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

Chenxi Wang
University of California, Los Angeles
USA

Huimin Cui
SKL Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences, China

Ting Cao
Microsoft Research
China

John Zigman
University of Sydney, Australia

Haris Volos
Google, USA

Onur Mutlu
ETH Zürich, Switzerland

Fang Lv
SKL Computer Architecture, ICT, CAS, China

Xiaobing Feng
SKL Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences, China

Guoqing Harry Xu
University of California, Los Angeles
USA

Abstract

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 in OpenJDK and Apache Spark. Our extensive evaluation demonstrates that Panthera reduces energy by **32 – 52%** at only a **1 – 9%** time overhead.

CCS Concepts • **Information systems → Data management systems;** • **Hardware → Non-volatile memory;** • **Software and its engineering → Memory management.**

Keywords hybrid memories, Big Data systems, memory management, garbage collection

ACM Reference Format:
Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314650

1 Introduction

Modern Big Data computing exemplified by systems such as Spark and Hadoop is extremely memory-intensive. Lack of memory can lead to a range of severe functional and performance issues including out-of-memory crashes, significantly degraded efficiency, or even loss of data upon node failures. Relying completely on DRAM to satisfy the memory need of a data center is costly in many different ways — *e.g.*, large-volume DRAM is expensive and energy-inefficient; furthermore, DRAM's relatively small capacity dictates that a large number of machines is often needed just to provide sufficient memory, resulting in underutilized CPU resources for workloads that cannot be easily parallelized.

Emerging non-volatile memory (NVM), such as phase change memory (PCM) [32, 55, 58], resistive random-access memory (RRAM) [54], Spin-transfer torque memory (STT-MRAM) [30] or 3D XPoint [4], is a promising technology that, compared to traditional DRAM, provides higher capacity and lower energy consumption. Systems with hybrid memories have therefore received much attention [9–11, 13, 16, 23, 31, 32, 35, 38, 40, 41, 46–48, 52, 53, 56, 59, 62, 63] recently from both academia and industry. The benefit of mixing NVM with DRAM for Big Data systems is obvious — NVM's high capacity makes it possible to fulfill the high memory requirement of a Big Data workload with a small number of compute nodes, holding the promise of significantly reducing the costs of both hardware and energy in large data centers.

1.1 Problems

Although using NVM for Big Data systems is a promising direction, the idea has not yet been fully explored. 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 [19, 51] . 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 [5] 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 (*e.g.*, memory, disk, unmaterialized, *etc.*). 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. For example, a frequently-accessed RDD should be placed in DRAM while a native-memory-based RDD would not be frequently used and placing it in NVM would be desirable. 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.

State of the Art. In summary, the key challenges in supporting hybrid memories for Big Data processing lie in how to develop runtime system techniques that can make memory allocation/migration decisions that match how data is actually used in an application. Although techniques such as Espresso [56] and Write Rationing [9] 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 [42, 43].

For example, Espresso defines a new programming model that can be used by the developer to allocate objects in persistent memory. However, real-world developers would be reluctant to completely re-implement their systems from scratch using such a new model. Shoaib *et al.* [9] introduced the Write Rationing GC, which moves the objects that experience a large/small number of writes into DRAM/NVM to prolong NVM's lifetime. Write Rationing pioneers the work of using the GC to migrate objects based on their access patterns. However, Big Data systems make heavy use of immutable datasets — for example, in Spark, most RDDs are immutable. Placing all immutable RDDs into NVM can incur a large overhead as many of these RDDs are frequently read and an NVM read is 2–4x slower than a DRAM read.

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 (§2).

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. This observation aligns well with prior work (*e.g.*, Facade [43] or Yak [42]) that requires simple annotations to specify epochs to perform efficient garbage collection for Big Data systems. 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. 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.

We focus on Apache Spark in this paper as it is the de-facto data-parallel framework deployed widely in industry. Spark hosts a range of applications in machine learning, graph analytics, stream processing, *etc.*, making it worthwhile to build a specialized runtime system, which can provide immediate benefit to all applications running atop. Although Panthera was built for Spark, our idea is applicable to other systems such as Hadoop as well; §4 provides a detailed discussion of Panthera's applicability.

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 (§3) 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 (§4). 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 (discussed in §2 in detail) 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 (*e.g.*, cached for iterative algorithms) and (2) long-lived RDDs that are cached primarily for fault tolerance. The first category of RDDs should be placed in the DRAM component of the old generation because they have long lifespans and DRAM provides desirable performance for frequent access to them. The second category should be placed in the NVM component of the old generation because they are infrequently accessed and hence NVM's large access latency has relatively small impact on overall performance.
- 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.

Results. We have evaluated Panthera extensively with graph computing (GraphX), machine learning (MLlib) and other iterative in-memory computing applications (Table 4). Results with various heap sizes and DRAM ratios demonstrate that Panthera makes effective use of hybrid memories — overall, the Panthera-enhanced JVM reduces the memory energy by **32%–52%** with only a **1%–9%** execution time overhead, whereas Write Rationing [9] that moves read-only RDD objects into NVM incurs a **41%** time overhead.

2 Background and Motivation

This section provides necessary background on Apache Spark [5] and a motivating example that illustrates the access patterns in a Spark program.

Spark Basics. Spark is a data-parallel system that supports acyclic data flow and in-memory computing. The major data representation used by Spark is *resilient distributed dataset* (RDD) [60], which represents a read-only collection of tuples. An RDD is a distributed memory abstraction partitioned in the cluster. Each partition is an array of data items of the

```
Top:obj  org/apache/spark/rdd/ShuffledRDD
 depth[0]:array ,  [Lscala/Tuple2;
  depth[1]:obj  scala/Tuple2
    depth[2]:obj  java/lang/String
      depth[3]:array  [C
    depth[2]:obj  spark/util/collection/CompactBuffer
      depth[3]:array ,  [Ljava/lang/String;
        depth[4]:obj  java/lang/String
          depth[5]:array  [C
        depth[4]:obj  java/lang/String
          depth[5]:array  [C
```

Figure 1. The heap structure of an example RDD.

same type. Each node maintains an RDD partition, which is essentially a multi-layer Java data structure — a top RDD object references a Java array, which, in turn, references a set of tuple objects such as key-value pairs. Figure 1 shows the heap structure for an example RDD where each element is a pair of a string (key) and a compact buffer (value).

A Spark pipeline consists of a sequence of *transformations* and *actions* over RDDs. A transformation produces a new RDD from a set of existing RDDs; examples are *map*, *reduce*, or *join*. An action is a function that computes statistics from an RDD, such as an aggregation. Spark leverages *lazy evaluation* for efficiency, that is, a transformation may not be evaluated until an action is performed later on the resulting RDD. Before data processing starts, the dependences between RDDs are first extracted from the transformations to form a *lineage graph*, which can be used to conduct lazy evaluation and RDD recomputation upon node failures.

With lazy evaluation, a transformation only creates a (top-level) RDD object without *materializing* the RDD (*i.e.*, the point at which its internal array and actual data tuples are created). Recomputing all RDDs is time-consuming when the lineage is long or when it branches out, and hence, Spark allows developers to cache certain RDDs in memory (by using the API `persist`) . Developers can specify a storage level for a persisted RDD, *e.g.*, in memory or on disk, in the serialized or deserialized form, *etc.* RDDs that are *not* explicitly persisted are temporary RDDs that will be garbage-collected when they are no longer used, while persisted RDDs are materialized and never collected.

The Spark scheduler examines the lineage graph to build a DAG of stages for execution. The lineage (transformation)-based dependences are classified into "narrow" and "wide". A narrow dependence exists from a parent to a child RDD if each partition of the parent is used by *at most one* partition of the child RDD. By contrast, a wide dependence exists when each partition of the parent RDD may be used by *multiple* child partitions. Distinguishing these two types of dependences makes it possible for Spark to determine whether a shuffle is necessary. For example, for narrow dependences shuffling is not necessary, while for wide dependences it is.

A Spark pipeline is split into a set of *stages* based on shuffles (and thus wide dependences) — each stage ends at a shuffle that writes RDDs onto the disk and the next stage starts by reading data from disk files. Transformations that exhibit narrow dependences are grouped into the same stage and executed in parallel.

RDD Characteristics. An RDD is, at a low level, an array of Java objects, which are managed by the semantics-agnostic GC in the JVM. RDDs often exhibit predictable lifetime and memory-access patterns. Our goal is to pass these patterns down to the GC, which can exploit such semantic information for efficient data placement. We provide a concrete example to illustrate these patterns.

Figure 2(a) shows the Spark program for PageRank [14], which is a well-known graph algorithm used widely by search engines to rank web pages. The program iteratively computes the rank of each vertex based on the contributions of its in-neighbors. Three RDDs can be seen from its source code: `links` representing edges from the input graph, `contribs` containing contributions from incoming edges of each vertex, and `ranks` that maps each vertex to its page rank. `links` is a static map computed from the input while `contribs` and `ranks` are recomputed per iteration of the loop.

In addition to these three developer-defined RDDs visible in the program, Spark generates many invisible RDDs to store intermediate results during execution. A special type of intermediate RDD is `ShuffledRDD`. Each iteration of the loop in the example forms a stage that ends at a shuffle, writing shuffled data into different disk files. In the beginning of the next stage, Spark creates a `ShuffledRDD` as input for the stage. Unlike other intermediate RDDs that are never materialized, `ShuffledRDDs` are immediately materialized because they contain data read freshly out of disk files. However, since they are not persisted, they will be collected when the stage is completed.

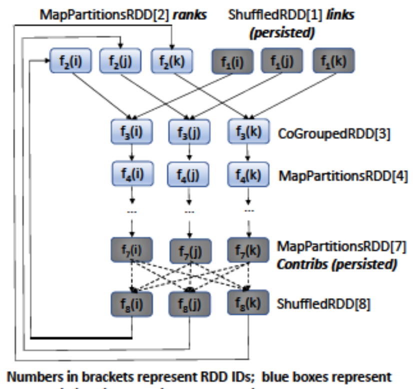
In summary, (1) persisted RDDs are materialized at the moment the method `persist` is called, and (2) non-persisted RDDs are not materialized unless they are `ShuffleRDDs` or an action is invoked on them.

Example. Figure 2(b) shows the set of RDDs that exists within a stage (*i.e.*, iteration) and their dependences. Suppose each RDD has three partitions (on three nodes). The dashed edges represent wide dependences (*i.e.*, shuffles) due to the reduction on Line 17. There are totally eight RDDs generated in each iteration. `ShuffledRDD[8]`, which stems from the reduction on Line 17, is transformed to `ranks` via a `map` transformation. `ranks` joins with `links` to form `CoGroupedRDD[3]`, which is then processed by four consecutive `map` functions, *i.e.*, $f_4 - f_7$, producing `contribs` at the end. For unmaterialized (blue) RDDs, the sequence of transformations (*e.g.*, $f_4 \circ \ldots \circ f_7$) is applied to each record from the source RDD in a *streaming* manner via iterators to produce a final record.

For `links` and `contribs`, the developer invokes the method `persist` to materialize these RDDs. The storage levels indicate that `links` is cached in memory throughout the execution (as it is used in each iteration) while `contribs` generated in each iteration is kept in memory but will be serialized to

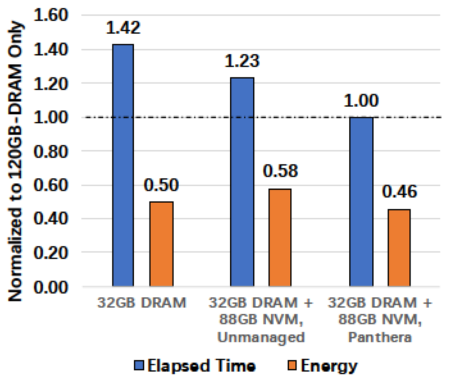
```
1  var lines = ctx.textFile(args[0],
       slices)
2  var links = lines.map{s=>
3    var parts = s.split("\\s+")
4    (parts(0), parts(1))
5  }.distinct().groupByKey()
6  .persist(StorageLevel.MEMORY_ONLY)
7
8  var ranks = links.mapValues(v => 1.0)
9  for(i <- 1 to iters){
10   var contribs =
         links.join(ranks).values.flatMap{
11     case(urls, rank) =>
12       val size = urls.size
13       urls.map(url=>(url, rank/size))
14       .persist(StorageLevel
15                 .MEMORY_AND_DISK_SER)
16   }
17   ranks = contribs.reduceByKey(_ + _).
18     mapValues(0.15 + 0.85 * _)
19 }
20 ranks.count()
```

(a) PageRank Program.


MapPartitionsRDD[2] *ranks*
ShuffledRDD[1] *links* *(persisted)*
$f_2(i)$ $f_2(j)$ $f_2(k)$ $f_1(i)$ $f_1(j)$ $f_1(k)$
$f_3(i)$ $f_3(j)$ $f_3(k)$ CoGroupedRDD[3]
$f_4(i)$ $f_4(j)$ $f_4(k)$ MapPartitionsRDD[4]
$f_7(i)$ $f_7(j)$ $f_7(k)$ MapPartitionsRDD[7] *Contribs (persisted)*
$f_8(i)$ $f_8(j)$ $f_8(k)$ ShuffledRDD[8]

Numbers in brackets represent RDD IDs; blue boxes represent unmaterialized intermediate results while black boxes represent materialized RDDs. Solid and dashed edges represent narrow and wide dependences, respectively.

(b) Transformations within a stage



(c) Results of DRAM-only and DRAM+NVM, managed by the OS and by Panthera.

Figure 2. Characteristics of RDDs in Spark PageRank.

disk upon memory pressure. ranks is not explicitly persisted. Hence, it is not materialized until the execution reaches Line 20 where action count is invoked on the RDD object.

The lifetime patterns of these different RDDs fall into two categories. Non-persisted intermediate RDDs are short-lived as their data objects are generated only during a pipelined execution. Persisted RDDs are long-lived and stay in memory/on disk until the end of the execution. Their access patterns are, however, more diverse. Objects in an intermediate RDD are accessed at most once during streaming. Objects in a persisted RDD can exhibit different types of behavior. For RDDs like `links` that are used in each iteration, their objects are frequently accessed. In contrast, RDDs like `contribs` are persisted primarily for speeding up recovery from faults, and hence, their objects are rarely used after generated.

Design Choices. The different characteristics of DRAM and NVM make them suitable for different types of datasets. DRAM has low capacity and fast access speed, while NVM has large capacity but slow speed. Hence, DRAM is a good choice for storing small-sized, frequently-accessed datasets, while large-sized, infrequently-accessed datasets fit naturally into NVM. The clear distinction in the lifespans and access patterns of different RDDs makes it easy for them to be placed into different memories suitable for their behavior. For example, intermediate (blue) RDDs are never materialized. Their objects are created individually during streaming and then quickly collected by the GC. These objects are allocated in the young generation and will eventually die there. Although they are short-lived, they are accessed frequently during streaming. This motivates our design choice of placing the young generation in DRAM.

Persisted RDDs, in contrast, have all their data objects created at the same time, and thus need large storage space. Since they are kept alive *indefinitely*, they should be allocated directly in the old generation. One category of persisted RDDs includes those that are frequently accessed, like `links`; they need to be placed in DRAM. Another category includes RDDs that are rarely accessed and cached for fault tolerance, like `contribs`, these RDDs should be placed in NVM. This behavioral difference motivates our choice of splitting the old generation into a DRAM and an NVM component.

We perform what we suggest on a system with 32GB DRAM and 88GB NVM using Spark-based PageRank as the benchmark. Figure 2(c) shows the performance and energy consumption normalized to a system with 120GB of DRAM. Compared to using only 32GB DRAM, adding 88GB NVM to the system provides modest performance benefit (15%) but leads to 16% higher energy consumption, without proper data placement across DRAM and NVM (see *Unmanaged*, §5.2). After applying Panthera, RDD `links` and `contribs` are placed into DRAM and NVM, respectively. With such careful placement of data across DRAM and NVM, we find that (1) performance increases by 42% compared to using only a 32GB DRAM, and becomes at the same level of the performance of using 120GB DRAM; (2) energy consumption is 9% less than using only a 32GB DRAM, and 54% less than using a 120GB DRAM. We conclude that careful data placement between DRAM and NVM can provide the performance of large DRAM system, while keeping the energy consumption at the level of a small DRAM system.

3 Static Inference of Memory Tags

Based on our observation that the access patterns of RDDs can often be identified from the program using them, we developed a simple static analysis that extracts necessary semantic information for efficient data placement. The analysis automatically infers, for each persisted RDD visible in the program, whether it should be allocated in DRAM or

NVM. This information is then passed down to the runtime system for appropriate data allocation.

Static Analysis. In a Spark program, the developer can invoke `persist` with a particular storage level on an RDD to materialize the RDD, as illustrated in Figure 2. We piggyback on the storage levels to further determine if a persisted RDD should be placed into DRAM or NVM. In particular, Panthera statically analyzes the program to infer a memory tag (*i.e.*, DRAM or NVM) for each `persist` call. Each of the ten existing storage levels (*e.g.*, `MEMORY_ONLY`), except for `OFF_HEAP` and `DISK_ONLY`, is expanded into two sub-levels, annotated with NVM and DRAM, respectively (*e.g.*, `MEMORY_ONLY_DRAM` and `MEMORY_ONLY_NVM`). `OFF_HEAP` is translated directly into `OFF_HEAP_NVM` because RDDs placed in native memory are rarely used, while `DISK_ONLY` does not carry any memory tag.

Our static analysis performs inference based on the *def-use* information *w.r.t.* each RDD variable declared in the program as well as the loop(s) in which the variable is defined/used. Our key insight is that if the variable is *defined* in each iteration of a computational loop, most of the RDD instances represented by the variable are *not* used frequently. This is because Spark RDDs are often immutable and hence, every definition of the RDD variable creates a new RDD instance at run time, leaving the old RDD instance cached and unused. Hence, we tag the variable "NVM", instructing the runtime system to place these RDDs in NVM. An example is the `contribs` variable in Figure 2(a), which is defined in every iteration of the loop — although the variable is also used in each iteration, the use refers to the most recent RDD instance created in the last iteration while the instances created in all the other past iterations are left unused.

By contrast, if a variable is *used-only* (*i.e.*, never defined) in the loop, such as `links`, we create a tag "DRAM" for it since only one instance of the RDD exists and is repeatedly used. Panthera analyzes not only RDD variables on which `persist` is explicitly called, but also those on which actions are invoked, such as the `ranks` variable in Figure 2(a). The tag inferred for an RDD variable (say v) is passed, at the materialization point of every RDD instance (v refers to), into the runtime system via automatically instrumented calls to auxiliary (native) methods provided by the Panthera JVM. We piggyback on a tracing GC to propagate this tag from the RDD object down to each data object contained in the RDD — when the GC runs, it moves objects with the same tag together into the same (DRAM or NVM) region (see §4).

One constraint that needs to be additionally considered is the location of the loop relative to the location of the materialization point of the RDD. We analyze the loop only if the materialization point *precedes or is in* the loop. Otherwise, whether the variable is used or defined in the loop does not matter as the RDD has not been materialized yet. For instance, although the `ranks` variable is defined in the loop that starts at Line 17, it does not get materialized until Line 20 after the loop finishes. Hence, its behavior in the loop does not affect its memory tag, which should actually depend on its *def-use* in the loops, if any, after Line 20.

If no loop exists in a program, the program has only one iteration and all RDDs receive an "NVM" tag as none of them are repeatedly accessed. If there are multiple loops to be considered for an RDD variable, we tag it "DRAM" if there exists one loop in which the variable is used-only and that loop follows or contains the materialization point of the RDD. The variable receives an "NVM" tag otherwise. If all persisted RDDs receive an "NVM" tag at the end of the analysis, we change the tags of all RDDs to "DRAM" — the goal is to fully utilize DRAM by first placing RDDs in DRAM. Once DRAM capacity is exhausted, the remaining RDDs, including those with a "DRAM" tag, will be placed in NVM.

Note that our analysis infers tags only for the RDD variables explicitly declared in the program. Intermediate RDDs produced during execution are not materialized and thus do not receive memory tags from our analysis. We discuss how to handle them in §4.

The memory tag of an RDD variable is a *static approximation* of its access pattern, which may not reflect the behavior of all RDD instances represented by the variable at run time. However, user code for data processing often has a simple batch-transformation logic. Hence, the static information inferred from our analysis is often good enough to help the runtime make an accurate placement decision for the RDD. In case the statically inferred tags do not precisely capture the RDD's access information, Panthera has the ability to move RDDs between NVM and DRAM (within the old generation) based on their access frequencies, when a full-heap GC occurs. §4 provides a full discussion for this mechanism.

***Dealing with* `ShuffledRDD`.** Recall from §2 that, in addition to the RDDs on which `persist` is explicitly invoked, `ShuffledRDDs`, which are created from disk files after a shuffle, are also materialized. These RDDs are often the input of a stage but invisible in the program code. The challenge here is where to place them. Our insight is that their placement should depend on the other materialized RDDs that are transformed from (*i.e.*, depend on) them in the same stage.

For example, in Figure 2(b), the input of the stage are two sets of `ShuffledRDDs`: [1] and [8]. `ShuffledRDD[1]` is the RDD represented by `links` and our static analysis already infers tag "DRAM" for it. `ShuffledRDD[8]` results from the reduction in the previous stage. Because `ShuffledRDD[8]` transitively produces `MapPartitionRDD[7]` (represented by `contribs`) and `MapPartitionRDD[7]` has a memory tag "NVM" inferred by our static analysis, we tag `ShuffledRDD[8]` "NVM" as well.

The main reason is that RDDs belonging to the same stage may share many data objects for optimization purposes. For example, a `map` transformation that only changes the values (of key-value pairs) in RDD *A* may generate a new RDD *B* that references the same set of key objects as in *A*. If *B* has already received a memory tag from our static analysis, it is better to assign the same tag to *A* so that these shared objects

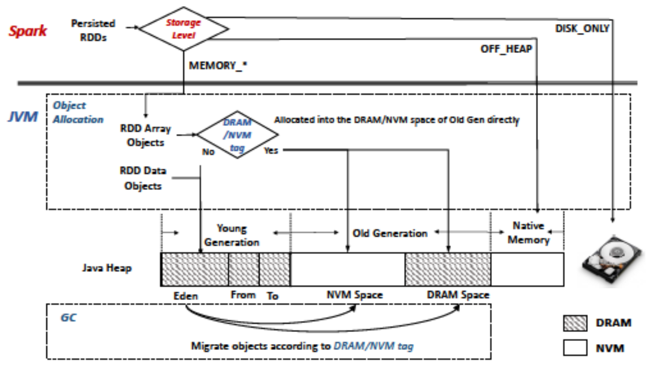


Figure 3. The Panthera heap and allocation policies. Here RDD array objects refer to RDDs' backbone arrays while data objects refer to other non-array objects in an RDD structure.

do not receive inconsistent tags and would not need to be moved from one memory to another when B is generated from A. This is especially beneficial when the transformation is in a computational loop — a large number of objects would be moved if A and B have different memory tags.

To assign the same tag to A and B, we add support that scans the lineage graph at the beginning of each stage to propagate the memory tag *backward*, starting from the lowest materialized RDD in the graph that has received a tag from our analysis. Conflicts may occur during the propagation — an RDD encountered during the backward traversal may have an existing tag that is different from the tag being propagated. To resolve conflicts, we define the following priority order: DRAM > NVM, which means that upon a conflict, the resulting tag is always DRAM. This is because our goal is to minimize the NVM-induced overhead; RDDs with a "DRAM" tag inferred will be frequently used and putting them in NVM would cause large performance degradation.

4 The Panthera Garbage Collector

While our static analysis (§3) determines where RDDs should be allocated, this information has to be communicated down to the runtime system, which recognizes only objects, not RDDs. Hence, our goal is to develop a new GC that, when placing/moving data objects, is aware of (1) the high-level semantics about where (DRAM or NVM) these RDDs should be placed and (2) the low-level information about the RDDs to which these objects belong.

We have implemented our new collection algorithm in OpenJDK 8 (build jdk8u76-b02) [7]. In particular, we have modified the object allocator, the interpreter, the two JIT compilers (C1 and Opto), and the Parallel Scavenge collector.

4.1 Design Overview

Heap Design. The Panthera GC is based on the Parallel Scavenge collector, which is the default GC in OpenJDK8. The collector divides the heap into a young and an old generation. As discussed earlier in §1, Panthera places the young generation in DRAM and splits the old generation into a DRAM component and an NVM component. The off-heap native memory is placed entirely in NVM. We reserve two unused bits, referred to as `MEMORY_BITS`, from the header of each object to indicate whether the object should be allocated into DRAM (01) or NVM (10). The default value for these bits is 00 — objects that do not receive a tag have this default value. They will be promoted to the NVM component of the old generation if they live long enough. Figure 3 illustrates the heap structure and our allocation policies.

Allocation Policies. As discussed in §3, each materialized RDD carries a memory tag that comes from our static analysis or lineage-based tag propagation. However, at a low level, an RDD is a structure of objects, as illustrated in Figure 1, and these objects are created at different points in the execution. Our goal is to place all objects belonging to the same logical RDD — including the top object, the array object, tuple objects, and other objects reachable from tuples — together in the space suggested by the RDD's memory tag, because these objects likely have the same access pattern and lifetime.

However, this is rather challenging — our static analysis infers a memory tag for each *top RDD object* (whose type is a subtype of `org.apache.spark.rdd.RDD`) in the user program and we do not know what other objects belong to this RDD by just analyzing the user program. Statically identifying what objects belong to a logical data structure would require precise context-sensitive static analysis of *both user and system code*, which is difficult to do due to Spark's extremely large codebase and the scalability issues of static analysis.

Our idea to solve this problem is that instead of attempting to allocate all objects of an RDD directly into the space (say S) suggested by the RDD's tag, we *allocate only the array object into S upon its creation*. This is much easier to do — Panthera instruments each materialization point (*e.g.*, before a call to `persist` or a Spark action) in the user program to pass the tag down to the runtime system without needing to analyze the Spark system code. Since the array is created at materialization, the runtime system can just use the tag to determine where to place it. All other objects in the RDD are not immediately allocated in S due to the aforementioned difficulties in finding their allocation sites. They are instead allocated in the young generation. Later, we use the GC to move these objects into S as tracing is performed.

Another important reason why we first allocate the array object into S is because the array is often much larger than the top and tuple objects. It is much more efficient to allocate it directly into the space it belongs to rather than allocating it somewhere else and moving it later.

Table 1 shows our allocation policies for different types of objects in an RDD. For RDDs with tag "DRAM", array objects are allocated directly into the DRAM component of the old generation if it has enough space. Otherwise, they have to be allocated in the NVM component. For RDDs with tag "NVM", array objects are allocated directly into the NVM component. Intermediate RDDs without tags are all allocated in the young generation (DRAM). Most of them end up dying there and never get promoted, while a small number of objects that eventually become old enough will be promoted to the NVM space of the old generation. Top RDD objects and data tuple objects, as discussed earlier, are all allocated into the young generation and moved later by the GC to the spaces containing their corresponding arrays.

Table 1. Panthera's allocation policies.

Tag	Obj Type	Initial Space	Final Space
DRAM	RDD Top	Young Gen.	DRAM of Old Gen.
	RDD Array	DRAM of Old Gen.	DRAM of Old Gen.
	Data Objs	Young Gen.	DRAM of Old Gen.
NVM	RDD Top	Young Gen.	NVM of Old Gen.
	RDD Array	NVM of Old Gen.	NVM of Old Gen.
	Data Objs	Young Gen.	NVM of Old Gen.
NONE	RDD Top	Young Gen.	Young Gen. or NVM of Old Gen.
	RDD Array	Young Gen.	Young Gen. or NVM of Old Gen.
	Data Objs	Young Gen.	Young Gen. or NVM of Old Gen.

4.2 Implementation and Optimization

This subsection describes our implementation techniques and various optimizations.

4.2.1 Passing Tags

Right before each materialization point (*i.e.*, the invocation of `persist` or a Spark action), our analysis inserts a call to a native method `rdd_alloc(rdd, tag)`, with the RDD's top object (`rdd`) and the inferred memory tag (`tag`) as the arguments. This method first sets a thread-local state variable to DRAM or NVM, according to the tag, informing the current thread that a large array for an RDD will be allocated soon. Next, `rdd_alloc` sets the `MEMORY_BITS` of the top object `rdd` based on `tag`. Regardless of where it currently is, this top object will eventually be moved by the GC to the space corresponding to `tag`.

The thread then transitions into a "wait" state, waiting for this large array. In this state, the first allocation request for an array whose length exceeds a user-defined threshold (*i.e.*, a million used in our experiments) is recognized as the RDD array. Panthera then allocates the array directly into the space indicated by `tag`. To implement this, we modified both the fast allocation path, assembly code generated by the JIT compiler, and the slow path, functions implemented in C++. After this allocation, the state variable is reset and the thread exits the wait state. If `tag` is null, the array is allocated in the young generation, preferably through the thread-local allocation buffer (TLAB), and the `MEMORY_BITS` of the top object remains as the default value (00).

4.2.2 Object Migration

There are two major challenges in how to move objects: *cross-generation migration* and *object compaction*. As Panthera piggybacks on a generational GC, objects in the young generation that survive several minor GCs are deemed long-lived and moved into the old generation. We leverage this opportunity to move together objects that belong to the same logical RDD — as discussed earlier, these objects might not have been allocated in the same space initially.

Minor GC. To do this, we modified the minor collection algorithm in the Parallel Scavenge GC on which Panthera is built. The existing minor GC contains three tasks: root-task, which performs object tracing from the roots (*e.g.*, stack and global variables); old-to-young-task, which scans references from objects in the old generation to those in the young generation to identify (directly or transitively) reachable objects; and steal-task, which performs work stealing for load balancing. To support our object migration, we split old-to-young-task into a DRAM-to-young-task and NVM-to-young-task, which find objects that should be moved into the DRAM and NVM parts of the old generation, respectively.

For these two tasks, we modified the tracing algorithm to propagate the tag — for example, scanning a reference from a DRAM-based RDD array (with tag "DRAM") to a tuple object (in the young generation) propagates the tag to the tuple object (by setting its `MEMORY_BITS`). Hence, when tracing is done, all objects reachable from the array have their `MEMORY_BITS` set to the same value as that of the array. In the original GC algorithm, an object does not get promoted from the young to the old generation until it survives several minor GCs. In Panthera, however, we move the objects whose `MEMORY_BITS` is set as 01 (10) in tracing immediately to DRAM (NVM) space in the old generation, We refer to this mechanism as *eager promotion*. Objects whose `MEMORY_BITS` is not set, 00, in tracing belong to intermediate RDDs or are control objects not associated with any RDDs. The migration of these objects follows the original algorithm, that is, they will be moved only if they survive several minor GCs.

Furthermore, we also need to move RDD top objects to the appropriate part of the old generation. These top objects, whose `MEMORY_BITS` was set by the instrumented call to `rdd_alloc` at their materialization points, are visited when root-task is executed because these objects are referenced directly by stack variables. We modified the root-task algorithm to identify objects with the set `MEMORY_BITS`. These RDD top objects will also be moved to (the DRAM (01) or NVM (10) space of) the old generation by the minor GC.

Major GC. When a major GC runs, it performs memory compaction by moving objects together (in the old generation) to reduce fragmentation and improve locality. We modified the major GC to guarantee that compaction does not occur across the boundary between DRAM and NVM. Furthermore, when the major GC performs a full-heap scan, Panthera re-assesses, for each RDD array object, where the object should actually be placed based on the RDD's runtime access frequency. This frequency is measured by counting, using instrumentation, how many times a method (*e.g.*, `map` or `reduce`) has been invoked on this RDD object.

We maintain a hash table that maps each RDD object to the number of calls made on the object. Our static analysis inserts, at each such call site, a JNI (Java Native Interface) call that invokes a native JVM method to increment the call frequency for the RDD object. Frequently (infrequently) accessed array objects are moved from the NVM (DRAM) space to the DRAM (NVM) space within the old generation and all objects reachable from these arrays are moved as well. Their `MEMORY_BITS` will be updated accordingly. At the end of each major GC, the frequency for each RDD is reset.

The DRAM space of the old generation can be quickly filled up as it is much smaller than the NVM space. When the DRAM space is full, the minor GC moves all objects from the young generation to the NVM space of the old generation regardless of their memory tags.

Conflicts. If an object is reachable from multiple references and different tags are propagated through them, a conflict occurs. As discussed earlier, we resolve conflicts by giving "DRAM" higher priority than "NVM". As long as the object receives "DRAM" from any reference, it is a DRAM object and will be moved to the DRAM space of the old generation.

4.2.3 Card Optimization

In OpenJDK, the heap is divided into many *cards*, each representing a region of 512 bytes. Every object can take one or more cards, and the write barrier maintains a card table that marks certain cards dirty upon reference writes. The card table can be used to efficiently identify references during tracing. For example, upon `a.f = b`, the card that contains the object referenced by *a* is set to dirty. When a minor GC runs, the old-to-young scavenge task cleans a card if the target objects of the (old-to-young) references contained in the memory region represented by the card have been copied to the old generation.

However, if a card contains two large arrays (say *A* and *B*) — *e.g.*, *A* ends in the middle of the card while *B* starts there immediately — significant inefficiencies can result when they are scanned by two different GC threads. The card would remain dirty even if all objects referenced by *A* and *B* have been moved from the young to the old generation — neither thread could clean the card due to its unawareness of the status of the array scanned by another thread. This would cause every minor GC to scan every element of each array in the dirty card until a major GC occurs.

This is a serious problem for Big Data applications that make heavy use of large arrays. Shared cards exist pervasively when these arrays are frequently allocated and deallocated. Frequent scanning of such cards with multiple threads can incur a large overhead on NVM due to its higher read latency and reduced bandwidth. We implemented a simple optimization that adds an *alignment padding* for the allocation of each RDD array to make the end of the array align with the end of a card. Although this leads to space inefficiencies, the amount of wasted space is small (*e.g.*, less than 512 bytes for each array of hundreds of megabytes) while card sharing among arrays is completely eliminated, resulting in substantial reduction in GC time.

4.3 Applicability

Our static analysis is designed specifically for Spark and not easily reusable to other framework. However, the APIs for data placement and migration provided by the Panthera runtime system can be employed to manage memory for any Big Data system that uses a key-value array as its backbone data structure. Examples include Apache Hadoop, Apache Flink, or database systems such as Apache Cassandra.

Panthera provides two major APIs, one for pre-tenuring data structures with tags and a second for dynamic monitoring and migration. The first API takes as input an array and a tag, performing data placement as discussed earlier in this section. The tag can come from the developer's annotations in the program or from a static analysis that is designed specifically for the system to be optimized.

To illustrate, consider Apache Hadoop where both a map worker and a reduce worker may need to hold large data structures in memory. Some of these data structures are loaded from HDFS as immutable input, while others are frequently accessed. In the case of HashJoin, which is a building block for SQL engines, one input table is loaded entirely in memory while the second table is partitioned across map workers. If map workers are executed in separate threads, they all share the first table and join their own partitions of the second table with it. The first table is long-lived and frequently accessed. Hence, it should be tagged DRAM and placed in the DRAM space of the old generation, while different partitions of the second table can be placed in the young generation and they will die there quickly.

Panthera's second API takes as input a data structure object to track the number of calls made on the object. If this API is used to track the access frequency of the data structure, the data structure (and all objects reachable from it) would not be pretenured (as specified by the first API), but rather, they are subject to dynamic migration performed in the major GC. We can use this API to dynamically monitor certain objects and migrate them if their access patterns are not easy to predict statically.

Use of these two APIs enables a flexible allocation/migration mechanism that allows certain parts of the data structure (*e.g.*, for which memory tags can be easily inferred) to be pretenured and other parts to be dynamically migrated.

5 Evaluation

We have added/modified 9186 lines of C++ code in OpenJDK (build jdk8u76-b02) to implement the Panthera GC and written 979 lines of Scala code to implement the static analysis.

5.1 NVM Emulation and Hardware Platform

Most of the prior works on hybrid memories used simulators for experiments. However, none of them support Java applications well. We cannot execute managed-runtime-based distributed systems on these simulators. There also exist emulators such as Quartz [51] and PMEP [18] that support emulation of NVM for large programs using commodity multi-socket (NUMA) hardware, but neither Quartz nor PMEP could run OpenJDK. These emulators require developers to use their own libraries for NVM allocation, making it impossible for the Panthera GC to migrate objects without re-implementing the entire allocator and GC from scratch using these libraries.

As observed in [8, 51], NUMA's remote memory latency is close to NVM's latency, and hence, researchers have used a NUMA architecture as the baseline to measure emulation accuracy. Following this observation, we built our own emulator on NUMA machines to emulate hybrid memories for JVM-based Big Data systems.

We followed Quartz [51] when implementing our emulator. Quartz has two major components: (1) it uses the *thermal control register* to limit the DRAM bandwidth; and (2) it creates a daemon thread for each application process and inserts delay instructions to emulate the NVM latency. For example, if an application's CPU stall time is S, Quartz scales the CPU stall time to $S \times \frac{NVM\_latency}{DRAM\_latency}$ to emulate the latency effect of NVM. For (1), we used the same thermal control register to limit the read/write bandwidth. Like Quartz, we currently do not support different bandwidths for reads and writes. For (2), we followed Quartz's observation to use the latency of NUMA's remote memory to model NVM's latency.

An alternative approach to emulating NVM's latency is to instrument loads/stores during JIT compilation, injecting a software-created delay at each load/store. The limitation of this approach, however, is that it does not account for caching effects and memory-level parallelism.

We used one CPU to run all the computation, the memory local to the CPU as DRAM, and the remote memory as NVM. In particular, DRAM and NVM are emulated, respectively, using 2 local and 2 remote memory channels. The performance specifications of the emulated NVM are the same as those used in [51], reported in Table 2. To emulate NVM's slow write speed, we used the thermal control register to limit the bandwidth of remote memory — the read and write bandwidth is 10GB/s each. The remote memory's latency in our setting is 2.5× of that of the local memory.

Table 2. Emulated DRAM and NVM parameters.

	DRAM	NVM
Read latency (ns)	120	300
Bandwidth (GB/s)	30	10 (limited by the thermal control register)
Capacity per CPU	100s of GBs	Terabytes
Estimated price	5×	1×

Energy Estimation. We followed Lee et al. [32] to estimate energy for NVM. We used Micron's DDR4 device specifications [39] to model DRAM's power. NVM's energy has a *static* and *dynamic* component. The static component is negligible compared to DRAM [33]. The dynamic component consists of the energy consumed by reads and writes. PCM array reads consume about 2.1× larger energy than DRAM due to its need for high temperature operation [32].

NVM writes consume much more energy than DRAM writes. Upon a row-buffer miss, the energy consumed by each write has three components: (1) an *array write* that evicts data from the row buffer into the bank array, (2) an *array read* that fetches data from the bank array to the row buffer, and (3) a *row buffer write* that writes new data from the CPU last level cache to the row buffer. Assuming the row-buffer miss ratio is 0.5, we computed these three components separately by considering the row buffer's write energy (1.02pJ/bit), size (*i.e.*, 8K bits for DRAM [39], 32-bit-wide partial writeback to NVM [32]) and miss rate (0.5), as well as the array's write-back energy (16.8pJ/bit × 7.6% for NVM) and read energy (2.47pJ/bit for NVM). The factor of 7.6% is due to Lee et al.'s optimization [32] that writes only 7.6% of the dirty words back to the NVM array.

CPU's uncore events, collected with *VTune* [6], were employed to compute the numbers of reads and writes. In particular, the events we used were `UNC_M_CAS_COUNT.RD` and `UNC_M_CAS_COUNT.WR`. VTune can also distinguish reads and writes from/to local and remote memories.

5.2 Experiment Setup

We set up a small cluster to run Spark with one master node and one slave node — these two servers have a special Intel chipset with a "scalable memory buffer" that can be tuned to produce the 2.5× latency for remote memory accesses, which matches NVM's read/write latency. Since our focus is *not* on distributed computing, this cluster is sufficient for us to execute real workloads on Spark and understand their performance over hybrid memories. Table 3 reports the hardware configurations of the Spark master and Spark slave nodes. Each node has two 8-core CPU and the Parallel Scavenge collector on which Panthera was built creates 16 GC threads in each GC to perform parallel tracing and compaction.

Table 3. Hardware configuration for our servers.

Arch	NUMA, 4 sockets QPI 6.4GT/S, directory-based MESIF
CPU	E7-4809 v3 2.00GHz, 8 cores, 16 HW threads
L1-I	8 way, 32KB/core, private
L1-D	8 way, 32KB/core, private
L2	8 way, 256KB/core, private
L3	20 way, 20MB, shared
Memory	DDR 4, 1867MHz, SMI 2 channels

The negative impact of the GC latency increases with the number of compute nodes. As reported in [36], a GC run on a single node can hold up the entire cluster — when a node requests a data partition from another server that is running GC, the requesting node cannot do anything until the GC is done on the second node. Since Panthera can significantly improve the GC performance on NVM, we expect Panthera to provide even greater benefit when Spark is executed on a large NVM cluster.

System Configurations. Each CPU has a 128GB DRAM. We reserved 8GB of DRAM for the OS and the maximum amount of DRAM that can be used for Spark is 120GB. We experimented with two different heap sizes for the Spark-running JVM (64GB and 120GB) and three different DRAM sizes (1/4, 1/3, and 100% of the heap size; the rest of the heap is NVM). The configuration with 100% DRAM was used as a baseline to compute the overhead of Panthera under hybrid memories.

Prior works on NVM often used smaller DRAM ratios in their configurations. For example, Write Rationing [9] used 1GB DRAM and 32GB NVM in their experiments. However, as we deal with Big Data systems, it would not be possible for us to use a very small DRAM ratio — in our experiments, a regular RDD consumes 10-30GB memory, and hence, we had to make DRAM large enough to hold at least one RDD.

The nursery space is placed entirely in DRAM. We have experimented with several different sizes (1/4, 1/5, 1/6, and 1/7 of the heap size) for the nursery space. The performance differences between the 1/4, 1/5, and 1/6 configurations were marginal (even under the original JVM), while the configuration of 1/7 led to worse performance. We ended up using 1/6 in our experiments to achieve good nursery performance and simultaneously leave more DRAM to the old generation.

Programs and Datasets. We selected a diverse set of 7 programs. Table 4 lists these programs and the datasets used to run them. These are representative programs for a wide variety of tasks including data mining, machine learning, graph and text analytics. PR, KM, LR, and TC run directly on Spark; CC and SSSP are graph programs running on GraphX [22], which is a distributed graph engine built over Spark; BC is a program in MLib, a machine learning library built on top of Spark. We used real-world datasets to run all the seven programs. Note that although the sizes of these input datasets are not very large, there can be large amounts of intermediate data generated during the computation.

Baselines. Our initial goal was to compare Panthera with both Espresso [56] and Write Rationing [9]. However, neither of them is publicly available. Espresso proposes a programming model for developers to develop new applications. Applying it to Big Data systems would mean that we need to rewrite each allocation site, which is clearly not practical. In addition, Espresso does not migrate objects based on their access patterns.

Table 4. Spark programs and datasets.

Program	Dataset	Initial Size
PageRank (PR)	Wikipedia Full Dump, German [3]	1.2GB
K-Means (KM)	Wikipedia Full Dump, English [3]	5.7GB
Logistic Regression (LR)	Wikipedia Full Dump, English [3]	5.7GB
Transitive Closure (TC)	Notre Dame Webgraph [2]	21MB
GraphX-Connected Components (CC)	Wikipedia Full Dump, English [3]	5.7GB
GraphX-Single Source Shortest Path (SSSP)	Wikipedia Full Dump, English [3]	5.7GB
MLlib-Naive Bayes Classifiers (BC)	KDD 2012 [1]	10.1GB

The Write Rationing GC has two implementations: *Kingsguard-Nursery* (KN) and *Kingsguard-Writes* (NW). KN places the young generation in DRAM and the old generation in NVM. KW also places the young generation in DRAM. Different from KN, KW monitors object writes and dynamically migrates write-intensive objects into DRAM. Although we could not directly compare Panthera with these two GCs, we have implemented similar algorithms in OpenJDK. Under KW, almost all persisted RDDs were quickly moved to NVM. The frequent NVM reads from these RDDs, together with write barriers used to monitor object writes, incurred an average of **41%** performance overhead for our benchmarks. This is because Big Data applications exhibit different characteristics from regular, non-data-intensive Java applications.

KN appears to be a good baseline at the first sight. However, implementing it naïvely in the Parallel Scavenge collector can lead to non-trivial overhead — the reduced bandwidth in NVM can create a huge impact on the performance of a multi-threaded program; this is especially the case for Parallel Scavenge that attempts to fully utilize the CPU resources to perform parallel object scanning and compaction.

To obtain a better baseline, we placed the young generation in DRAM and supported the old generation with a mix of DRAM and NVM. In particular, we divided the virtual address space of the old generation into a number of chunks, each with 1GB, and used a probability to determine whether a chunk should be mapped to DRAM or NVM. The probability is derived from the DRAM ratio in the system. For example, in a system where the DRAM-to-memory ratio is 1/4 (1/4 DRAM), each chunk is mapped to DRAM with 1/4 probability and to NVM with 3/4 probability. Note that this is common practice [21, 53] to utilize the combined bandwidth of DRAM and NVM. We refer to this configuration as *unmanaged*, which outperforms both KN and KW for our benchmarks.

5.3 Performance and Energy

Figure 4 reports the overall performance and energy results when a 64GB heap is used and DRAM to memory ratio is 1/3 (1/3 DRAM). The performance and energy results of each configuration are normalized *w.r.t.* those of the 64GB DRAM-only version. Compared to the DRAM-only version, the unmanaged version reduces energy by 26.7% with a

21.4% execution time overhead. In contrast, Panthera reduces energy by 32.3% at a 4.3% execution time overhead.

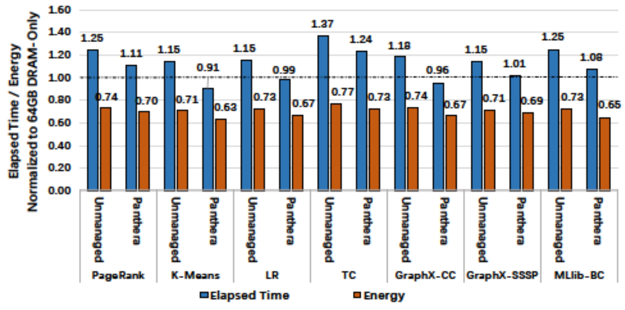


Figure 4. Overall performance and energy results under a 64GB heap; DRAM to memory ratio is 1/3.

When the heap size is 120 GB (not shown in Figure 4, but summarized later in Figure 6 and Figure 7), the unmanaged version reduces energy by 39.7% at a 19.3% execution time overhead. In contrast, Panthera reduces energy by 47.0% with less than 1% execution time overhead. Clearly, considering the RDD semantics in data placement provides significant benefits in both energy and performance.

GC Performance. To understand the GC performance, we broke down the running time of each program into the mutator and GC time; these results (under the 64GB heap) are shown in Figure 5. Compared to the baseline, the unmanaged version introduces performance overhead of 60.4% and 6.9% in the GC and computation, respectively; while for Panthera these two overheads are, respectively, 4.7% and 4.5%. Under the 120GB heap, the GC performance overhead of the unmanaged version and Panthera are, respectively, 58.0% and 3.1%. Note that due to large amounts of intermediate data generated, the GC is frequently triggered for these programs.

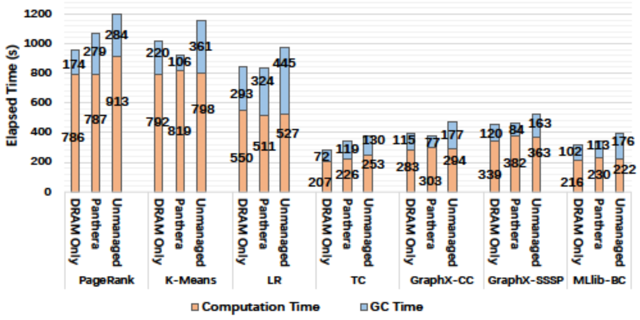


Figure 5. GC performance (64GB heap).

Since the GC is a memory-intensive workload, inappropriate data placement can lead to significantly higher memory access time and thus a large penalty. The penalty comes from two major sources. First, NVM's limited bandwidth (which is about 1/3 of that of DRAM) has a large negative impact on the performance of Parallel Scavenge, which launches 16 threads to perform parallel tracing and object copying in each (nursery and full-heap) GC. Given this high degree of parallelism, the performance of the nursery GC is degraded significantly when scanning objects in NVM. Second, object tracing is a read-intensive task, which suffers badly from NVM's higher read latency.

Panthera improves the GC performance by pretenuring frequently-accessed RDD objects in DRAM and performing optimizations including *eager promotion* (§4.2.2) and *card padding* (§4.2.3). *Eager promotion* reduces the cost of (old-to-young) tracing in each minor GC, while *card padding* eliminates unnecessary array scans in NVM, which are sensitive to both latency and bandwidth. A further breakdown shows that *eager promotion*, alone, contributes an average of 9% of the total GC performance improvement. The contribution of *card padding* is much more significant — without this optimization, the GC time increases by 60% due to the impact of NVM's substantially limited bandwidth and increased latency on the performance of parallel card scanning. In fact, this impact is so large that the other optimizations would not work well when card padding is disabled.

Varying Heaps and Ratios. To understand the impact of the heap sizes and DRAM ratios (DRAM to total memory), we have conducted experiments with two heap sizes (64GB, 120GB) and two DRAM ratios (1/3, 1/4) on four programs PR, LR, CC, and BC. Figure 6 reports the time results of these configurations. Panthera's time overheads are, on average, 9.5%, 3.4%, 2.1%, and 0%, respectively, under the four configurations (64GB, 1/4), (64GB, 1/3), (120GB, 1/4), and (120GB, 1/3). The overheads for the unmanaged version are 25.9%, 20.9%, 23.9%, and 19.3%, respectively, under these same four configurations.

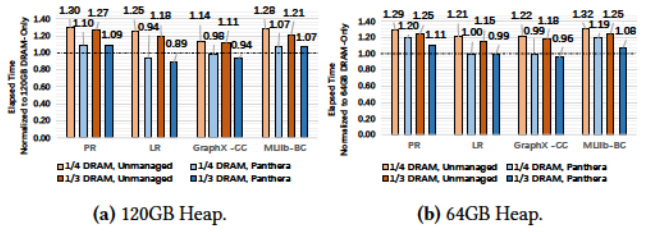
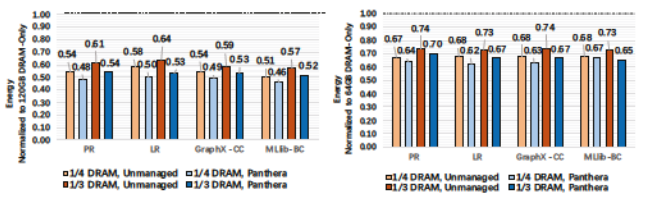


Figure 6. Performance for two DRAM ratios + two heaps.

We make two interesting observations. First, Panthera is more sensitive to the DRAM ratio than the heap size. The time overhead can be reduced by almost 10% when the DRAM ratio increases from 1/4 to 1/3. The reason is that more frequently accessed RDDs are moved to DRAM, reducing the memory latency and bandwidth bound of NVM. Another observation is that the unmanaged version is much less sensitive to DRAM ratio — the time overhead is reduced by only 5% when the DRAM ratio increases to 1/3. This is because arbitrary data placement leaves much of the frequently-accessed data in NVM, making CPUs stall heavily when accessing NVM.

Figure 7 depicts the energy results for the two heaps and two DRAM/NVM ratios. For the 64GB heap, the unmanaged version reduces energy by an average of 32.2% and 26.5%, respectively, under the 1/4 and 1/3 DRAM ratio, while Panthera reduces energy by 36.0% and 32.7% under these same ratios. The energy reductions for the 120GB heap are much more significant — the unmanaged version reduces energy by 45.7% and 39.7%, respectively, under the 1/4 and 1/3 DRAM ratios, while the energy reduction under Panthera increases to 51.7% and 47.0% for these two ratios.



(a) 120GB Heap. (b) 64GB Heap.

Figure 7. Energy for two DRAM ratios + two heaps.

5.4 Memory Access Analysis

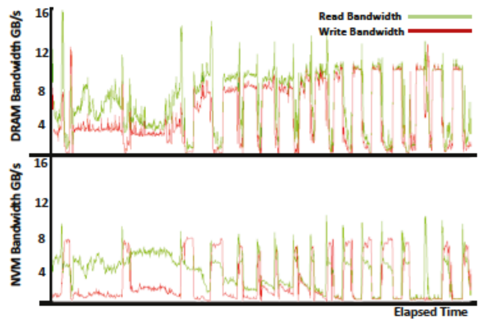
NVM has high latency and low bandwidth. In general, the performance penalty caused by high latency increases with the number of memory accesses. For the same number of memory accesses, NVM incurs higher performance penalty for applications that have instantaneous bandwidth requirements which are beyond NVM’s bandwidth. Figure 8 depicts the read/write bandwidth for unmanaged and Panthera on GraphX-CC. Compared to the unmanaged version, Panthera migrates most of the memory reads/writes from NVM to DRAM and reduces the high instantaneous memory access bandwidth requirements (*i.e.*, peaks in the figure). Because Panthera allocates/moves frequently accessed data to DRAM, it reduces unnecessary NVM accesses (§4.2.2, §4.2.3).

5.5 Overhead of Monitoring and Migration

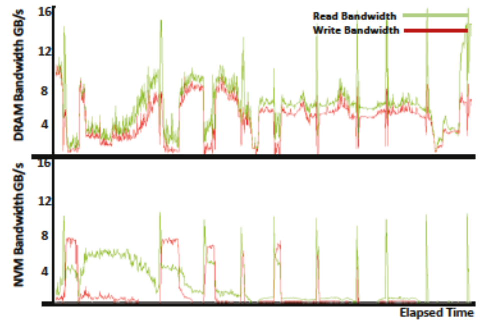
As discussed in §4.2, Panthera performs lightweight method-level monitoring on RDD objects to detect misplaced RDDs for dynamic migration. This subsection provides a closer examination of dynamic migration’s overhead.

As we monitor only method calls invoked on RDD objects, we find dynamic monitoring overhead is negligible, *i.e.* it is less than 1% across our benchmarks. For example, for PageRank, only about 300 calls were observed on all RDD objects in a 20-minute execution. The second column of Table 5 reports the number of calls monitored for each application. For GraphX applications, which has thousands of RDD calls, the monitoring overheads are still less than 1%.

Dynamic migration (performed by the major GC) rarely occurs in our experiments, as can be seen from the third column of Table 5. There are two main reasons. First, the frequency of a major collection is very low because a majority of objects die young and most of the collection work is done by the minor GC. Second, for four applications (PR, KM, TC, and LR), our static analysis results are accurate enough and, hence, dynamic migration is never needed.



(a) Unmanaged, DRAM ratio = 1/3



(b) Panthera, DRAM ratio = 1/3

Figure 8. GraphX-CC’s memory access bandwidth.

Table 5. Dynamic monitoring and migration.

Program	# Calls monitored	# RDDs migrated
PR	328	0
KM	550	0
LR	333	0
TC	217	0
CC	2945	1
SSSP	3632	1
BC	336	0

We observed that only two RDDs (during the executions of CC and SSSP) were migrated dynamically. Note that both CC and SSSP are GraphX applications. Each iteration of the processing creates new RDDs representing the updated graph and persists them. At the end of each iteration, the RDDs representing the old graph are explicitly *unpersisted*. Our static analysis, due to lack of support for the *unpersist* call, marks both old and new graph RDDs as hot data and generates a DRAM tag for all them. These RDD objects are then allocated in DRAM and their data objects are promoted eagerly to the DRAM space of the old generation. The RDD objects representing the old graphs, if they can survive a major GC, are migrated to the NVM space of the old generation due to their low access frequency.

To have better understanding of the individual contributions of pretenuring and dynamic migration, we have disabled the monitoring and migration and rerun the entire experiments. The performance difference was negligible (*i.e.*, less than 1%). Hence, we conclude that most of Panthera's benefit stems from pretenuring, which improves the performance of both the mutator and the GC. However, dynamic monitoring and migration increases the generality of Panthera's optimizations, making Panthera applicable to applications with diverse access characteristics.

6 Related Work

Hybrid Memories for Managed Runtime. To our knowledge, Panthera is the first practical work to optimize data layout in hybrid memories for managed-runtime-based distributed Big Data platforms. Existing efforts [9, 12, 21, 26, 27, 44, 49, 52, 56] that attempt to support persistent Java focus on regular applications or need to rebuild the platforms.

Inoue and Nakatani [24] identify code patterns in Java applications that can cause cache misses in L1 and L2. Gao et al. [20] propose a framework including support from hardware, the OS, and the runtime to extend NVM's lifetime. Two recent works close to Panthera are Espresso [56] and Write Rationing [9]. However, they were not designed for Big Data systems. Espresso is a JVM-based runtime system that enables persistent heaps. Developers can allocate objects in a persistent heap using a new instruction pnew while the runtime system provides crash consistency for the heap. Applying Espresso requires rewriting the Big Data platforms (*e.g.*, Spark) using pnew, which is not practical.

Write Rationing [9] is a GC technique that places highly mutated objects in DRAM and mostly-read objects in NVM to increase NVM lifetime. Like Espresso, this GC focuses on individual objects and does not consider application semantics. Panthera's nursery space is also placed in DRAM, similar to the Kingsguard-Nursery in Write Rationing. However, instead of focusing on individual objects, Panthera utilizes Spark semantics to obtain access information at the array granularity, leading to effective pretenuring and efficient runtime object tracking.

Memory Structure. There are two kinds of hybrid-memory structures: *flat structure*, where DRAM and NVM share a single memory space, and *vertical structure*, where DRAM is used as a buffer for NVM to store hot data. The vertical structure is normally managed by hardware and transparent to the OS and applications [28, 32, 35, 37, 46, 57, 59, 62]. Qureshi et al. [46] shows that a vertical structure with only 3% DRAM can reach similar performance to its DRAM-only version. However, the overhead of page monitoring and migration increases linearly with the working set [53]. The space overhead *e.g.*, the tag store space of DRAM buffer, can also be high with a large volume of NVM [38].

Page-based Migration. A great number of existing works use memory controllers to monitor page read/write frequency [15, 16, 19, 23, 34, 45, 47, 53, 57, 61] and migrate the top-ranked pages to DRAM. Another type of hybrid memory, composed of 3D-stacked DRAM and commodity DRAM, also adapts similar page monitoring policies [17, 25]. However, none of these techniques were designed for Big Data systems. Hassan et al. [23] show that, for some applications, migrating data at the object level can reduce power consumption.

For Big Data applications that have very large memory consumption, continuous monitoring at the page granularity can incur an unreasonable overhead. Page migration also incurs overhead in time and bandwidth. Bock et al. [13] report that page migration can increase execution time by 25% on average. Panthera uses static analysis to track memory usage at the RDD granularity, incorporating program semantics to reduce the dynamic monitoring overheads.

Static Data Placement. There exists a body of work that attempts to place data directly in appropriate spaces based either on their access frequencies [15, 34, 45, 50, 57] or on the result of a program analysis [19, 23, 53]. Access frequency is normally calculated using a static data liveness analysis or offline profiling. Chatterjee et al. [15] place a single cache-line across multiple memory channels. Critical words (normally the first word) in a cache-line are placed in a low-latency channel. Wei et al. [53] show that the group of objects allocated by the same site in the source code exhibit similar lifetime behavior, which can be leveraged for static data placement.

Li et al. [34] develop a binary instrumentation tool to statistically report memory access patterns in stack, heap, and global data. Phadke and Narayanasamy [45] profile an application's MLP and LLC misses to determine from which type of memory the application could benefit the most. Kim et al. [29] develop a key-value store for high-performance computers with large distributed NVM, which provides developers with a high-level interface to use the distributed NVM. However, none of these techniques were designed for managed Big Data systems.

7 Conclusion

We present Panthera, the first memory management technique for managed Big Data processing over hybrid memories. Panthera combines static analysis and GC techniques to perform semantics-aware data placement in hybrid memory systems. Our evaluation shows that Panthera reduces energy significantly without incurring much extra time overhead.

Acknowledgments

We thank the anonymous reviewers for their thorough and insightful comments. We are especially grateful to our shepherd Jennifer Sartor for her feedback, helping us improve the paper substantially. The work is supported in part by the National Key R&D Program of China grant 2016YFB1000201, National Science Foundation grants CNS-1613023, CNS-1703598, and CNS-1763172, Office of Naval Research grants N00014-16-1-2913 and N00014-18-1-2037, National Natural Science Foundation of China grants 61802368, 61521092, 61432016, 61432018, 61332009, 61702485, and 61872043, as well as gifts from Alibaba, Intel and VMware.

References

[1] 2012. LIBSVM Data: Classification. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets.

[2] 2017. Notre dame network dataset. http://konect.uni-koblenz.de/networks/web-NotreDame.

[3] 2017. Wikipedia links, network dataset. http://konect.uni-koblenz.de/networks.

[4] 2018. 3D XPoint^{TM}: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[5] 2019. Apache Spark^{TM}. https://spark.apache.org.

[6] 2019. Intel VTune^{TM} Amplifier. https://software.intel.com/en-us/vtune.

[7] 2019. OpenJDK. https://openjdk.java.net.

[8] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Emulating Hybrid Memory on NUMA Hardware. *CoRR* (2018).

[9] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-Rationing Garbage Collection for Hybrid Memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. 62–77.

[10] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-performance Latch-free Range Index for Non-volatile Memory. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 553–565.

[11] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind Logging. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 337–348.

[12] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. 1996. An Orthogonally Persistent Java. *SIGMOD Rec.* 25, 4 (Dec. 1996), 68–75.

[13] Santiago Bock, Bruce R. Childers, Rami G. Melhem, and Daniel Mossé. 2014. Concurrent page migration for mobile systems with OS-managed hybrid memory. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14)*. 31:1–31:10.

[14] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.* 30, 1-7 (April 1998), 107–117.

[15] N. Chatterjee, M. Shevgoor, R. Balasubramonian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer. 2012. Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. 13–24.

[16] G. Dhiman, R. Ayoub, and T. Rosing. 2009. PDRAM: A hybrid PRAM and DRAM main memory system. In *Proceedings of the 46th Annual Design Automation Conference (DAC '09)*. 664–669.

[17] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. 2010. Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. 1–11.

[18] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. 15:1–15:15.

[19] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. 15:1–15:16.

[20] Tiejun Gao, Karin Strauss, Stephen M. Blackburn, Kathryn S. McKinley, Doug Burger, and James R. Larus. 2013. Using managed runtime systems to tolerate holes in wearable memories. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI '13)*. 297–308.

[21] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: A Garbage Collector for Big Data on Big NUMA Machines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. 661–673.

[22] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. 599–613.

[23] Ahmad Hassan, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2015. Software-managed Energy-efficient Hybrid DRAM/NVM Main Memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers (CF '15)*. 23:1–23:8.

[24] Hiroshi Inoue and Toshio Nakatani. 2012. Identifying the Sources of Cache Misses in Java Programs Without Relying on Hardware Counters. In *Proceedings of the 2012 International Symposium on Memory Management (ISMM '12)*. 133–142.

[25] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. 2010. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *The Sixteenth International Symposium on High-Performance Computer Architecture (HPCA'10)*. 1–12.

[26] Mick Jordan. 1996. Early Experiences with Persistent Java. In *The First International Workshop on Persistence and Java.*

[27] Mick Jordan and Malcolm Atkinson. 2000. *Orthogonal Persistence for the Java^{TM} Platform: Specification and Rationale.* Technical Report. Mountain View, CA, USA.

[28] Taeho Kgil, David Roberts, and Trevor Mudge. 2008. Improving NAND Flash Based Disk Caches. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. 327–338.

[29] Jungwon Kim, Seyong Lee, and Jeffrey S. Vetter. 2017. PapyrusKV: A High-performance Parallel Key-value Store for Distributed NVM Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. 57:1–57:14.

[30] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '13)*. 256–267.

[31] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. 460–477.

[32] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable DRAM Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. 2–13.

[33] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (Jan. 2010), 143–143.

[34] Dong Li, Jeffrey S. Vetter, Gabriel Marin, Collin McCurdy, Cristian Cira, Zhuo Liu, and Weikuan Yu. 2012. Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12)*. 945–956.

[35] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. 2017. Utility-Based Hybrid Memory Management. In *2017 IEEE International Conference on Cluster Computing (CLUSTER '17)*. 152–165.

[36] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. 457–471.

[37] Prasanth Mangalagiri, Karthik Sarpatwari, Aditya Yanamandra, VijayKrishnan Narayanan, Yuan Xie, Mary Jane Irwin, and Osama Awadel Karim. 2008. A Low-power Phase Change Memory Based Hybrid Cache Architecture. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI (GLSVLSI '08)*. 395–398.

[38] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. 2012. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. *IEEE Computer Architecture Letters* 11, 2 (July 2012), 61–64.

[39] Micron. 2017. TN-40-07: Calculating Memory Power for DDR4 SDRAM Introduction. https://www.micron.com/-/media/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf.

[40] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. 2009. Operating System Support for NVM+DRAM Hybrid Main Memory. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems (HotOS '09)*. 14–14.

[41] Gaku Nakagawa and Shuichi Oikawa. 2015. NVM/DRAM hybrid memory management with language runtime support via MRW queue. In *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD '15)*. 357–362.

[42] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-performance Big-data-friendly Garbage Collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. 349–365.

[43] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. 675–690.

[44] James O'Toole, Scott Nettles, and David Gifford. 1993. Concurrent Compacting Garbage Collection of a Persistent Heap. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. 161–174.

[45] Sujay Phadke and Satish Narayanasamy. 2011. MLP aware heterogeneous memory system. In *Proceedings of 2011 IEEE Design, Automation Test Conference in Europe (DATE '11)*. 1–6.

[46] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)*. 24–33.

[47] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. 85–95.

[48] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. 672–685.

[49] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. 1994. Lightweight Recoverable Virtual Memory. *ACM Trans. Comput. Syst.* 12, 1 (Feb. 1994), 33–57.

[50] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B. Gibbons, and Onur Mutlu. 2018. A Case for Richer Cross-layer Abstractions: Bridging the Semantic Gap with Expressive Memory. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. 207–220.

[51] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. 2015. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15)*. 37–49.

[52] Chenxi Wang, Ting Cao, John Zigman, Fang Lv, Yunquan Zhang, and Xiaobing Feng. 2016. Efficient Management for Hybrid Memory in Managed Language Runtime. In *16th IFIP International Conference on Network and Parallel Computing (NPC '16)*. 29–42.

[53] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. 2015. Exploiting Program Semantics to Place Data in Hybrid Memory. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT '15)*. 163–173.

[54] H. S. P. Wong, H. Lee, S. Yu, Y. Chen, Y. Wu, P. Chen, B. Lee, F. T. Chen, and M. Tsai. 2012. Metal-Oxide RRAM. *Proc. IEEE* 100, 6 (June 2012), 1951–1970.

[55] H. S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. 2010. Phase Change Memory. *Proc. IEEE* 98, 12 (Dec 2010), 2201–2227.

[56] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. 70–83.

[57] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael Harding, and Onur Mutlu. 2012. Row buffer locality aware caching policies for hybrid memories. In *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD '12)*. 337–344.

[58] Hanbin Yoon, Justin Meza, Naveen Muralimanohar, Norman P. Jouppi, and Onur Mutlu. 2014. Efficient Data Mapping and Buffering Techniques for Multilevel Cell Phase-Change Memories. *ACM Trans. Archit. Code Optim.* 11, 4 (Dec. 2014), 40:1–40:25.

[59] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. 2017. Banshee: Bandwidth-efficient DRAM Caching via Software/Hardware Cooperation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*. 1–14.

[60] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 12)*. 15–28.

[61] Wangyuan Zhang and Tao Li. 2009. Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*. 101–112.

[62] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. 14–23.

[63] Omer Zilberberg, Shlomo Weiss, and Sivan Toledo. 2013. Phase-change Memory: An Architectural Perspective. *ACM Comput. Surv.* 45, 3 (July 2013), 29:1–29:33.