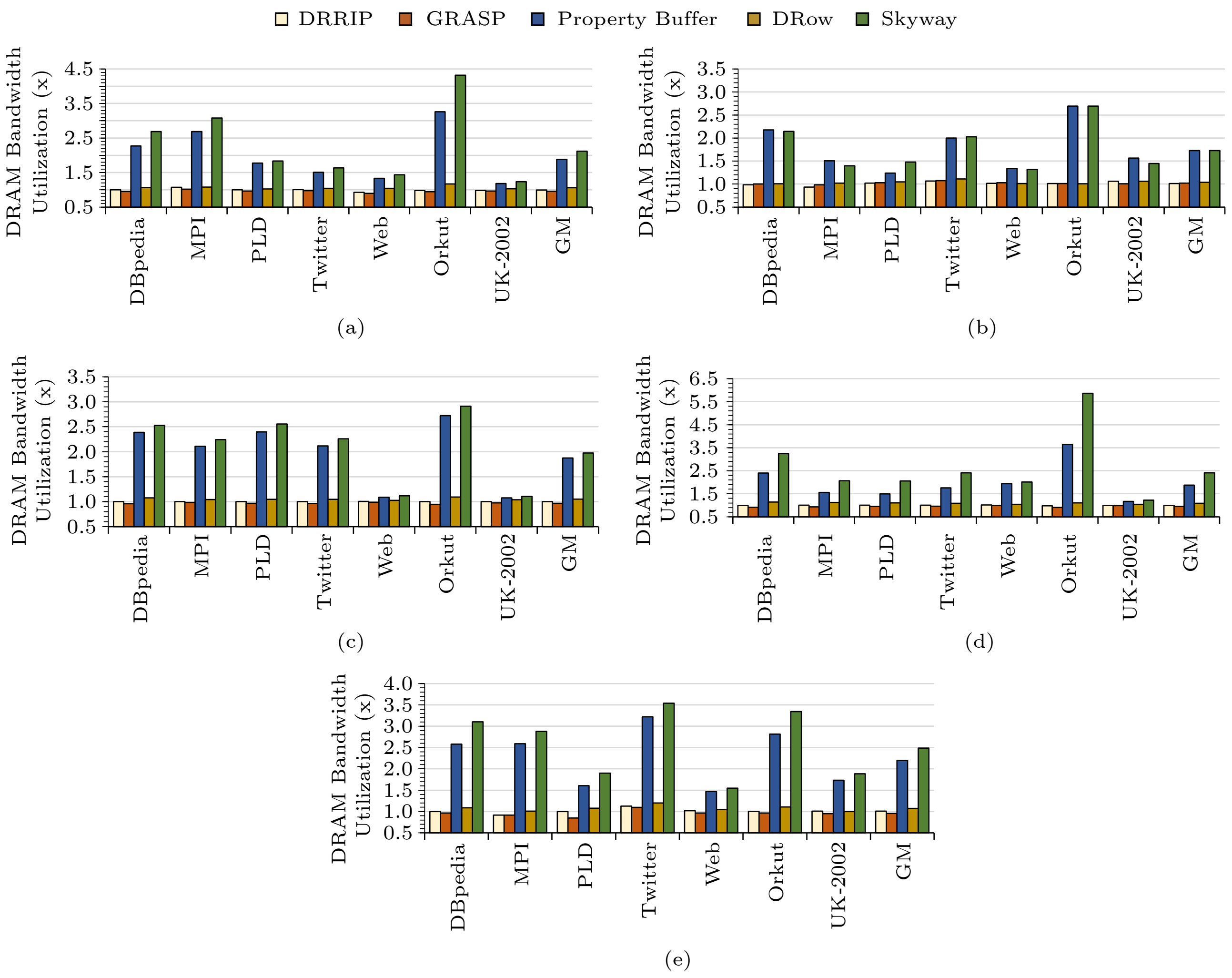


Fig.15. DRAM bandwidth utilization of (a) BFS, (b) BC, (c) CC, (d) PR, and (e) SSSP over the baseline.

rive at the memory controller. For the requests to the state array with low locality, PBuf has optimized the datapath with less hierarchies and fine-grained buffer management. For the other requests, the efficiency of the cache hierarchy is improved without the inference of low locality data accesses.

• DRow helps to minimize the impact of irregular access patterns in DRAM. With DRow, DRAM directly serves accesses to the hot state array elements and edge array elements without updating the row buffer, which allows DRAM to serve more requests within a shortened time interval.

Among prior techniques, DRRIP fails to learn reuse patterns in graph applications, and GRASP is only efficient for high-skewed graphs. We find that GRASP receives the highest performance improvement on the high-skewed graphs but is less effective on the low-skewed graphs. Table 6 shows the vertex percentage and corresponding edge percentage on two datasets. For example, in the dataset DBpedia, 46% of the vertices occupy 99% of the total edges, while in the dataset Orkut, only 23% of the vertices occupy 99% of the total edges. The higher the skew, the lower the vertex percentage. Therefore Orkut is high-skewed and DBpedia is low-skewed. Our evaluation indicates that GRASP yields only 3.7% speedup on the low-skewed DBpedia graph. On the contrary, Skyway accelerates the graph applications by 38.4% on the graph dataset DBpedia. Because GRASP classifies the state array into several regions based on vertex degrees, the scale of the hot region may exceed cache capacity on low-skewed graphs, which limits the accelerating ability. Unlike GRASP, on the cache side, Skyway classifies data accesses based on which array they belong to. Skyway is not based on the skew feature and works well even on low-skewed graphs. Furthermore, both DRRIP and GRASP process requests with different locality following a unified strategy. Application performance is degraded by the interfered access patterns, which lowers the DRAM bandwidth utilization.

Table 6. Power-Law Distribution of DBpedia and Orkut

Edge Percentage (%)	Vertex Percentage (%)	
	DBpedia	Orkut
70	7	5
75	9	6
80	11	7
85	13	8
90	16	11
95	21	15
99	46	23

In summary, the efficiency of Skyway comes from the ingenious combination of hardware optimization and software framework execution characteristics. We find that multiple data access patterns exist in graph applications. However, the current multi-level cache hierarchy and row buffer design work well only for regular data accesses. Inspired by this key observation, we believe that "divide and conquer" is a promising hardware optimization. Moreover, such an optimization reduces cache pollution because irregular data will not be stored in cache. As a result, Skyway improves the bandwidth utilization and the performance of graph applications.

6.4 Impact of Dual-Path on the Memory Controller

Cache bypassing techniques, as well as PBuf design in Skyway, increase the temporal density of memory requests arriving at the memory controller. Therefore, we show how these techniques affect the memory traffic and performance.

Fig.16 presents the memory traffic of Core-DRAM, L1-DRAM, L2-DRAM, and PBuf over the baseline. On average, PBuf produces 1.29x, 1.64x, 1.43x, 1.59x, and 1.31x data traffic from the memory controller to DRAM for BFS, BC, CC, PR, and SSSP, respectively, over the baseline. However, the other three cache bypassing techniques are not effective, with an average memory traffic of 9.45x, 2.39x, and 2.06x, respectively, over the baseline. The three cache bypassing managements produce too many DRAM accesses. They are unable to find a trade-off between faster DRAM accesses and more DRAM accesses, failing to limit memory traffic in a tolerable area and thus hurting the performance. In contrast, PBuf benefits from the fine-granularity organization and independence from cache hierarchy, resulting in fewer DRAM accesses even compared with L2-DRAM.

Additionally, we further analyze the dynamic occupancy of the DRAM request queue in PBuf. As shown in Fig.17, taking dataset DBpedia as an example, the dynamic occupancies of both read and write queues are under 10 across all the five applications, which is far below the common queue capacity (i.e., 64-entry in our configuration) and will not cause the read or write drain. Therefore, we can summarize that the increment of memory requests caused by the pro-

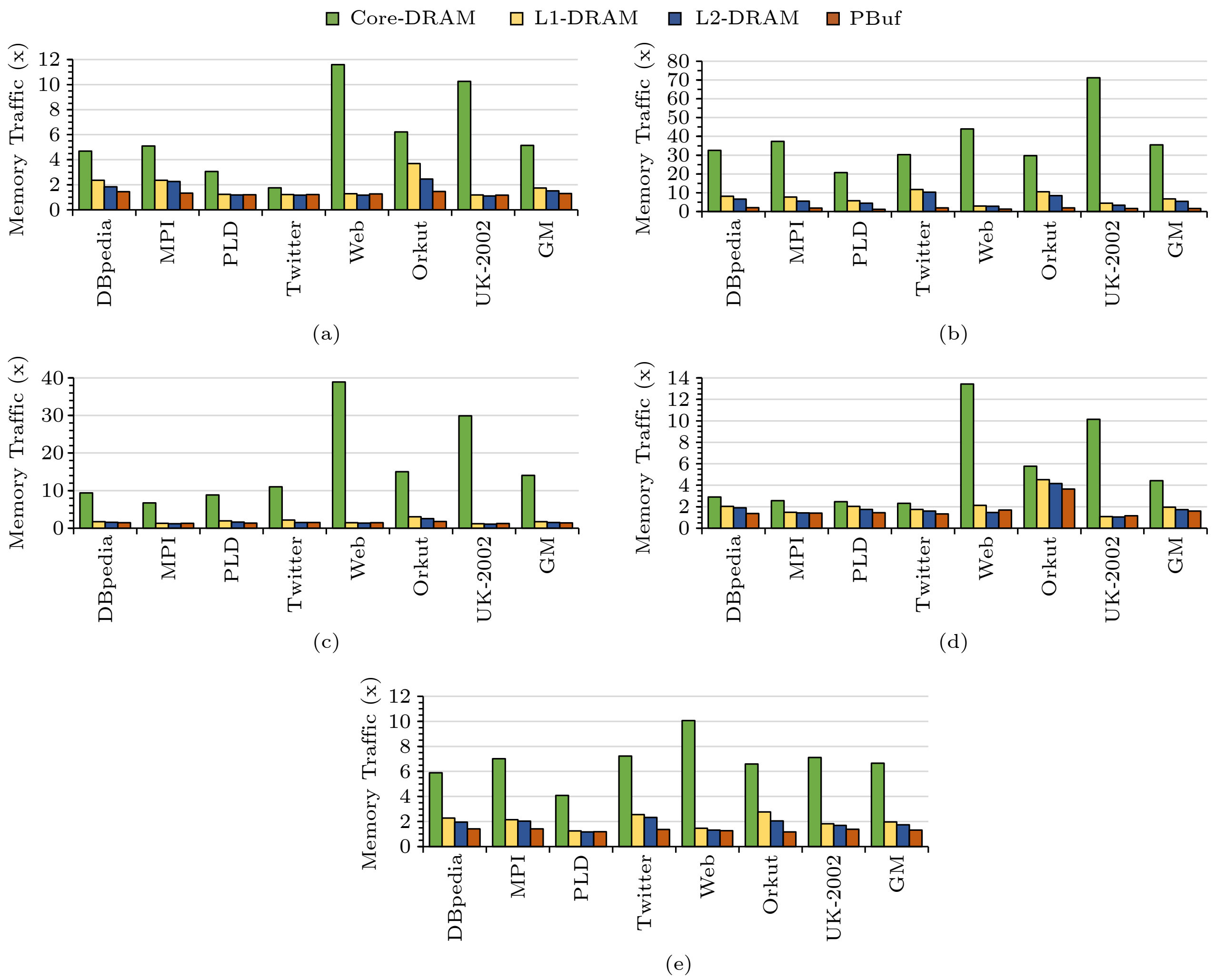


Fig.16. Memory traffic of (a) BFS, (b) BC, (c) CC, (d) PR, and (e) SSSP over the baseline.

posed PBuf will not hurt the system performance.

6.5 Row Buffer Conflicts Reduction with DRow

DRow design benefits the application from two aspects. First, it protects the accesses with moderate or even high temporal locality, avoiding them being interrupted in the row buffer by irregular accesses. Second, it imports a fast datapath to return buffered accesses. Overall, DRow reduces row buffer conflicts significantly. As Fig.18 shows, on average, DRow reduces row buffer conflicts by 15% over the baseline and up to 58% in the application PR with dataset Orkut.

6.6 Limitations of Skyway

In our evaluation, DRow always improves the performance. The exception happens in PBuf. We find that PBuf cannot accelerate graph applications when the input graphs show a good community feature[10] (i.e., Web and UK-2002). In such graphs, state array data accesses perform a good locality and utilize multi-layer cache hierarchy efficiently. Unfortunately, PBuf cannot leverage multi-layer cache hierarchy and decreases the performance in some scenarios (e.g., CC-Web). As a result, although DRow speeds up the execution in all scenarios, Skyway degrades the performance over the baseline in specific cases because of PBuf. We leave the study of classifying the graph inputs based on their community feature and exploiting PBuf adaptively for future work.

6.7 Sensitivity of ProCache and LineBuf Capacity

Fig.19(a) shows the PBuf performance with 8 KB, 16 KB, 32 KB, 64 KB, and 128 KB ProCache per

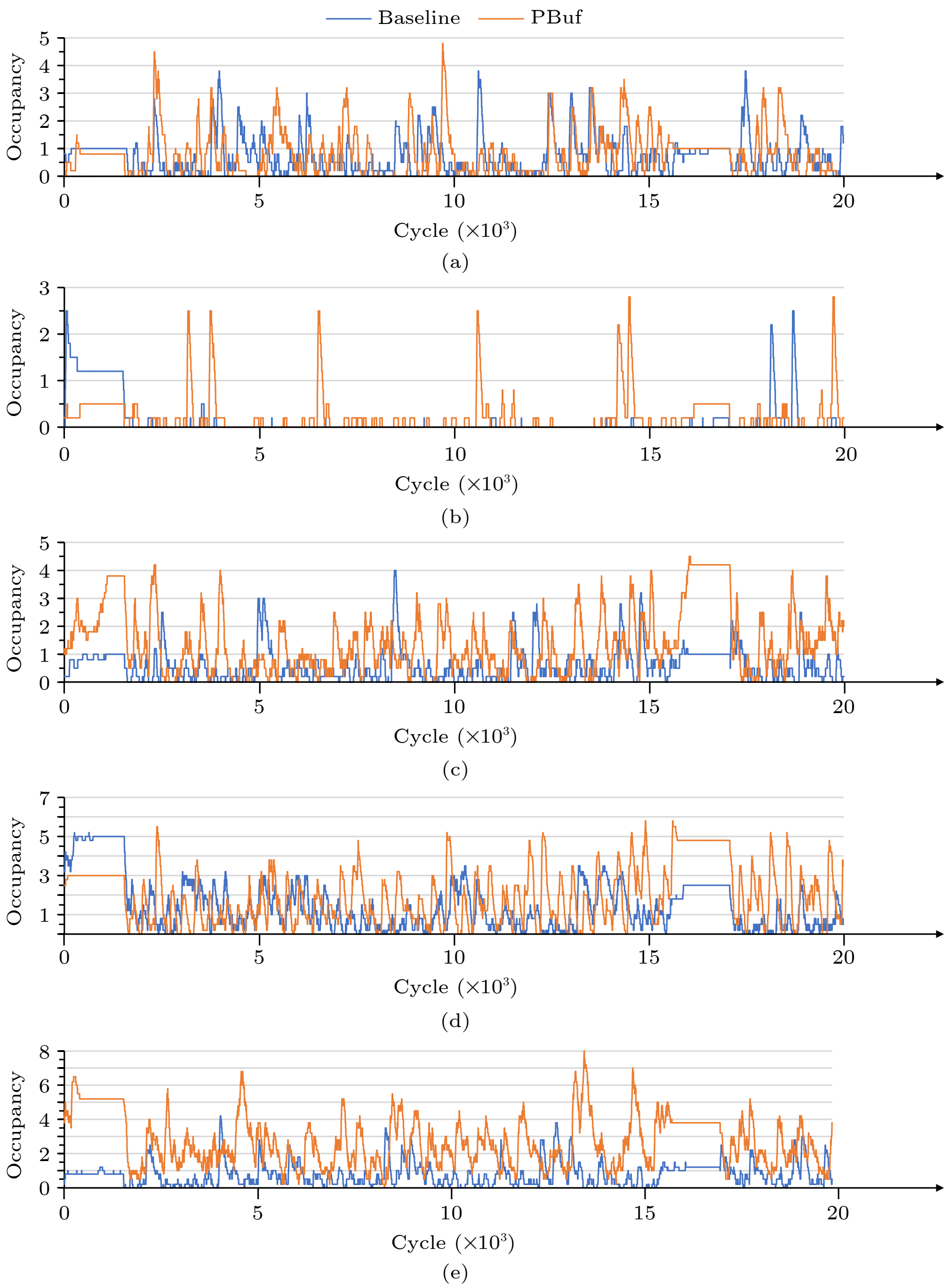


Fig.17. Run-time request queue occupancy of PBuf and the baseline of (a) BFS, (b) BC, (c) CC, (d) PR, and (e) SSSP.

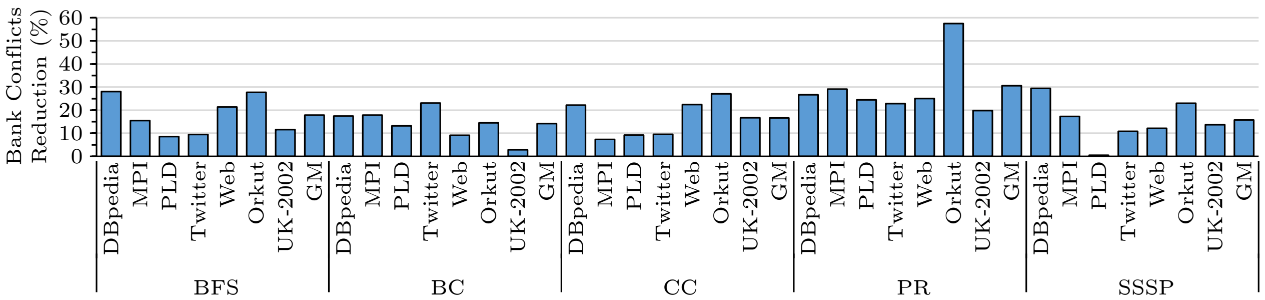


Fig.18. Bank conflicts reduction of DRow over the baseline.

core. We observe that doubling ProCache capacity from 16 KB to 32 KB and 32 KB to 64 KB brings application speedups of 2% and 0.8%, respectively. To achieve a balance between hardware overhead and performance improvement, we use 32 KB ProCache per core in our evaluation.

We also measure the performance sensitivity to LineBuf capacity. Fig.19(b) shows the average speedups when varying LineBuf cachelines. We notice that using more backup cachelines slightly outperforms fewer backup cachelines, mainly because the irregular state access pattern makes cacheline less efficient. We choose 16 backup cachelines to save extra hardware overhead and allow applications to exploit locality.

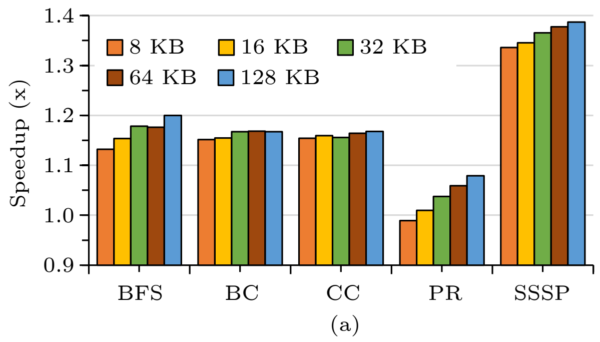
6.8 Sensitivity of DRow Capacity and Segment Size

To check the effectiveness of the DRow capacity, we vary the setting from 1 row to 16 rows and compare the system performance. As shown in Fig.20(a), a larger DRow always gains a better performance by providing more hit opportunities. However, the performance increment is not multiplied with a double capacity. On average, application speedups are from 6.2% (4 rows) to 8.3% (8 rows) and then to 10% (16 rows). To balance the performance and storage overhead, we implement four rows per bank.

The segment count refers to the number of segments in one DRow row and determines the granularity of DRow. As shown in Fig.20(b), as we vary the segment counts from 1 (8 KB per segment) to 32 (256 B per segment), a larger segment is beneficial when multiple access addresses are adjacent. However, a smaller segment size is more appropriate for data with low spatial locality but with a better reuse rate. It is hard to find a perfect segment size that outperforms all the other sizes across all workloads because the access pattern is application- and graph-dependent. We choose the 1 KB segment in our evaluations since it receives the best performance in most applications.

7 Related Work

Data Duplication in DRAM. Duplicon Cache[46] reserves a specialized space in each bank and maintains an accessing counter for each row to determine which row should be duplicated. However, since how to determine the threshold of its counter is indefinite, it is difficult to gain steady performance improvement in



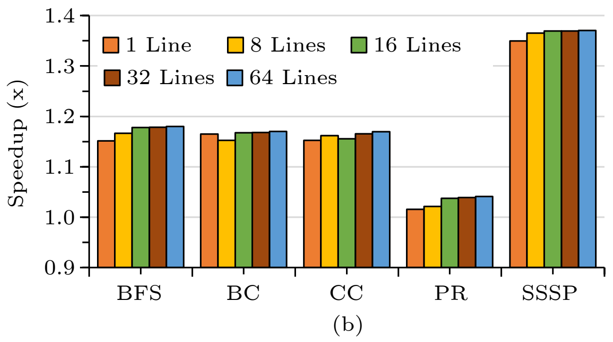
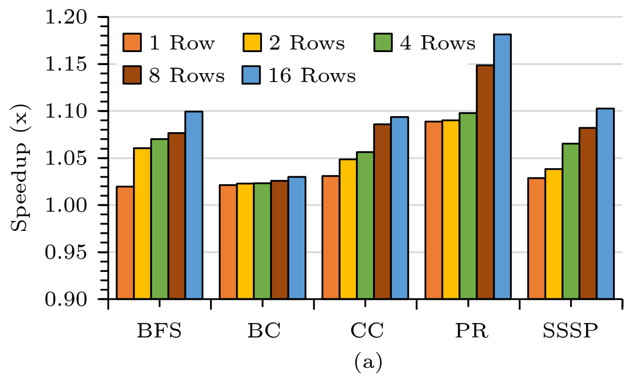


Fig.19. Performance improvements for different (a) ProCache capacities and (b) LineBuf capacities.



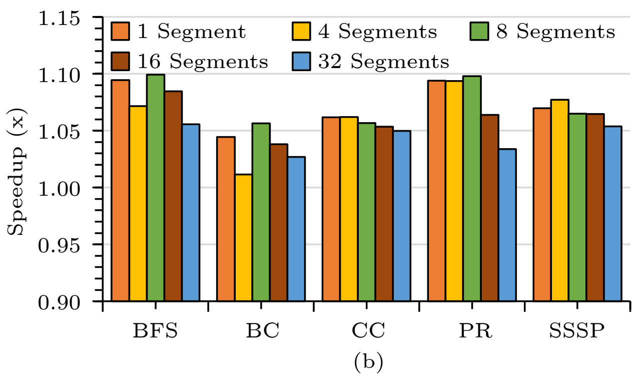


Fig.20. Performance improvements for different (a) DRow numbers and (b) segments in each row.

practice, especially in graph applications with complex access patterns. CROW[35] reserves a space within each bank and copies data from the row buffer when a row buffer miss occurs in the granularity of segments. FIGARO[45] extends CROW to duplicate data in finer granularity. Both CROW and FIGARO ignore the fact that bank conflicts occurred by different memory requests have different influences on the overall performance. Unlike prior work[35, 45, 46], Skyway utilizes the information of different access patterns for various key data structures in graph applications to direct the data duplication, which simplifies the accurate identification of the duplication.

Cache Bypassing. Adaptive Cache Bypassing[49], Annex Cache[51], LMP[52], and Random Sampling Filtered Cache[53] utilize a predictor to determine whether a memory access should bypass the cache. AMB[54], RHP and RTP[55], and LRF[50] track the history access information for cache blocks, skipping specific cache layers or bypassing memory requests to DRAM based on the recorded knowledge. Our proposed Skyway outperforms the prior work in three ways. First, Skyway directly utilizes the special memory access characteristics of graph applications according to simplified address comparison, and determines whether to use the direct datapath, which is accurate compared with prior prediction-based schemes and avoids the unnecessary history tracking latency. Second, prior work[49, 51–53] focuses on selecting a specific path but ignores the reuse within cachelines. Our direct datapath is organized in fine-grained data management, which improves the hardware resource utilization for serving the irregular access pattern in graph applications. Finally, we not only optimize cache design but also extend the key idea to DRAM.

GPU-Based Graph Processing. GPUs are popular accelerators for graph processing. However, as the graph size grows, the performance of graph processing is limited by available device memory capacity. Grus[56] reduces page faults through a clever unified-memory management scheme (e.g., reducing memory footprint and prefetching graph data). Moreover, Grus reduces expensive atomic operations through low-cost write operations. Skywalker[57] proposes a novel graph sampling and random walk algorithm to eliminate the capacity gap between input graphs and GPU capacity. Subway[58] generates a subgraph in almost every iteration to minimize data movements between CPU and GPU. Unlike these optimizations, we focus on CPUs. The main bottleneck we solve is the inefficiency of the multi-layer cache hierarchy and the row buffer design on graph applications.

8 Conclusions

This paper showed that a graph can be represented in three data arrays, while only memory requests to the state array exhibit irregular access patterns. The current memory hierarchy is far from fully utilized for graph applications due to random and unpredictable memory accesses. To accelerate graph applications, this paper presented Skyway, a data-aware hardware architecture with 1) a fine-grained direct datapath from core to main memory, opening a fast path for irregular requests, and 2) a memory-side row buffer hardware, preserving selected data segments before flushing them back. In doing so, Skyway processes memory requests efficiently by mitigating the memory access interference. On a set of graph workloads, Skyway improves application performance by 29% on average and up to 86% over the baseline without any hardware optimizations. Skyway also outperforms GRASP and DRRIP, which are the existing state-of-the-art hardware optimizations. While Skyway is motivated by graph processing, the key idea behind the design can be extended to accelerate any applications with multiple access patterns.

Conflict of Interest Xian-He Sun is an associate editor for Journal of Computer Science and Technology and was not involved in the editorial review of this article. All authors declare that there are no other competing interests.

References

[1] Fan W F. Graph pattern matching revised for social network analysis. In *Proc. the 15th International Conference on Database Theory*, Mar. 2012, pp.8–21. DOI: 10.1145/2274576.2274578.

[2] Kwak H, Lee C, Park H, Moon S. What is Twitter, a social network or a news media? In *Proc. the 19th International Conference on World Wide Web*, Apr. 2010, pp.591–600. DOI: 10.1145/1772690.1772751.

[3] Tang L, Liu H. Graph mining applications to social network analysis. In *Managing and Mining Graph Data*, Aggarwal C C, Wang H X (eds.), Springer, 2010, pp.487–513. DOI: 10.1007/978-1-4419-6045-0_16.

[4] Caetano T S, McAuley J J, Cheng L, Le Q V, Smola A J. Learning graph matching. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2009, 31(6): 1048–1058. DOI: 10.1109/TPAMI.2009.28.

[5] Navlakha S, Schatz M C, Kingsford C. Revealing biological modules via graph summarization. *Journal of Computational Biology*, 2009, 16(2): 253–264. DOI: 10.1089/cmb.2008.11TT.

[6] Han S, Liu X Y, Mao H Z, Pu J, Pedram A, Horowitz M A, Dally W J. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 2016, 44(3): 243–254. DOI: 10.1145/3007787.3001163.

[7] Mukkara A, Beckmann N, Abeydeera M, Ma X S, Sanchez D. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *Proc. the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2018. DOI: 10.1109/MICRO.2018.00010.

[8] Arai J, Shiokawa H, Yamamuro T, Onizuka M, Iwamura S. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *Proc. the 2016 IEEE International Parallel and Distributed Processing Symposium*, May 2016, pp.22–31. DOI: 10.1109/IPDPS.2016.110.

[9] Balaji V, Lucia B. When is graph reordering an optimization? Studying the effect of lightweight graph reordering across applications and input graphs. In *Proc. the 2018 IEEE International Symposium on Workload Characterization*, Sept. 30–Oct. 2, 2018, pp.203–214. DOI: 10.1109/IISWC.2018.8573478.

[10] Faldu P, Diamond J, Grot B. A closer look at lightweight graph reordering. In *Proc. the 2019 IEEE International Symposium on Workload Characterization*, Nov. 2019. DOI: 10.1109/IISWC47752.2019.9041948.

[11] Lakhotia K, Singapura S, Kannan R, Prasanna V. ReCALL: Reordered cache aware locality based graph processing. In *Proc. the 24th International Conference on High Performance Computing*, Dec. 2017, pp.273–282. DOI: 10.1109/HiPC.2017.00039.

[12] Wei H, Yu J X, Lu C, Lin X M. Speedup graph processing by graph ordering. In *Proc. the 2016 International Conference on Management of Data*, Jun. 2016, pp.1813–1828. DOI: 10.1145/2882903.2915220.

[13] Zhang Y M, Kiriansky V, Mendis C, Amarasinghe S, Zaharia M. Making caches work for graph analytics. In *Proc. the 2017 IEEE International Conference on Big Data*, Dec. 2017, pp.293–302. DOI: 10.1109/BigData.2017.8257937.

[14] Zou M, Zhang M Z, Wang R J, Sun X H, Ye X C, Fan D R, Tang Z M. Accelerating graph processing with lightweight learning-based data reordering. *IEEE Computer Architecture Letters*, 2022, 21(1): 5–8. DOI: 10.1109/LCA.2022.3151087.

[15] Balaji V, Crago N, Jaleel A, Lucia B. P-OPT: Practical optimal cache replacement for graph analytics. In *Proc. the 2021 IEEE International Symposium on High-Performance Computer Architecture*, Feb. 27–/Mar. 3, 2021, pp.668–681. DOI: 10.1109/HPCA51647.2021.00062.

[16] Faldu P, Diamond J, Grot B. Domain-specialized cache management for graph analytics. In *Proc. the 2020 IEEE International Symposium on High Performance Computer Architecture*, Feb. 2020, pp.234–248. DOI: 10.1109/HPCA47549.2020.00028.

[17] Mukkara A, Beckmann N, Sanchez D. PHI: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates. In *Proc. the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2019, pp.1009–1022. DOI: 10.1145/3352460.3358254.

[18] Rahman S, Abu-Ghazaleh N, Gupta R. GraphPulse: An event-driven hardware accelerator for asynchronous graph processing. In *Proc. the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2020, pp.908–921. DOI: 10.1109/MICRO50266.2020.00078.

[19] Yan M Y, Hu X, Li S C, Basak A, Li H, Ma X, Akgun I, Feng Y J, Gu P, Deng L, Ye X C, Zhang Z M, Fan D R, Xie Y. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In *Proc. the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2019, pp.615–628. DOI: 10.1145/3352460.3358318.

[20] Zhang D, Ma X Y, Thomson M, Chiou D. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. *ACM SIGPLAN Notices*, 2018, 53(2): 593–607. DOI: 10.1145/3296957.3173197.

[21] Zhang Y, Liao X F, Jin H, He L G, He B S, Liu H K, Gu L. DepGraph: A dependency-driven accelerator for efficient iterative graph processing. In *Proc. the 2021 IEEE International Symposium on High-Performance Computer Architecture*, Feb. 27–Mar. 3, 2021, pp.371–384. DOI: 10.1109/HPCA51647.2021.00039.

[22] Zou M, Yan M Y, Li W M, Tang Z M, Ye X C, Fan D R. GEM: Execution-aware cache management for graph analytics. In *Proc. the 22nd International Conference on Algorithms and Architectures for Parallel Processing*, Oct. 2022, pp.273–292. DOI: 10.1007/978-3-031-22677-9_15.

[23] Maass S, Min C, Kashyap S, Kang W, Kumar M, Kim T. Mosaic: Processing a trillion-edge graph on a single machine. In *Proc. the 12th European Conference on Computer Systems*, Apr. 2017, pp.527–543. DOI: 10.1145/3064176.3064191.

[24] Shun J L, Blelloch G E. Ligra: A lightweight graph processing framework for shared memory. In *Proc. the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2013, pp.135–146. DOI: 10.1145/2442516.2442530.

[25] Beamer S, Asanović K, Patterson D. The GAP benchmark suite. arXiv: 1508.03619, 2015. https://doi.org/10.48550/arXiv.1508.03619, Jan. 2024.

[26] Kyrola A, Blelloch G, Guestrin C. GraphChi: Large-scale graph computation on just a PC. In *Proc. the 10th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2012, pp.31–46.

[27] Sundaram N, Satish N, Patwary M M A, Dulloor S R, Vadlamudi S G, Das D, Dubey P. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 2015, 8(11): 1214–1225. DOI: 10.14778/2809974.2809983.

[28] Faloutsos M, Faloutsos P, Faloutsos C. On power-law relationships of the Internet topology. In *The Structure and*

Dynamics of Networks, Newman M, Barabási A L, Watts D J (eds.), Princeton University Press, 2006, pp.195–206. DOI: 10.1515/9781400841356.195.

[29] Gonzalez J E, Low Y, Gu H J, Bickson D, Guestrin C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. the 10th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2012, pp.17–30.

[30] Jiang L, Chen L S, Qiu J. Performance characterization of multi-threaded graph processing applications on many-integrated-core architecture. In *Proc. the 2018 IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2018, pp.199–208. DOI: 10.1109/ISPASS.2018.00033.

[31] Sanchez D, Kozyrakis C. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer Architecture News*, 2013, 41(3): 475–486. DOI: 10.1145/2508148.2485963.

[32] Li S, Yang Z Y, Reddy D, Srivastava A, Jacob B. DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator. *IEEE Computer Architecture Letters*, 2020, 19(2): 106–109. DOI: 10.1109/LCA.2020.2973991.

[33] Basak A, Li S C, Hu X, Oh S M, Xie X F, Zhao L, Jiang X W, Xie Y. Analysis and optimization of the memory hierarchy for graph processing workloads. In *Proc. the 2019 IEEE International Symposium on High Performance Computer Architecture*, Feb. 2019, pp.373–386. DOI: 10.1109/HPCA.2019.00051.

[34] Rixner S, Dally W J, Kapasi U J, Mattson P, Owens J D. Memory access scheduling. *ACM SIGARCH Computer Architecture News*, 2000, 28(2): 128–138. DOI: 10.1145/342001.339668.

[35] Hassan H, Patel M, Kim J S, Yaglikci A G, Vijaykumar N, Ghiasi N M, Ghose S, Mutlu O. CROW: A low-cost substrate for improving DRAM performance, energy efficiency, and reliability. In *Proc. the 46th International Symposium on Computer Architecture*, Jun. 2019, pp.129–142. DOI: 10.1145/3307650.3322231.

[36] Beamer S, Asanovic K, Patterson D. Direction-optimizing breadth-first search. In *Proc. the 2012 International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2012. DOI: 10.1109/SC.2012.50.

[37] Madduri K, Ediger D, Jiang K, Bader D A, Chavarria-Miranda D. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Proc. the 2009 IEEE International Symposium on Parallel & Distributed Processing*, May 2009. DOI: 10.1109/IPDPS.2009.5161100.

[38] Sutton M, Ben-Nun T, Barak A. Optimizing parallel graph connectivity computation via subgraph sampling. In *Proc. the 2018 IEEE International Parallel and Distributed Processing Symposium*, May 2018, pp.12–21. DOI: 10.1109/IPDPS.2018.00012.

[39] Zhang Y M, Brahmakshatriya A, Chen X Y, Dhulipala L, Kamil S, Amarasinghe S, Shun J. Optimizing ordered graph algorithms with Graphit. In *Proc. the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, Feb. 2020, pp.158–170. DOI: 10.1145/3368826.3377909.

[40] Auer S, Bizer C, Kobilarov G, Lehmann J, Cyganiak R, Ives Z. DBpedia: A nucleus for a web of open data. In *Proc. the 6th International Semantic Web Conference on the Semantic Web*, Nov. 2007, pp.722–735. DOI: 10.1007/978-3-540-76298-0_52.

[41] Lehmberg O, Meusel R, Bizer C. Graph structure in the web: Aggregated by pay-level domain. In *Proc. the 2014 ACM Conference on Web Science*, Jun. 2014, pp.119–128. DOI: 10.1145/2615569.2615674.

[42] Kunegis J. KONECT: The Koblenz network collection. In *Proc. the 22nd International Conference on World Wide Web*, May 2013, pp.1343–1350. DOI: 10.1145/2487788.2488173.

[43] Cha M, Haddadi H, Benevenuto F, Gummadi K. Measuring user influence in Twitter: The million follower fallacy. In *Proc. the 2010 International AAAI Conference on Web and Social Media*, May 2010, pp.10–17. DOI: 10.1609/icwsm.v4i1.14033.

[44] Davis T A, Hu Y F. The university of Florida sparse matrix collection. *ACM Trans. Mathematical Software*, 2011, 38(1): Article No. 1. DOI: 10.1145/2049662.2049663.

[45] Wang Y H, Orosa L, Peng X J, Guo Y, Ghose S, Patel M, Kim J S, Luna J G, Sadrosadati M, Ghiasi N M, Mutlu O. FIGARO: Improving system performance via fine-grained In-DRAM data relocation and caching. In *Proc. the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2020, pp.313–328. DOI: 10.1109/MICRO50266.2020.00036.

[46] Lin B, Healy M B, Miftakhutdinov R, Emma P G, Patt Y. Duplicon cache: Mitigating off-chip memory bank and bank group conflicts via data duplication. In *Proc. the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2018, pp.285–297. DOI: 10.1109/MICRO.2018.00031.

[47] Muralimanohar N, Balasubramonian R, Jouppi N P. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proc. the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2007, pp.3–14. DOI: 10.1109/MICRO.2007.33.

[48] Jaleel A, Theobald K B, Steely S C, Emer J. High performance cache replacement using re-reference interval prediction (RRIP). In *Proc. the 37th International Symposium on Computer Architecture*, Jun. 2010, pp.60–71. DOI: 10.1145/1815961.1815971.

[49] Gupta S, Gao H L, Zhou H Y. Adaptive cache bypassing for inclusive last level caches. In *Proc. the 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp.1243–1253. DOI: 10.1109/IPDPS.2013.16.

[50] Xiang L X, Chen T Z, Shi Q S, Hu W. Less reused filter: Improving L2 cache performance via filtering less reused lines. In *Proc. the 23rd International Conference on Supercomputing*, Jun. 2009, pp.68–79. DOI: 10.1145/1542275.1542290.

[51] John L K, Subramanian A. Design and performance evaluation of a cache assist to implement selective caching. In *Proc. the 1997 International Conference on Computer De-*

sign VLSI in Computers and Processors, Oct. 1997, pp.510–518. DOI: 10.1109/ICCD.1997.628916.

[52] Malkowski K, Link G, Raghavan P, Irwin M J. Load miss prediction-exploiting power performance trade-offs. In *Proc. the 2007 IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007. DOI: 10.1109/IPDPS.2007.370536.

[53] Etsion Y, Feitelson D G. Exploiting core working sets to filter the L1 cache with random sampling. *IEEE Trans. Computers*, 2012, 61(11): 1535–1550. DOI: 10.1109/TC.2011.197.

[54] Collins J D, Tullsen D M. Hardware identification of cache conflict misses. In *Proc. the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, Nov. 1999, pp.126–135. DOI: 10.1109/MICRO.1999.809450.

[55] Jalminger J, Stenstrom P. A novel approach to cache block reuse predictions. In *Proc. the 2003 International Conference on Parallel Processing*, Oct. 2003, pp.294–302. DOI: 10.1109/ICPP.2003.1240592.

[56] Wang P Y, Wang J, Li C, Wang J Z, Zhu H J, Guo M Y. Grus: Toward unified-memory-efficient high-performance graph processing on GPU. *ACM Trans. Architecture and Code Optimization*, 2021, 18(2): Article No. 22. DOI: 10.1145/3444844.

[57] Wang P Y, Li C, Wang J, Wang T L, Zhang L, Leng J W, Chen Q, Guo M Y. Skywalker: Efficient alias-method-based graph sampling and random walk on GPUs. In *Proc. the 30th International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2021, pp.304–317. DOI: 10.1109/PACT52795.2021.00029.

[58] Sabet A H N, Zhao Z J, Gupta R. Subway: Minimizing data transfer during out-of-GPU-memory graph processing. In *Proc. the 15th European Conference on Computer Systems*, Apr. 2020, Article No. 12. DOI: 10.1145/3342195.3387537.

Mo Zou received her Bachelor's degree in software engineering from Shandong University, Jinan, in 2017, and her Ph.D. degree in computer architecture from University of Chinese Academy of Sciences, Beijing, in 2023. She is now a postdoc researcher in Institute of Computing Technology, Chinese Academy of Sciences, Beijing. Her research interests include computer architecture and memory system, especially on domain-specific hardware optimization.

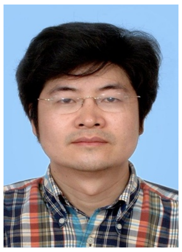
Ming-Zhe Zhang is currently an associate professor at the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing. His research interests include NVM, memory-centric architecture, and domain specific accelerators.

Ru-Jia Wang received her Bachelor's degree in automation from Zhejiang University, Hangzhou, in 2013, and her M.S. and Ph.D. degrees in electrical and computer engineering from the University of Pittsburgh, Pittsburgh, in 2015 and 2018, respectively. She is now an assistant professor in computer science at the Illinois Institute of Technology, Chicago. Her research interests are in the broader computer architecture and systems area, including scalable, secure, reliable, and high-performance memory systems and architectures.

Xian-He Sun is a University Distinguished Professor and the Ron Hochsprung Endowed Chair of the Department of Computer Science at the Illinois Institute of Technology (Illinois Tech), Chicago. Dr. Sun is an IEEE Fellow and is known for his memory-bounded speedup model, also called Sun-Ni's Law, for scalable computing. His research interests include high-performance computing, memory and I/O systems, and performance evaluation and optimization. Dr. Sun is the Editor-in-Chief of IEEE Transactions on Parallel and Distributed Systems. Dr. Sun received the Golden Core Award from IEEE CS Society in 2017, the Overseas Outstanding Contributions Award from CCF in 2018. More information about Dr. Sun can be found at his website: www.cs.iit.edu/~scs/sun.

Xiao-Chun Ye received his Ph.D. degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences (CAS), Beijing, in 2010. Currently he is a professor and the director of the High-Throughput Computer Research Center in Institute of Computing Technology, CAS, Beijing. His main research interests include many-core processor architecture and graph accelerator.

Dong-Rui Fan received his Ph.D. degree in computer architecture from Institute of Computing Technology, Chinese Academy of Sciences (CAS), Beijing, in 2005. He is currently a professor and Ph.D. supervisor in Institute of Computing Technology, CAS, Beijing. His main research interests include high-throughput computer architecture and high performance computer architecture.



Zhi-Min Tang received his B.S. degree from the Department of Computer Science, Nanjing University, Nanjing, in 1985, and his Ph.D. degree from Institute of Computing Technology, Chinese Academy of Sciences (CAS), Beijing, in 1990, both in computer science. He is currently a professor in Institute of Computing Technology, CAS, Beijing. His research interests include high performance computer architecture, parallel processing, and VLSI design.