Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories

Chenxi Wang
University of California, Los Angeles
USA

Guoqing Harry Xu
University of California, Los Angeles
USA

1 Overview

To process real-world datasets, modern data-parallel systems often require extremely large amounts of memory, which are both costly and energy-inefficient. Emerging non-volatile memory (NVM) technologies offer high capacity compared to DRAM and low energy compared to SSDs. Hence, NVMs have the potential to fundamentally change the dichotomy between DRAM and durable storage in Big Data processing. However, most Big Data applications are written in *managed languages* and executed on top of a *managed runtime* that already performs various dimensions of memory management. Supporting hybrid physical memories adds in a new dimension, creating unique challenges in data replacement.

This paper proposes Panthera, a *semantics-aware, fully automated* memory management technique for Big Data processing over hybrid memories. Panthera analyzes user programs on a Big Data system to infer their coarse-grained access patterns, which are then passed to the Panthera runtime for efficient data placement and migration.

For Big Data applications, the coarse-grained data division information is accurate enough to guide the GC for data layout, which hardly incurs overhead in data monitoring and moving. We implemented Panthera [7] in OpenJDK and Apache Spark. Our extensive evaluation demonstrates that Panthera reduces energy by **32 – 52%** at only a **1 – 9%** time overhead.

1.1 Problems

Although using NVM for Big Data systems is a promising direction, the idea has not yet been fully explored. Although techniques such as Espresso [8] and Write Rationing [2] support NVM for managed programs, neither of them was designed for Big Data processing whose data usage is greatly different than that of regular, non-data-intensive Java applications [4, 5]. Adding NVM naïvely would lead to large performance penalties due to its significantly increased access latency and reduced bandwidth — *e.g.*, the latency of an NVM read is 2-4× larger than that of a DRAM read and NVM's bandwidth is about 1/8 - 1/3 of that of DRAM [3, 6] . Hence, a critical research question that centers around all hybrid-memory-related research is *how to perform intelligent data allocation and migration between DRAM and NVM so that we can maximize the overall energy efficiency while minimizing the performance overhead?* To answer this question in the context of Big Data processing, there are two major challenges.

Challenge #1: Working with Garbage Collection (GC). A common approach to managing hybrid memories is to modify the OS or hardware to (1) monitor access frequency of physical memory pages, and (2) move the hot (frequently-accessed) data into DRAM. This approach works well for native language applications where data stays in the memory location it is allocated into. However, in *managed languages*, the garbage collector keeps changing the data layout in memory by copying objects to different physical memory pages, which breaks the bonding between data and physical memory address. Most Big Data systems are written in such managed languages, e.g., Java and Scala, for the quick development cycle and rich community support they provide. Managed languages are executed on top of a managed runtime such as the JVM, which employs a set of sophisticated memory management techniques such as garbage collection. As a traditional garbage collector is not aware of hybrid memories, allocating and migrating hot/cold pages at the OS level can easily lead to interference between these two different levels of memory management.

Challenge #2: Working with Application-level Memory Subsystems. Modern Big Data systems all contain sophisticated memory subsystems that perform various memory management tasks *at the application level*. For instance, Apache Spark [1] uses resilient distributed datasets (RDDs) as its data abstraction. An RDD is a distributed data structure that is partitioned across different servers. At a low level, each RDD partition is an array of Java objects, each representing a data tuple. RDDs are often immutable but can exhibit diverse lifetime behavior. For example, developers can explicitly persist RDDs in memory for memoization or fault tolerance. Such RDDs are long-lived while RDDs storing intermediate results are short-lived.

An RDD can be at one of many storage levels. Spark further allows developers to specify, with annotations, where an RDD should be allocated, *e.g.*, in the managed heap or native memory. Objects allocated natively are not subject to GC, leading to increased efficiency. However, data processing tasks, such as `shuffle`, `join`, `map`, or `reduce`, are performed over the managed heap. A native-memory-based RDD cannot be directly processed unless it is first moved into the heap. Hence, where to allocate an RDD depends on when and how it is processed. Clearly, efficiently using

hybrid memories requires appropriate coordination between these orthogonal data placement polices, *i.e.*,, the heap, native memory, or disk, vs. NVM or DRAM.

1.2 Our Contributions

Our Insight. Big Data applications have two unique characteristics that can greatly aid hybrid memory management. First, they perform bulk object creation, and data objects exhibit strong *epochal behavior and clear access patterns*. For example, Spark developers program with RDDs, each of which contains objects with exactly the same access/lifetime patterns. Exploiting these patterns at the runtime would make it much easier for Big Data applications to enjoy the benefits of hybrid memory.

Second, the data access and lifetime patterns are often *statically* observable in the *user program*. For example, an RDD is a coarse-grained data abstraction in Spark and the access patterns of different RDDs can often be inferred from the way they are created and used in the program.

Hence, unlike regular, non-data-intensive applications for which profiling is often needed to understand the access patterns of individual objects, we can develop a simple static analysis for a Big Data application to infer the access pattern of each coarse-grained data collection, in which all objects share the same pattern. The static analysis does not incur any runtime overhead, yet it can produce precise enough data access information for the runtime system to perform effective allocation and migration.

Panthera [7] Based on our extensive experience with Big Data applications, we propose Panthera, which divides a mess of data objects into several data collections according to application's semantics and infers the coarse-grained data usage behavior by light-weight static program analysis and dynamic data usage monitoring. Panthera leverages garbage collection to migrate data between DRAM and NVM, incurring almost no runtime overhead.

Panthera enhances both the JVM and Spark with two major innovations. First, based on the observation that access patterns in a Big Data application can be identified statically, we develop a static analysis that analyzes a Spark program to infer a memory tag (*i.e.*,, NVM or DRAM) for each RDD variable based on the variable's location and the way it is used in the program. These tags indicate which memory the RDDs should be allocated in.

Second, we develop a new semantics-aware and physical-memory-aware generational GC. Our static analysis instruments the Spark program to pass the inferred memory tags down to the runtime system, which uses these tags to make allocation/migration decisions. Since our GC is based on a high-performance generational GC in OpenJDK, Panthera's heap has two spaces, representing a young and an old generation. We place the entire young generation in DRAM while splitting the old generation into a small DRAM component and a large NVM component. The insight driving this design is based on a set of key observations we make over the lifetimes and access patterns of RDDs in representative Spark executions:

- Most objects are allocated initially in the young generation. Since they are frequently accessed during initialization, placing them in DRAM enables fast access to them.
- Long-lived objects in Spark can be roughly classified into two categories: (1) long-lived RDDs that are frequently accessed during data transformation. They should be placed in DRAM. (2) long-lived RDDs that are cached primarily for fault tolerance. They should be placed in NVM.
- There are also short-lived RDDs that store temporary, intermediate results. These RDDs die and are then reclaimed in the young generation quickly, leading to frequent accesses to this area. This is another reason why we place the young generation within DRAM.

Based on these observations, we modified both the minor and major GC, which allocate and migrate data objects, based on their RDD types and the semantic information inferred by our static analysis, into the spaces that best fit their lifetimes and access patterns. Our runtime system also monitors the transformations invoked over RDD objects to perform runtime (re)assessment of RDDs' access patterns. Even if the static analysis does not accurately predict an RDD's access pattern and the RDD gets allocated in an undesirable space, Panthera can still migrate the RDD from one space to another using the major GC.

References

[1] Apache spark^{TM}. https://spark.apache.org, 2019.

[2] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. In *PLDI '18*, pages 62–77, 2018.

[3] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *EuroSys '16*, pages 15:1–15:16, 2016.

[4] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *OSDI '16*, pages 349–365, 2016.

[5] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS '15*, pages 675–690, 2015.

[6] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15)*, pages 37–49, 2015.

[7] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *PLDI '19*, pages 347–362, 2019.

[8] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility. In *ASPLOS '18*, pages 70–83, 2018.