NVMW '22

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

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

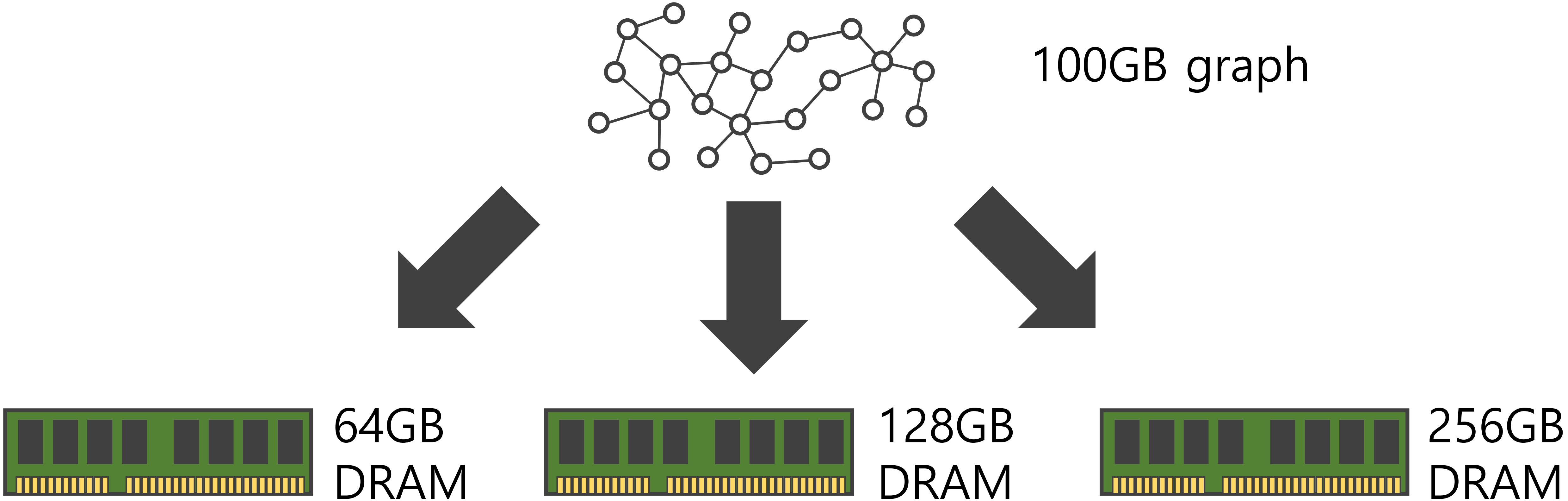
KAIST[1], Sookmyung Women's University[2]

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMELab

How Much RAM Do We Need?



100GB graph

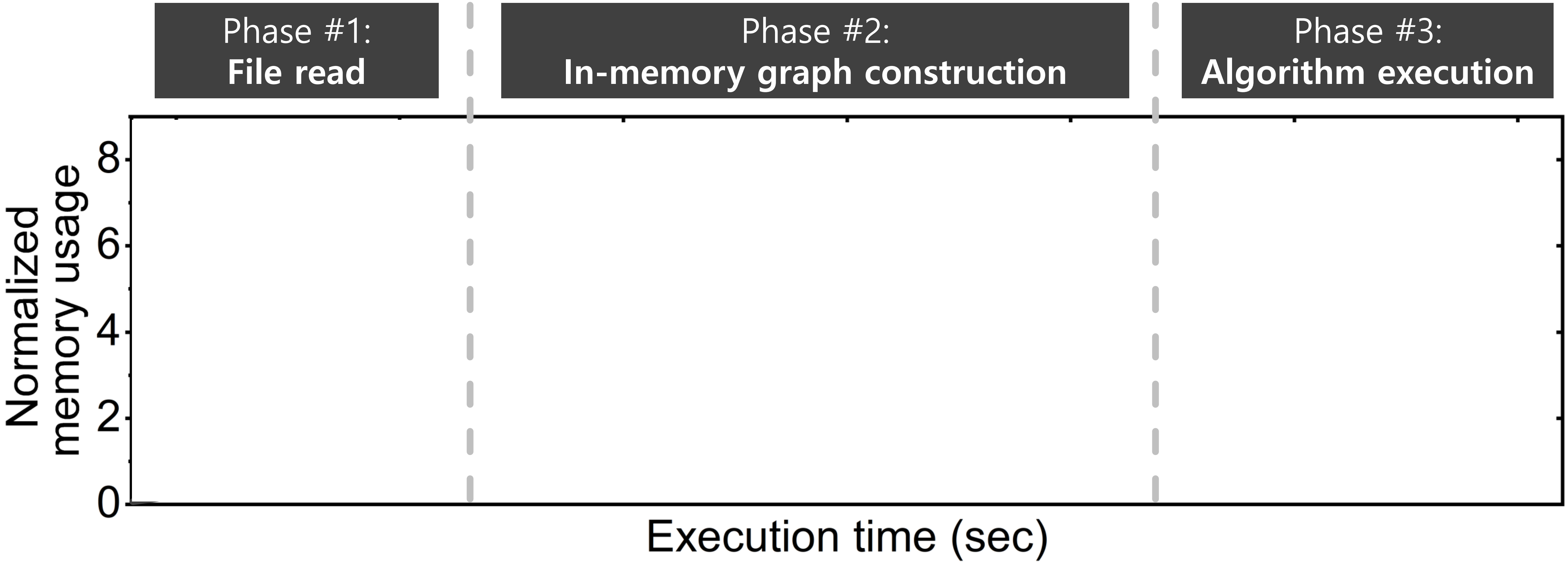
64GB DRAM

128GB DRAM

256GB DRAM

“None of them is sufficient”

Data Amplification



Phase #1: **File read**

Phase #2: **In-memory graph construction**

Phase #3: **Algorithm execution**

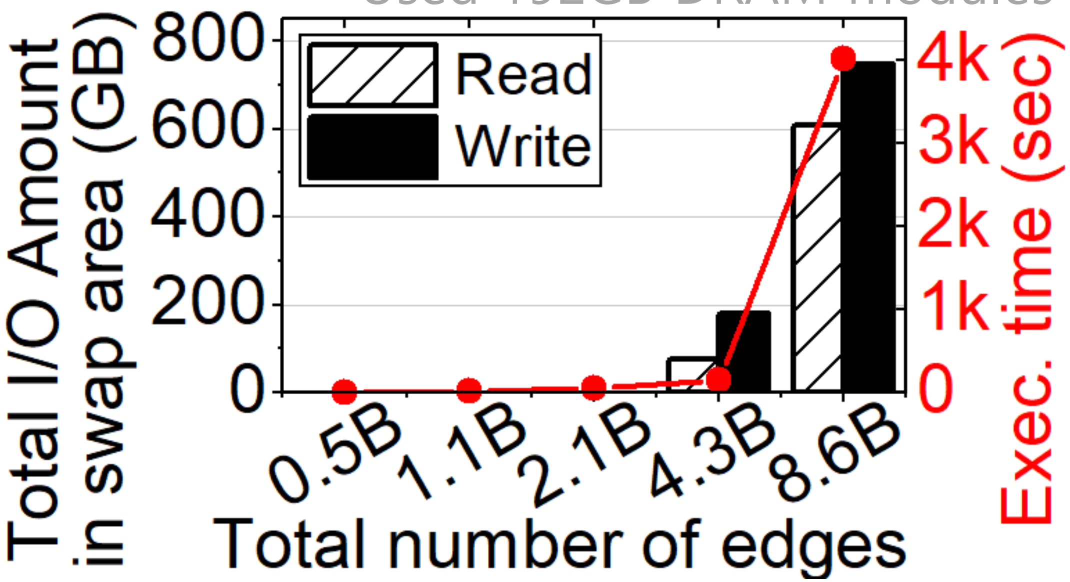
Observation #1: Runtime data > raw graph data

Out of Memory and Memory Expansion Overhead

“Out of memory”

Serious swap overhead

** Used 192GB DRAM modules



➢ Need to increase the size of system memory

Available Solution: Non-Volatile Memory (NVM)

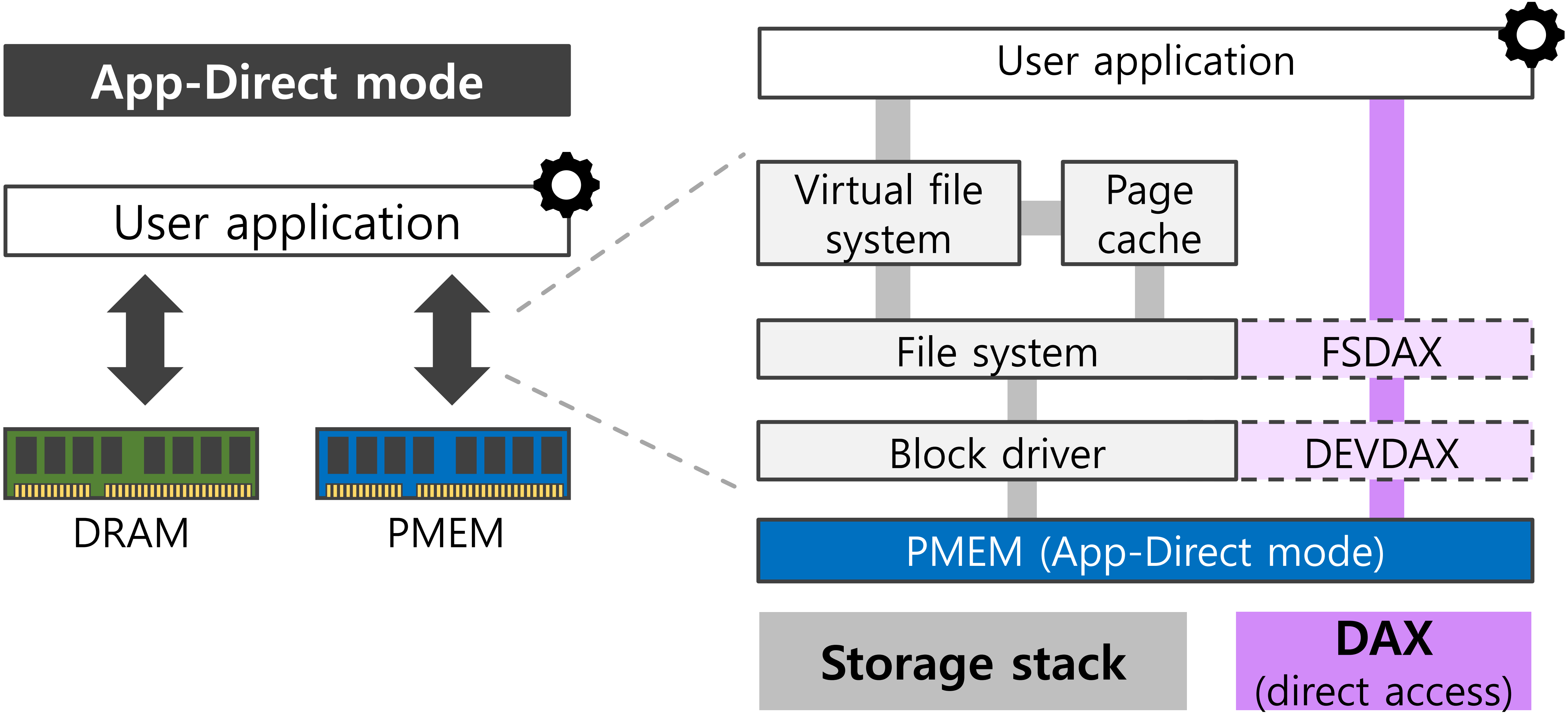
App-Direct mode

Memory mode

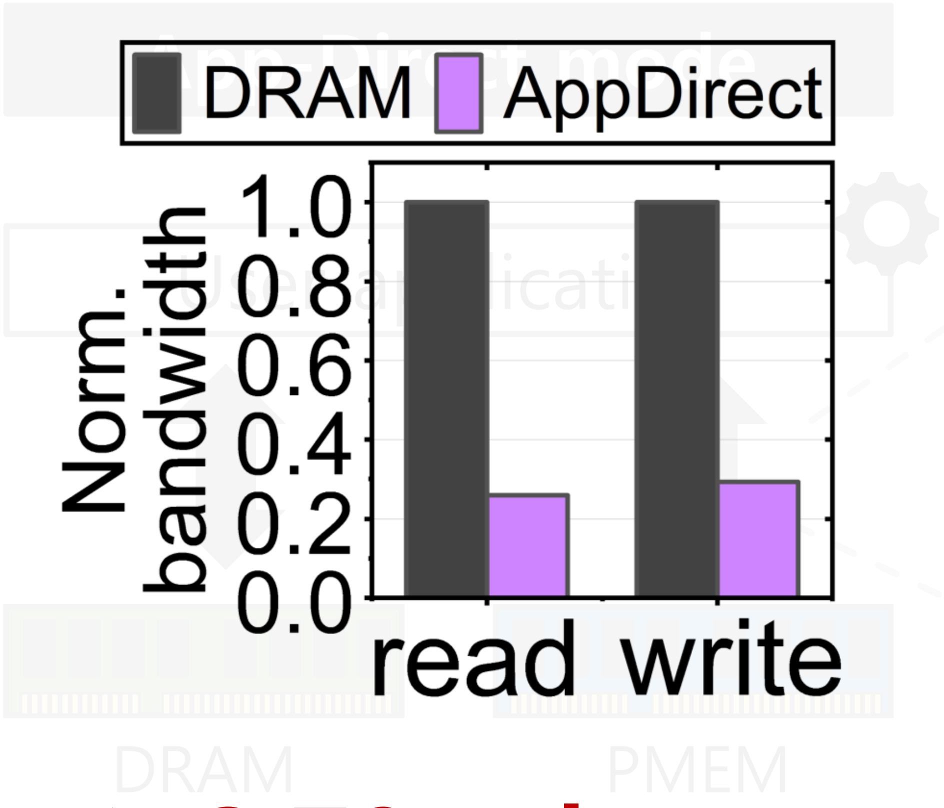
Intel Optane
Persistent Memory Module (PMEM)

- 8x denser than conventional DRAMs

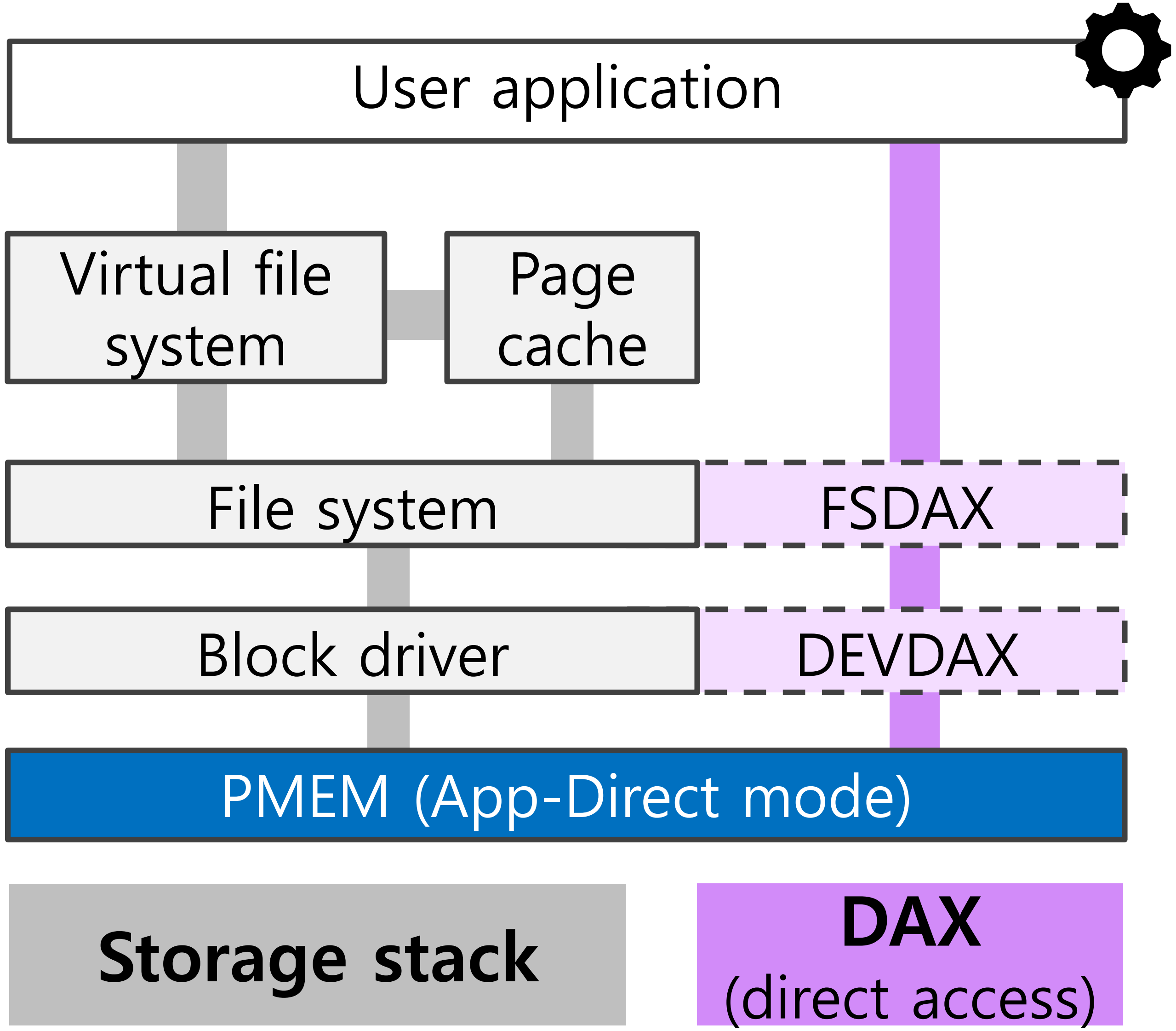
App-Direct Mode



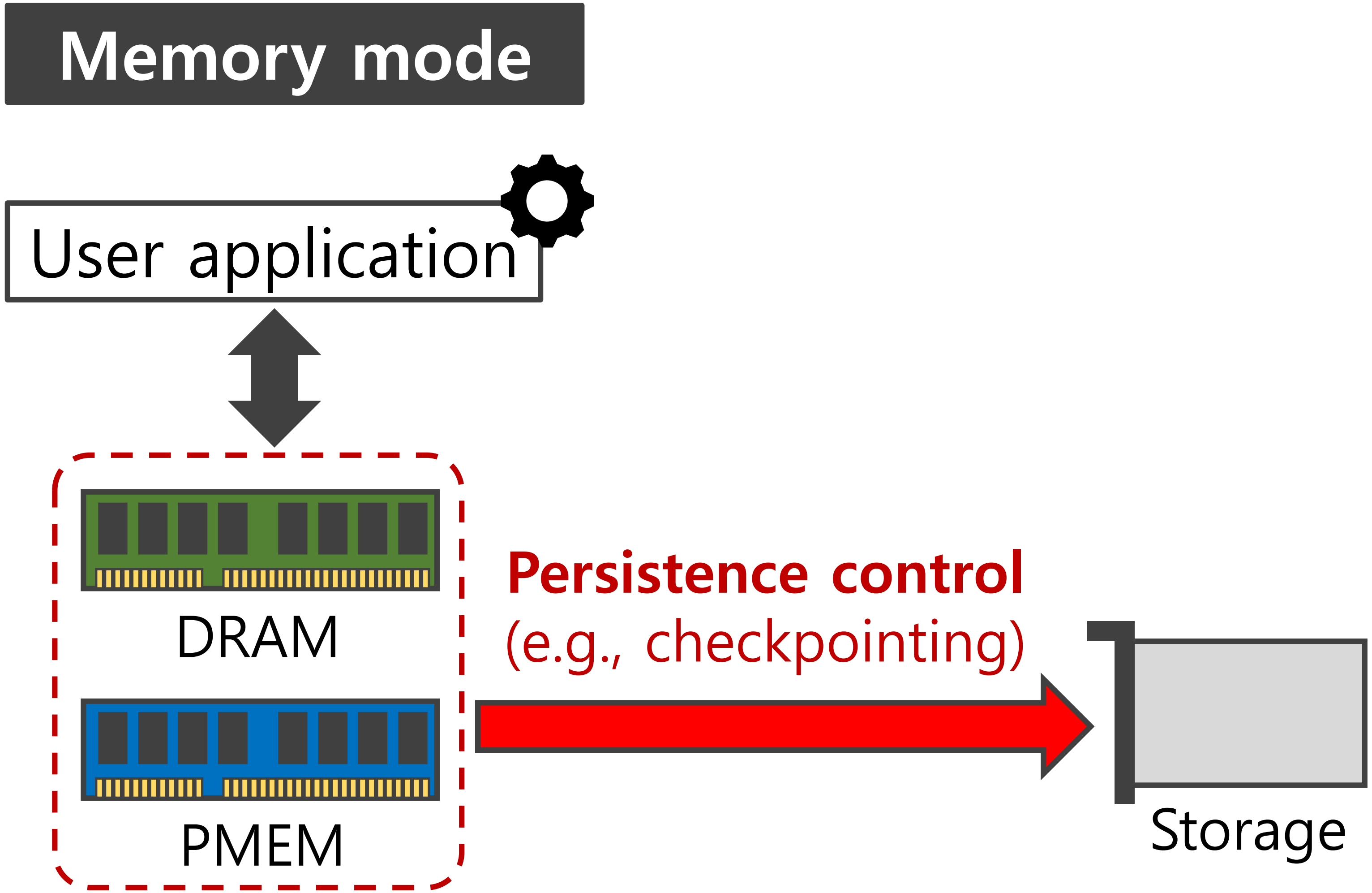
App-Direct Mode



- ➢ **3.70x slower than DRAM**



Memory Mode



Memory mode

User application

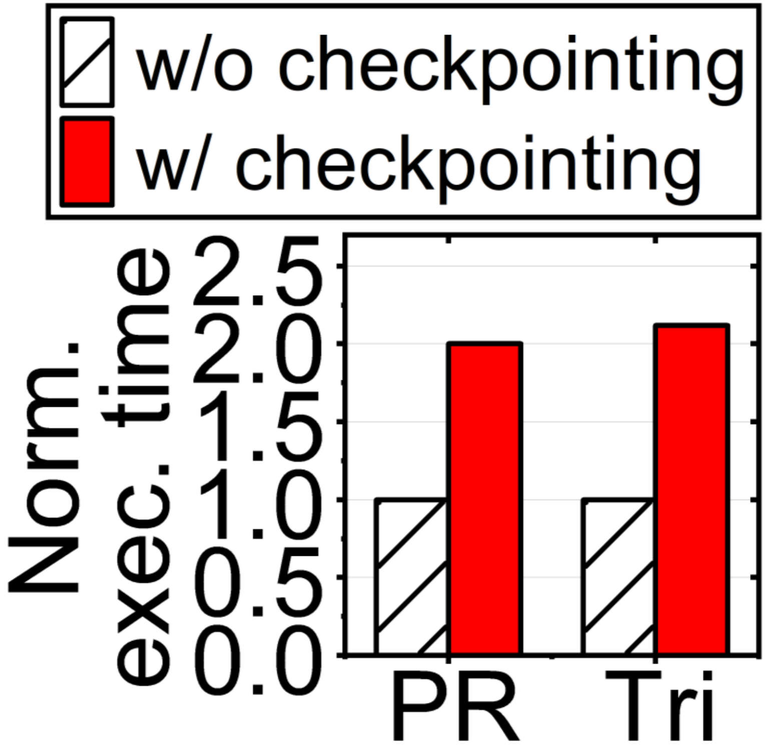
DRAM

PMEM

Persistence control

(e.g., checkpointing)

Storage



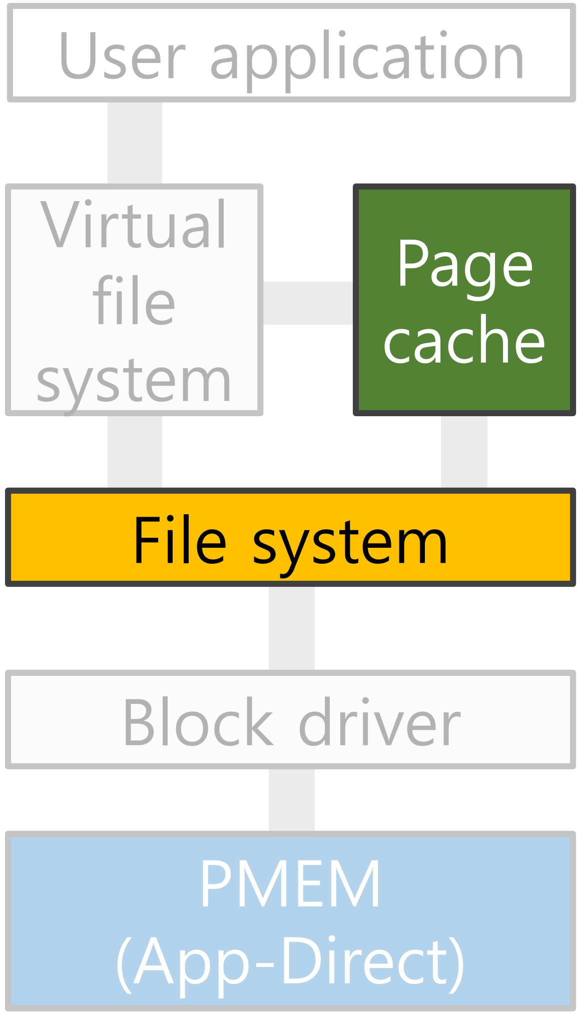
➢ **Becomes 2.06x slower**

What Is the Best Solution?

1. Maximize the **performance** of in-memory graph systems

2. Minize the overhead imposed by **data persistence control**

Back to the basics: “Employ a slow **storage stack on PMEM**”

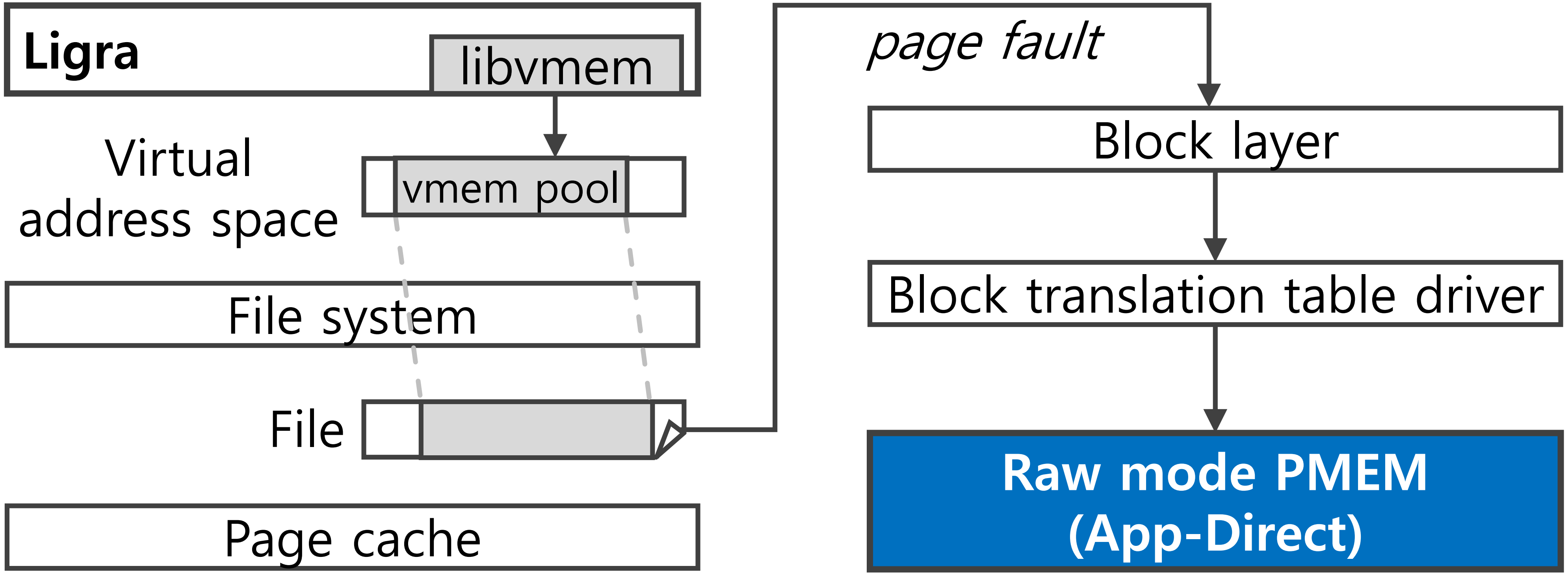


Natually reaps the benefits of DRAM caching

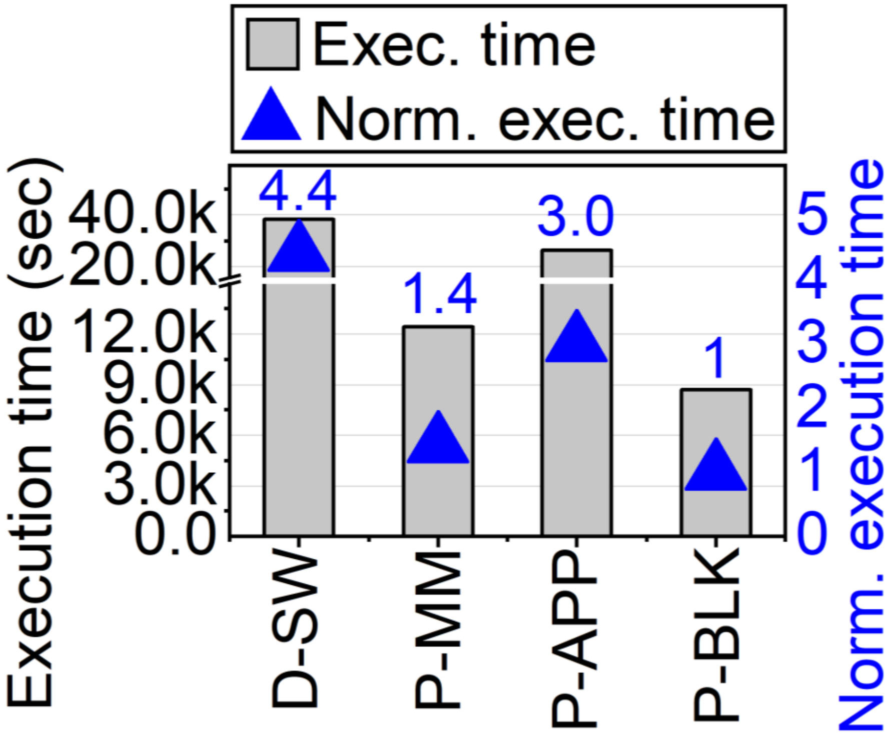
Guarantees
- Atomicity
- Consistency
- Isolation
- Durability

Modification of an In-Memory Graph Framework

- Modified Ligra (in-memory graph framework) to make it utilize the merits of storage stack



What Is the Best Solution?

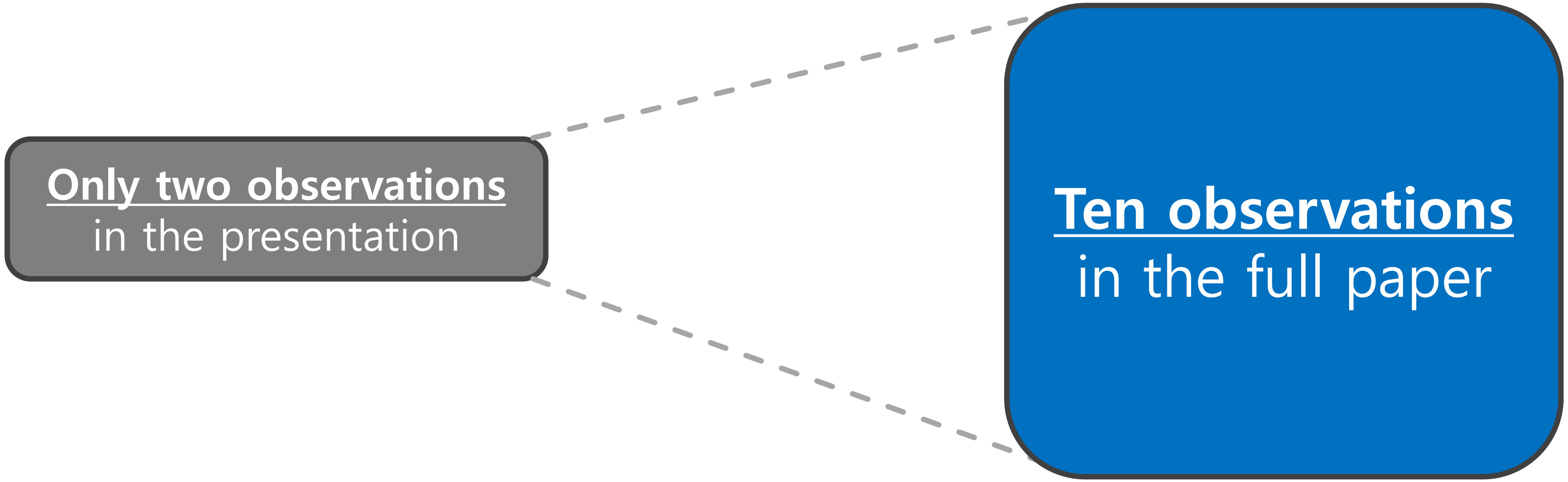


- D-SW: DRAM + NVMe SSD (swap)
- P-MM: PMEM in the memory mode
- P-APP: PMEM in the app-direct mode + DAX
- **P-BLK: PMEM in the app-direct mode + storage stack**

Observation #2: **Storage stack** could be the best solution

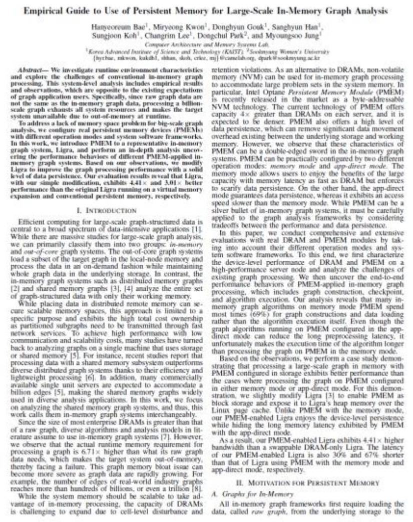
Conclusion

- Comprehensive and extensive **evaluation with real PMEM devices** to reveal the characteristics and challenges of in-memory graph processing
- **Modified Ligra** to utilize the benefits of the storage stack
  - **4.41x** better performance than the Ligra running on a **virtual memory expansion**
  - **3.01x** better performance than the Ligra running on a **conventional persistent memory**



Only two observations in the presentation

Ten observations in the full paper



Empirical Guide to Use of Persistent Memory for Large-Scale In-Memory Graph Analysis

Hanyeoreum Bae[1], Miryeong Kwon[1], Donghyun Gouk[1], Sanghyun Han[1],
Sungjoon Koh[1], Changrim Lee[1], Dongchul Park[2], and Myoungsoo Jung[1]
Computer Architecture and Memory Systems Lab.
[1]*Korea Advanced Institute of Science and Technology (KAIST),* [2]*Sookmyung Women's University*
{hycbae, mkwon, kukdh1, shhan, skoh, crlee, mj}@camelab.org, dpark@sookmyung.ac.kr

***Abstract*— We investigate runtime environment characteristics and explore the challenges of conventional in-memory graph processing. This system-level analysis includes empirical results and observations, which are opposite to the existing expectations of graph application users. Specifically, since raw graph data are not the same as the in-memory graph data, processing a billion-scale graph exhausts all system resources and makes the target system unavailable due to out-of-memory at runtime.**

To address a lack of memory space problem for big-scale graph analysis, we configure real persistent memory devices (PMEMs) with different operation modes and system software frameworks. In this work, we introduce PMEM to a representative in-memory graph system, Ligra, and perform an in-depth analysis uncovering the performance behaviors of different PMEM-applied in-memory graph systems. Based on our observations, we modify Ligra to improve the graph processing performance with a solid level of data persistence. Our evaluation results reveal that Ligra, with our simple modification, exhibits 4.41× and 3.01× better performance than the original Ligra running on a virtual memory expansion and conventional persistent memory, respectively.

I. Introduction

Efficient computing for large-scale graph-structured data is central to a broad spectrum of data-intensive applications [1]. While there are massive studies for large-scale graph analysis, we can primarily classify them into two groups: *in-memory* and *out-of-core* graph systems. The out-of-core graph systems load a subset of the target graph in the local-node memory and process the data in an on-demand fashion while maintaining whole graph data in the underlying storage. In contrast, the in-memory graph systems such as distributed memory graphs [2] and shared memory graphs [3], [4] analyze the entire set of graph-structured data with only their working memory.

While placing data in distributed remote memory can secure scalable memory spaces, this approach is limited to a specific purpose and exhibits the high total cost ownership as partitioned subgraphs need to be transmitted through fast network services. To achieve high performance with low communication and scalability costs, many studies have turned back to analyzing graphs on a single machine that uses storage or shared memory [5]. For instance, recent studies report that processing data with a shared memory subsystem outperforms diverse distributed graph systems thanks to their efficiency and lightweight processing [6]. In addition, many commercially available single unit servers are expected to accommodate a billion edges [5], making the shared memory graphs widely used in diverse analysis applications. In this work, we focus on analyzing the shared memory graph systems, and thus, this work calls them in-memory graph systems interchangeably.

Since the size of most enterprise DRAMs is greater than that of a raw graph, diverse algorithms and analysis models in literature assume to use in-memory graph systems [7]. However, we observe that the actual runtime memory requirement for processing a graph is 6.71× higher than what its raw graph data needs, which makes the target system out-of-memory, thereby facing a failure. This graph memory bloat issue can become more severe as graph data are rapidly growing. For example, the number of edges of real-world industry graphs reaches more than hundreds of billions, or even a trillion [8].

While the system memory should be scalable to take advantage of in-memory processing, the capacity of DRAMs is challenging to expand due to cell-level disturbance and retention violations. As an alternative to DRAMs, non-volatile memory (NVM) can be used for in-memory graph processing to accommodate large problem sets in the system memory. In particular, Intel Optane *Persistent Memory Module* (PMEM) is recently released in the market as a byte-addressable NVM technology. The current technology of PMEM offers capacity 4× greater than DRAMs on each server, and it is expected to be denser. PMEM also offers a high level of data persistence, which can remove significant data movement overhead existing between the underlying storage and working memory. However, we observe that these characteristics of PMEM can be a double-edged sword in the in-memory graph systems. PMEM can be practically configured by two different operation modes: *memory mode* and *app-direct mode*. The memory mode allows users to enjoy the benefits of the large capacity with memory latency as fast as DRAM but enforces to scarify data persistence. On the other hand, the app-direct mode guarantees data persistence, whereas it exhibits an access speed slower than the memory mode. While PMEM can be a silver bullet of in-memory graph systems, it must be carefully applied to the graph analysis frameworks by considering tradeoffs between the performance and data persistence.

In this paper, we conduct comprehensive and extensive evaluations with real DRAM and PMEM modules by taking into account their different operation modes and system software frameworks. To this end, we first characterize the device-level performance of DRAM and PMEM on a high-performance server node and analyze the challenges of existing graph processing. We then uncover the end-to-end performance behaviors of PMEM-applied in-memory graph processing, which includes graph construction, checkpoint, and algorithm execution. Our analysis reveals that many in-memory graph algorithms on memory mode PMEM spend most times (69%) for graph constructions and data loading rather than the algorithm execution itself. Even though the graph algorithms running on PMEM configured in the app-direct mode can reduce the long preprocessing latency, it unfortunately makes the execution time of the algorithm longer than processing the graph on PMEM in the memory mode.

Based on the observations, we perform a case study demonstrating that processing a large-scale graph in memory with PMEM configured in storage exhibits better performance than the cases where processing the graph on PMEM configured in either memory mode or app-direct mode. For this demonstration, we slightly modify Ligra [3] to enable PMEM as block storage and expose it to Ligra's heap memory over the Linux page cache. Unlike PMEM with the memory mode, our PMEM-enabled Ligra enjoys the device-level persistence while hiding the long memory latency exhibited by PMEM with the app-direct mode.

As a result, our PMEM-enabled Ligra exhibits 4.41× higher bandwidth than a swappable DRAM-only Ligra. The latency of our PMEM-enabled Ligra is also 30% and 67% shorter than that of Ligra using PMEM with the memory mode and app-direct mode, respectively.

II. Motivation for Persistent Memory

A. Graphs for In-Memory

All in-memory graph frameworks first require loading the data, called *raw graph*, from the underlying storage to the

NVMW ’22

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

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

KAIST[1], Sookmyung Women’s University[2]

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMELab