TeraCache: Efficient Spark Caching Over Fast Storage Devices

Iacovos G. Kolokasis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Co-Advisors: Prof. Polyvios Pratikakis and Dr. Foivos Zakkak

This work has been performed at the Institue of Computer Science (ICS) Foundation for Research and Technology - Hellas (FORTH).

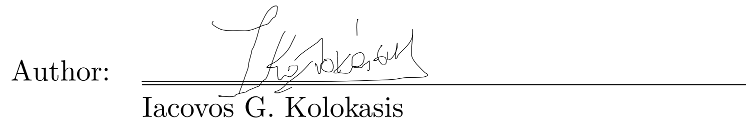
The work has been supported by the Evolve Project (Grant Agreement N° 825061), funded by the European Union Horizon 2020 Research and Innovation Programme.

University of Crete
Computer Science Department

TeraCache: Specialized Heap for Managed Big Data Systems on Fast Storage Devices

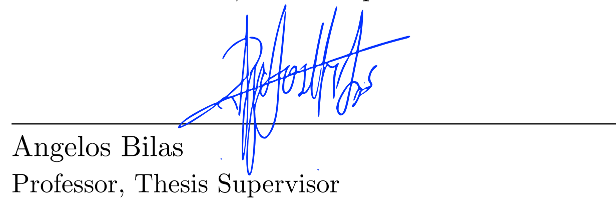
Thesis submitted by
Iacovos G. Kolokasis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

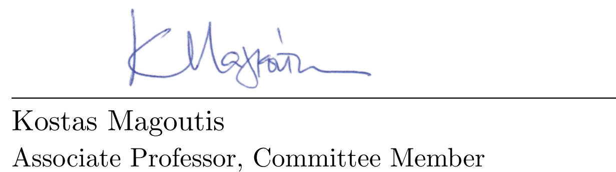
THESIS APPROVAL

Author:

Iacovos G. Kolokasis

Committee approvals:

Polyvios Pratikakis
Assistant Professor, Thesis Supervisor


Angelos Bilas
Professor, Thesis Supervisor


Kostas Magoutis
Associate Professor, Committee Member

Departmental approval:

Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, Oct 2020

Abstract

Many analytics computations are dominated by iterative processing stages, executed until a convergence condition is met. To accelerate such workloads while keeping up with the exponential growth of data and the slow scaling of DRAM capacity, Spark employs off-heap caching of intermediate results. However, off-heap caching requires serialization and deserialization (*serdes*) of data that add significant overhead especially with growing datasets.

This thesis proposes *TeraCache*, an extension of the Spark data cache that avoids the need of *serdes* by keeping all cached data on-heap but off-memory, using memory-mapped I/O (mmio). To achieve this, *TeraCache* extends the original JVM heap with a managed heap that resides on a memory-mapped fast storage device and is exclusively used for cached data. Preliminary results show that the *TeraCache* prototype can speed up Machine Learning (ML) workloads that cache intermediate results by up to 37% compared to the state-of-the-art *serdes* approach.

Περίληψη

Οι εφαρμογές που εκτελούν αλγόριθμους μηχανικής μάθησης αποτελούνται από μεγάλος πλήθος επαναληπτικών υπολογισμών επεξεργασίας δεδομένων που εκτελούνται μέχρι να ικανοποιήσουν μια συνθήκη σύγκλισης. Για να εκτελεί τους υπολογισμούς μηχανικής μάθησης σε μικρό χρόνο εκτέλεσης συμβαδίζοντας παράλληλα με την εκθετική αύξηση του μεγέθους των δεδομένων καθώς και την αργή αύξηση της κλιμακοσιμότητητας της μνήμης τυχαίας προσπέλασης (DRAM), το Spark χρησιμοποιεί γρήγορες συσκευές αποθήκευσης για την προσωρινή αποθήκευση των ενδιάμεσων αποτελεσμάτων εκτός της μνήμης. Ωστόσο, η προσωρινή αποθήκευση εκτός της μνήμης απαιτεί τη σειριοποίηση και την αποσειριοποίηση (serdes) των δεδομένων, το οποίο προσθέτει σημαντική επιβάρυνση στο χρόνο εκτέλεσης ειδικά όσο αυξάνεται το σύνολικό μέγεθος των δεδομένων επεξεργασίας.

Αυτή η διατριβή προτείνει το μηχανισμό TeraCache, μια επέκταση της προσωρινής μνήμης αποθήκευσης ενδιάμεσων δεδομένων του συστήματος ανάλυσης δεδομένων Spark που αποφεύγει την ανάγκη για σειριοποίηση/αποσειριοποίηση διατηρώντας όλα τα αποθηκευμένα δεδομένα στο σωρό (heap) αλλά εκτός μνήμης, χρησιμοποιώντας χαρτογραφημένη μνήμη εισόδου/εξόδου (mmio). Για να επιτευχθεί αυτό, η TeraCache επεκτείνει το σωρό της JAVA εικονικής μηχανής (JVM) με έναν διαχειριζόμενο σωρό που βρίσκεται σε μια γρήγορη χαρτογραφημένη στη μνήμη συσκευή αποθήκευσης και χρησιμοποιείται αποκλειστικά για την αποθήκευση ενδιαμεσων αποτελεσμάτων. Προκαταρκτικά αποτελέσματα δείχνουν ότι η πρωτότυπη υλοποίηση της TeraCache μπορεί να επιταχύνει τα εφαρμογές μηχανικής μάθησης που αποθηκεύουν ενδιάμεσα αποτελέσματα έως και 37% σε σύγκριση με τις υφιστάμενες μεθόδους αποθήκευσης.

Acknowledgements

My Master's study at the University of Crete is an incredible and happy journey in my life. I grew up from an undergraduate student to a graduate researcher who really enjoys exploring and solving new real and challenging problems.

I would like to express my sincere appreciation to my supervisors Prof. Angelos Bilas and Prof. Polyvios Pratikakis for their unreserved supervision, advice, guidance, help, and training. Their high-level critical thinking, research methodologies, and hard-working, educate, and improve me in research, social, and daily life. I am always proud, honored, and lucky for the collaboration with them. After 3 years of learning and training, I enjoy the meeting time with them and understand why always Prof. Bilas, said "Our work has no schedule and needs a lot of effort, but solving real problems is always fun!". I can always get advice from them for both research and career and I am always surprised by their deep knowledge, insightful feedback, and sharp research sense. I am so thankful for their kindness and unwavering support at every step of this journey.

Furthermore, I am grateful to my colleagues Anastasios Papagianis and Foivos Zakkak for their insightful advice, guidance, and help. I learned so much through the discussion and collaboration with them. I am really thankful for their help and support.

It is an honor for me to be a member of the CARV lab. I would like to thank: Christos Kozanitis, Manolis Marazakis, Manos Pavlidakis, Stelios Mavridis, Nikos Papakostantinou, Theocharis Vavouris, Yianis Sfakianakis, Eleni Kanellou, Christi Symeonidou, Klodjia Hidri, Giorgos Xanthakis, and Giorgos Saloustros. I really enjoy the collaboration with them!

Last but not least, none of these could have happened without the unfailing support from my wonderful friends (Iakovos, Pavlos, Rafail, and Anna) and my family (my mum Theano, my dad George, my sister Antigoni and my brother Nikos). I would like to acknowledge them for inspiring and helping me throughout my graduate studies in Crete, the good times and memories we have shared together, and their continued support in the years to come.

Contents

Table of Contents **i**

List of Figures and Tables **iii**

1 Introduction **1**

2 Background **7**

3 *TeraCache* Design **11**
- 3.1 Motivation 11
- 3.2 System Overview 12
- 3.3 *TeraCache* heap allocation using *mmap* on NVMe 13
- 3.4 *TeraCache* heap management avoids costly GC 13
- 3.5 Caching RDDs in *TeraCache* 14
- 3.6 Batch Transfer of Objects to *TeraCache* during Full GC 14
- 3.7 Reclamation of Cached Data Objects 15
- 3.8 Dealing with Class Loading, Finalizers 17
- 3.9 Division of DRAM between DR1 and DR2 17
- 3.10 Prototype Implementation 19

4 Evaluation **21**

5 Related Work **27**
- 5.1 Hybrid Memory for Java Heap 27
- 5.2 Caching Mechanism in Spark 28
- 5.3 Region-based memory management for big data systems 29

6 Conclusions and Future Work **31**

Bibliography **33**

List of Figures

1.1 (a) Hybrid caching outperforms on-demand recomputation. (b) Disk-only caching incurs high *serdes* overhead and lower GC time, whereas hybrid caching exhibits the reverse behavior. 2

2.1 Internal processing of RDD caching in the Spark executor. 8

3.1 (a) Off-memory caching via *serdes*. (b) *TeraCache*, on-heap RDD cache over a memory-mapped fast storage device $S_{DR1} + S_{DR2} = S_{DRAM}$ ($\mathbf{S_{DRAM}}$=DRAM size, $\mathbf{S_{DR1}}$=DR1 size, $\mathbf{S_{DR2}}$=DR2 size, $\mathbf{S_{C}}$=Capacity of fast storage device). 12

3.2 Java object layout in JVM. We use an additional *TeraCache* word in object header for objects that need to be moved to *TeraCache*. . 14

4.1 (a) Applications performance using Linux swap, hybrid and *TeraCache*. (b) Execution Time, (c) Total GC time, and (d) Total number of page faults for each *TeraCache* configuration with 64GB dataset. 22

4.2 Cached data objects demographics 23

4.3 LgR on HDD vs NVMe SSD storage devices. 24

List of Tables

4.1 H1 size is equal to DR1. DR2 is the memory-mappings for *TeraCache* in DRAM. Total DRAM is the sum of DR1 and DR2. Total Virtual Heap is the sum of H1 and the *TeraCache* heap size. . . . 21

4.2 Spark heap demographics for popular machine learning benchmarks. *Allocation volumes are high and intermediate RDDs are less than 10% of total allocation but they need large amount of DRAM.* . . . 23

Chapter 1

Introduction

Analytic applications often use ML [21] algorithms to process massive amounts of data. Such applications iterate over one or more computation steps until a convergence condition is met. To speed up such workloads, Spark [45] caches large intermediate results of complex computation pipelines and reuses them in each step. Intermediate results are stored as Resilient Distributed Diastase (RDDs) [43] in a Least Recently Used (LRU) cache in memory.

RDD is an immutable collection of objects. Each RDD is split into multiple partitions, which is evaluated on the different nodes in the cluster across different stages. RDDs can survive in memory across stages, e.g., in a single iterative analytics job or across multiple interactive jobs. The Spark programming model exposes an API that supports cache RDD in-memory (on-heap) in deserialized form or off-memory (off-heap) in serialized form to be materialized in subsequent stages.

At a low level, an RDD is a read-only collection of similarly-typed tuples partitioned across a cluster. Each RDD partition resides on a node as a hierarchical data structure consisting of a root RDD which references a Java array. The array, in turn, consists of a set of objects, e.g., key-value pairs. In the case of node failures, Spark maintains enough information to recompute an RDD partition.

Figure 1.1(a) shows the performance impact of RDD caching for three well-known ML workloads: Linear Regression (LR), Logistic Regression (LgR), and Support-Vector Machines (SVM). All workloads run with a 64GB dataset and a single Spark executor using 32GB DRAM and 30 CPU cores. Caching RDDs in both memory and disk (hybrid) improves performance up to 90%, compared to recomputing intermediate results on demand at every stage.

DRAM-only caching is not a long-term solution, as the amount of data generated and processed increases at a high rate [34, 35], while DRAM scaling reaches its limits [23, 26, 18]. Cached data increasingly exceed physical DRAM size, making workloads prone to recomputing intermediate results at each step. Therefore, the current practice is to use fast storage devices to increase Spark's effective cache size for intermediate RDDs [47]. NAND flash storage devices, such as SSD and

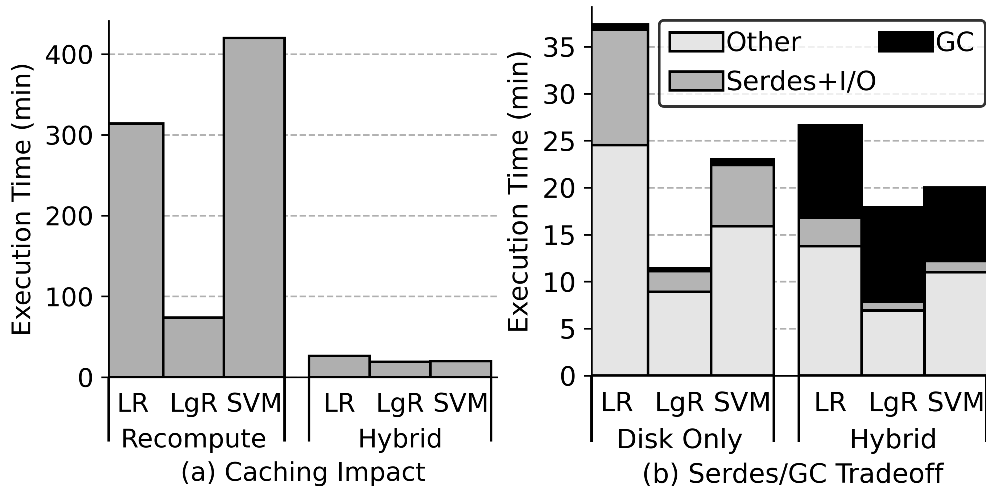


Figure 1.1: (a) Hybrid caching outperforms on-demand recomputation. (b) Disk-only caching incurs high *serdes* overhead and lower GC time, whereas hybrid caching exhibits the reverse behavior.

NVMe block devices [1], as well as NVMs, have higher density and capacity than DRAM [33, 29].

SSD and NVMe devices scale to terabytes per PCIe slot [25] at a lower cost [8], while DRAM scales to GBs per DIMM. Moreover, NVMe block devices provide higher capacity at a lower cost compared to NVM, while still having access latency in the order of a few μs and they perform hundreds of thousands of IOPS for both reads and writes. For these reasons, today Spark tries to exploit the large capacity of fast storage devices as an off-heap cache in combination with a small on-heap cache in-memory to store intermediate results.

During execution, Spark initially places RDDs in the on-heap (DRAM) cache. If an RDD block does not fit in memory, it is serialized to the off-heap (disk) cache and its memory is collected during the next garbage collection (GC) cycle. Similarly, if there is not enough memory, the LRU cache evicts older entries to the off-heap cache. When Spark refers to an RDD that is stored off-heap, it deserializes the serialized block from disk back into memory. Every block is serialized at most once since RDDs are immutable. However, deserialization can occur multiple times per block in each iterative stage.

Today, despite outperforming the recomputation strategy, caching in off-heap device caches incur high *serdes* overhead. According to Zhang *et al.* [46], *serdes* rather than disk I/O dominates overhead. Figure 1.1(b) shows the performance impact of off-heap RDD caches on LR, LgR, and SVM, when storing RDDs only off-heap (disk). All workloads run with a 64GB dataset and a single Spark executor using 32GB DRAM and 30 CPU cores. Note the large impact of *serdes* overhead with off-heap caching. *Serdes* accounts for 27% on average of total execution time using only disk storage. Deserialization accounts for 80%-90% of the total

serdes overhead because the workload retrieves immutable cached RDDs from the device in every iteration. *Serdes* overhead becomes worse as storage device speed improves and the gap to CPU and memory performance narrows down. The total execution time also increases in the case of disk-only caching, mainly due to reduced parallelism and idle CPU time, as disk throughput cannot keep 30 CPU cores at full load.

Using an on-heap cache (DRAM) together with an off-heap cache (disk) reduces *serdes* cost, but it also incurs significant GC overhead. Figure 1.1(b) shows the performance impact of storing RDDs in a hybrid cache, both on-heap (DRAM) and off-heap (disk), as is currently common practice. Using a relatively large on-heap cache (Spark reserves 60% of the heap as cache), *serdes* overhead decreases considerably, by 20% on average, by keeping some RDDs in memory compared to storing them exclusively on disk. However, such a large on-heap cache increases GC time between 13x (SVM) and 36x (LgR), compared to disk-only caching. Cached RDDs are initially placed in the heap, resulting in a higher ratio of long-lived objects to short-live objects. Hence, GC consumes more time marking live objects in the JVM heap [10, 41] and ends up reclaiming a smaller percentage of the heap, since a big portion is occupied by cached RDDs. In essence, Spark uses the DRAM-only JVM heap both for execution and cache memory. This can lead to unpredictable performance or even failures because caching large data causes extra GC pressure during execution time.

In this work we argue that RDD caching should be performed only in a large, managed, on-heap cache, in a part of the heap that is memory-mapped onto a fast storage device and is not garbage collected. *TeraCache* divides the JVM heap into execution and a cache part, locating the execution heap solely in DRAM and the cache part in a DRAM-mapped block device. This inherently eliminates *serdes* and all associated overheads because the memory-mapped heap is mapped into the JVM virtual address space. Additionally, *TeraCache* is inline with device technology trends and server power limitations.

We recognize that cached RDDs are long-lived with similar lifetime objects and their scope is defined by the application. Cached RDDs properties enable us to organize *TeraCache* into regions and avoid garbage collection. Each region contains a group of similar-lifetime objects. As a result, we can free the entire region instead of individual objects in it. Object lifetime is managed explicitly by the application thus, we avoid GC traversals over them. All the non-cached objects stay in a regular, garbage collected heap (JVM-heap), which is used for execution. To ensure memory safety, we perform o deep copy of the RDDs (following the transitive-closure of their references) from the JVM-heap to *TeraCache*. This way we avoid creating references from *TeraCache* to the JVM-heap.

Finally, we observe a trade-off on how DRAM should be divided between execution and cache memory in Spark. Clearly, the execution part of the heap should use enough DRAM to not cause GC pressure during task execution. Conversely, the more DRAM used by the cache heap, the faster the access to the cached data. We propose a design where the JVM monitors memory pressure for execution and

caching, and dynamically adjusts the use of DRAM between the two parts.

TeraCache essentially relies on certain properties in JVM-based data processing frameworks, requiring a modified JVM that is aware of the framework notifications. Generally, *TeraCache* takes advantage of applications with the following properties:

- They create objects that can be grouped in sets of similar and preferably long lifetimes. Such groups of objects allow *TeraCache* to free regions in its *TeraCache* heap without the need for extensive GC. *TeraCache* relies on the runtime system to move these objects to its *TeraCache* heap, i.e., by annotating Spark caching code, which is transparent to the program and user.

- They create objects whose transitive closure does not cover the entire heap. To ensure safety, we avoid pointers from the *TeraCache* to JVM heap, by computing the transitive closure of references (i.e., all reachable objects) and move them in a single region in *TeraCache*. This implies that the transitive closure of such objects should not be the entire heap, otherwise using *TeraCache* will result in high overheads.

Note that although *TeraCache*'s approach could conceptually cope with mutable objects, e.g., by re-scanning their transitive closure after they are modified for new pointers, in Spark cached objects are immutable. This simplifies further the management of the *TeraCache* heap as a cache.

Recently, there is a fair amount of research activity towards extending DRAM in two directions: (1) the use of transparently caching NVMe (or NVM) devices to DRAM, an approach we follow as well, and (2) extending (but not caching) the system physical address space with byte-addressable NVM [37, 38] or block-oriented NVMe devices [7]. In both cases, employing a large heap occurs large GC overheads. Our approach shows that there are significant performance benefits in managing properly short-lived and a group of objects with similar properties in a co-design fashion by reducing GC overheads. Therefore, we believe that *TeraCache* could be used in both design alternatives.

Our approach, *TeraCache*, is a co-design approach for on-heap RDD caching over memory-mapped fast storage devices. *TeraCache* spans the Spark cache, JVM memory management and GC, and mmio. Contributions of this work are:

- It removes the need for *serdes* by caching only in a large memory-mapped heap, thus, allowing the JVM to access cached data directly using load/store operations.

- It reduces the frequency and length of GC cycles, despite keeping cached RDDs in the heap, by reclaiming RDD cached objects separately from the rest of the JVM heap.

- It dynamically adjusts the DRAM used for mmio and execution heap, to balance execution speed and I/O overhead.

We implement an early prototype of *TeraCache* that targets caching in Spark and investigate the dynamic resizing of DRAM between the two parts of the heap, evaluating its effectiveness on iterative ML workloads. Our evaluation shows performance improvement by up to 37% compared to the current state-of-the-art hybrid approach.

Chapter 2

Background

Spark [45] is a widely used framework for large-scale distributed analytics. Spark consists of one driver and multiple executor processes. The driver is the main process run by Spark users, which generates all tasks, while the executors are responsible for executing tasks of the driver. It supports acyclic data-flow similar to Hadoop MapReduce [14] but introduces convenient abstractions for in-memory computing. Spark targets machine learning and interactive mining applications that reuse transformed datasets across multiple iterations.

The main data abstraction in Spark is a resilient distributed dataset (RDD) [44]. RDD is an immutable collection of objects. Each RDD is split into multiple partitions, which is evaluated on the different nodes in the cluster across different stages. RDDs can survive in memory across stages, e.g., in a single iterative analytics job or across multiple interactive jobs. The Spark programming model exposes an API that supports caching RDDs in-memory (on-heap) in deserialized form or off-memory (off-heap) in serialized form to be materialized in subsequent stages.

At a low level, an RDD is a read-only collection of similarly-typed tuples partitioned across a cluster. Each RDD partition resides in a node as a hierarchical data structure consisting of a root RDD which references a Java array. The array, in turn, consists of a set of objects, e.g., key-value pairs. In the case of node failures, Spark maintains enough information to recompute an RDD partition.

The Spark driver is the main program that declares a series of transformations and actions over RDDs and submits these tasks to executors. Executors perform transformation and actions on the RDDs defined in the main program. Every transformation derives a new RDD from existing RDDs. Examples of transformation include map, reduce, join, or filter. An action usually computes a *measurable* quantity or a statistic (e.g., count). Spark follows lazy evaluation, i.e., transformations are not evaluated until an action is performed on the RDD.

Actions *materialize* an RDD that initially resides in memory. Spark analyzes the dependencies between RDD transformations to compute a *lineage graph* which allows for lazy evaluation and RDD recomputation (e.g., in case the node fails

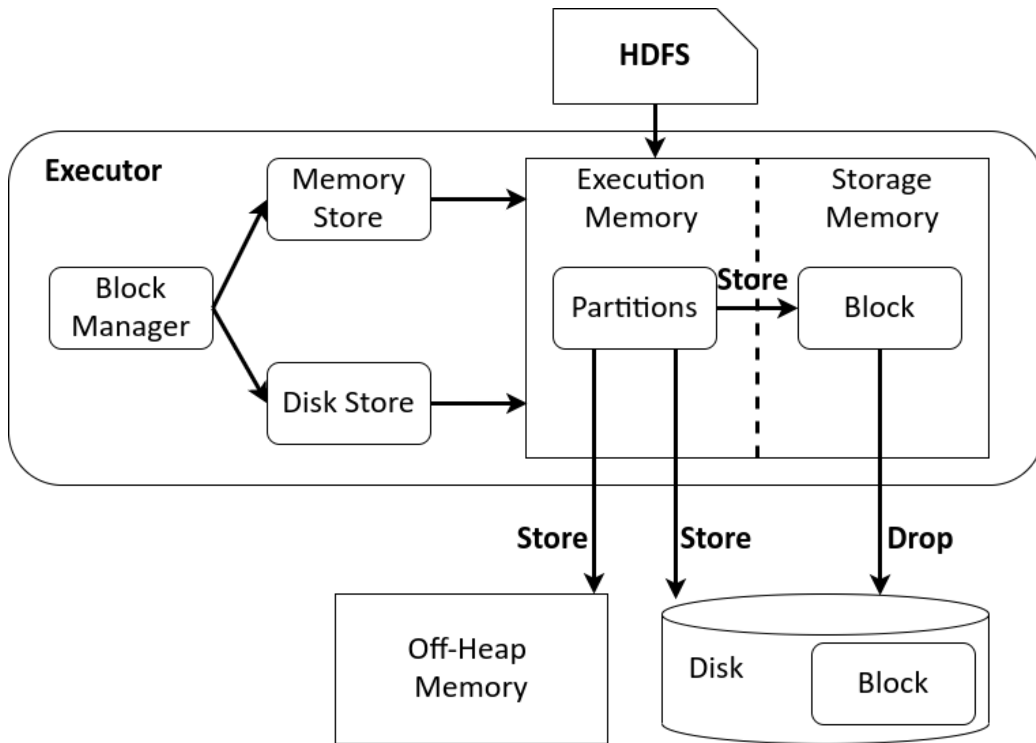


Figure 2.1: Internal processing of RDD caching in the Spark executor.

or runs out of memory). However, transformations only create an RDD object without filling the internal array with data objects. Once an RDD is materialized, it carries useful data in internal arrays. Recomputing the RDD is time-consuming, especially when the lineage graph has a few millions of nodes.

To avoid recomputation, Spark offers developers the flexibility of caching RDDs in memory or storage. In Spark's terminology, a *persist()* operation means RDD is stored in a user-defined storage level and a *cache()* operation means RDD is stored in main memory. The persist API is versatile [36]. Developers can specify a variety of storage levels for persisted RDDs, including memory in deserialized form or disk in serialized form or in both levels. Finally, persisted RDDs can reside on the JVM heap (on-heap) or outside of the heap (off-heap). The efficient management of RDDs on a contemporary server with multiple of memory and storage devices is challenging.

Figure 2.1 shows an internal architecture of executors in Spark. Executor memory is dynamically divided into execution memory and storage memory by the memory manager. Execution memory is used for storing temporary data during computation, and storage memory, for caching intermediate RDDs in an LRU cache. The Spark block manager manages storage memory and caches RDD partitions in the form of blocks. Spark supports the following storage levels for RDD caching.

NONE (NoCache) avoids caching intermediate data. Since the data is discarded immediately after their use, the application needs to recompute them when the same data is referenced again.

MEMORY_ONLY caches an RDD in-memory only. When the intermediate data caching is executed, the Spark block manager reserves a necessary amount of memory in storage memory and places each partition of the cached RDD in a block. Additionally, there is a storage level MEMORY_ONLY_SER which save the cached RDD in serialized to reduce the memory used by the cache.

OFF_HEAP holds the cached RDD in off-heap memory-only. When the intermediate data caching is performed, the Spark block manager serializes the partion and stores it outside the managed heap (off-heap), in memory. This storage level reduces the GC cost but increases the *serdes* overhead.

DISK_ONLY holds intermediate cached data in the local storage device only. When intermediate data caching is performed, the Spark block manager locates the cached RDD on disk in serialized form through the DiskStore interface. This storage level moves caching space on disk but introduces *serdes* overhead.

MEMORY_AND_DISK stores intermediate computed RDDs by using memory and disk. At first, this level tries to exploit memory preferentially and to save RDDs partitions as blocks in storage memory in de-serialized form. However if an RDD does not fit in memory, the Spark block manager will evict old block to the disk, in order to reclaim the necessary amount of memory for new blocks. The moved blocks will be chosen based on LRU until the required space is available. When Spark refers to the cached RDD located in the disk, the block manager checks the necessary space in storage memory. If there is space, the Spark block manager reads the blocks from the disk and stores them back to storage memory. The main benefit of this storage level is that it does not require users to select memory and disk and instead requires Spark to transparently use both.

Chapter 3

TeraCache Design

3.1 Motivation

Big data systems define the scope of object lifetime at the application level between well-defined operations. This work recognizes three cached RDDs objects characteristics that motivate new approaches for managing. We find these characteristics are not typical of objects in common Java applications:

Cached objects life cycle: In Spark, the cached object lifetime is defined between *persist* and *unpersist* Spark calls. However, the JVM is unaware of these operations, and as a result, manages cache objects as regular objects. Such groups of objects do not follow the generational hypothesis, based on which most newly allocated objects are more prone to become unreachable quickly, that state-of-the-art GCs are following. Because of this, GC performs unnecessary traversals over cached RDDs causing long tail-latency for applications. Therefore, Spark needs to provide hints to the JVM at runtime related to object life-span.

Group of similar lifetime objects: We observe that big-data analytics frameworks create objects that can be grouped in sets of similar and preferably long lifetimes. For example, cached data in Spark are groups of immutable objects that have similar life span. When these objects reach the end of their lifetime, the JVM can reclaim them in groups.

Cached objects immutability: RDDs are immutable, i.e., read-only. At low level, an RDD is an array of objects. By moving the entire transitive closure –i.e., performing a deep copy of the array– of an RDD to some memory region we know that a certain region can be marked as read-only and that any object in it will not point outside of the region. This characteristic removes the need for tracing the objects of cached RDDs that are kept in a different memory space during GC since they cannot point to objects in the heap. As a result, we reduce the GC

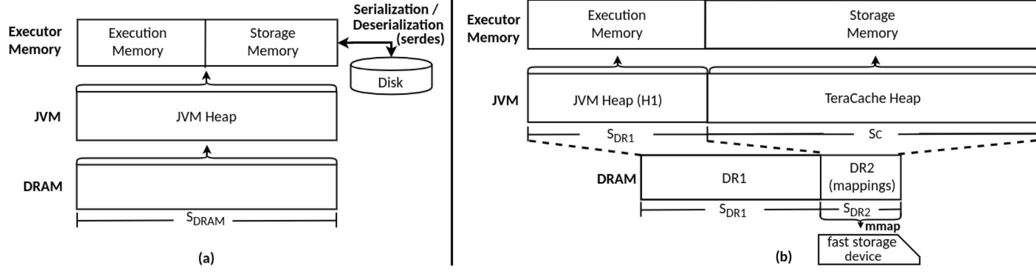


Figure 3.1: (a) Off-memory caching via *serdes*. (b) *TeraCache*, on-heap RDD cache over a memory-mapped fast storage device $S_{DR1} + S_{DR2} = S_{DRAM}$ ($\mathbf{S_{DRAM}}$=DRAM size, $\mathbf{S_{DR1}}$=DR1 size, $\mathbf{S_{DR2}}$=DR2 size, $\mathbf{S_{C}}$=Capacity of fast storage device).

overhead without compromising memory safety by moving large long-lived RDDs out of the standard heap.

3.2 System Overview

TeraCache exploits the characteristics of RDDs for efficient RDD management. A key goal of *TeraCache* is to leverage the large capacity of the fast storage devices to cache a large amount of intermediate data while eliminating *serdes* and reducing at the same time the GC cost. We start by briefly discussing the *TeraCache* architecture, and then describe each *TeraCache* component.

Figure 3.1(a) shows how Spark divides executor memory (top layer) into two logical parts: (1) execution memory, for storing temporary data during computation and (2) storage memory, for caching intermediate RDDs in an LRU cache. Each executor runs in a JVM instance and allocates memory from the JVM heap, which resides in DRAM. When an RDD does not fit in storage memory it gets serialized [2] and moved to the local storage device.

Our work takes advantage of this dual use of executor memory for computation and caching. We physically partition the JVM heap to serve these two roles. Then, we map each part of the JVM Heap to specific resources in the memory hierarchy, as follows (Figure 3.1(b)): (1) a JVM Heap (H1) is allocated exclusively on DRAM (DR1) which can be divided into generations [19], e.g., New and Old; and (2) a custom managed heap (*TeraCache* Heap) that contains all cached RDDs and its size is limited by device capacity (S_C).

The memory mappings for pages used by *mmap* reside in the remaining part of DRAM (DR2). S_{DR1} and S_{DR2} are dynamically adjusted by *TeraCache* using an adaptive policy at run-time. To do this, *TeraCache* requires a number of extensions in the Spark block manager and the JVM GC, as described below.

3.3 *TeraCache* heap allocation using *mmap* on NVMe

We modify the JVM to allocate an additional heap, memory-mapped onto a fast storage device, e.g., NAND-Flash SSD or NVMe, using Linux *mmap* [13]. Fast SSDs and NVMe devices, as opposed to HDDs, are amenable to mmio, due to the characteristics of the device and the access patterns produced as we described in section 4. We believe that using an optimized mmio path, such as FastMap [31], can further improve performance.

Another way to grow the JVM heap over a fast storage device to avoid *serdes*, would be to use the OS virtual memory system to use the NVMe as swap space [13]. Although this would enable storing very large JVM heaps in an NVMe, it cannot be used to target solely the RDD cache objects, and cannot avoid garbage collection of the resulting large heap. As *TeraCache* uses two separate heaps for execution and caching, in order to explicitly avoid GC in the cache, *mmap* is a better fitting mechanism to place the caching heap on the storage device. Figure 4.1(a) shows that being able to maintain separate heaps for the execution and caching parts of the heap, strictly use the NVMe for caching, and avoid GC in the cache, are all vital to performance. *TeraCache* yields up to 2x improvement compared to simply swapping a large, yet garbage-collected, single JVM heap onto the device.

3.4 *TeraCache* heap management avoids costly GC

Since the JVM is unaware of execution–storage memory separation, all objects get allocated on the JVM heap (middle layer of Figure 3.1(a)). This increases GC time, for two reasons: (1) cached RDDs are long-lived collection of objects and are managed by explicit persist and unpersist actions of Spark applications. Therefore, they rarely get collected and the GC spends a significant part of time traversing live objects; and (2) more GC cycles are required to reclaim enough space in execution memory since each GC cycle is able to free little memory due to long-lived cached RDDs. As cached objects lifetime is defined by how long they remain in the Spark cache, we can avoid GCs in *TeraCache*. Thus, our design uses a custom allocator to manage *TeraCache* and reduces object reclamation cost, as follows.

We augment the GC and make it aware of the differences between H1 and *TeraCache*. H1 is treated as a standard JVM heap and is collected using standard GC algorithms. *TeraCache* uses a custom region-based allocator [32] and conforms with the Java memory model [24]. We organize *TeraCache* into regions where each region is a collection of pages of memory and contains objects of the same RDD and all its reference objects. Consequently, *TeraCache* can free batches of objects with identical lifetimes allocated in the same region at once, similarly to Broom [17]. To maintain memory safety, we do not allow references from *TeraCache* to H1. All objects residing in *TeraCache* may only reference objects residing in *TeraCache*. We achieve this by migrating each RDD object and all objects reachable from it

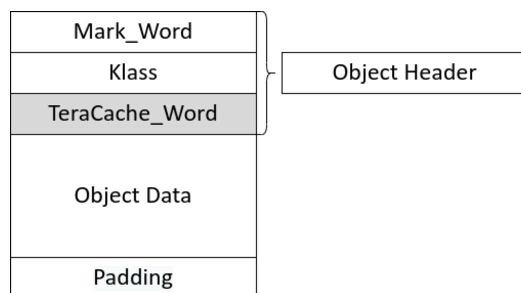


Figure 3.2: Java object layout in JVM. We use an additional *TeraCache* word in object header for objects that need to be moved to *TeraCache*.

to *TeraCache*. Then, the GC does not need to traverse *TeraCache* to identify live-objects in H1. RDDs are immutable and always safe to move to *TeraCache* without corrupting other objects.

3.5 Caching RDDs in *TeraCache*

Since RDD caching is explicitly managed by developers through the Spark RDD API [5], we introduce two JNI calls in the corresponding code in the Spark block manager to notify JVM for every caching operation. The Spark block manager is the Spark component within the executor that manages caching, serialization, data transfers, etc. Internally, the Spark block manager keeps a hash-map with all cached RDDs. When a persist operation occurs, the Spark block manager creates a new entry to the hash-map. Every entry in the hash map points to a partition of the cached RDD.

Because JVM is unaware of caching operations and cached data objects, we augment the garbage collector using JNI calls for every new entry in the hash map. Then we mark the object that represents the new entry to be moved in *TeraCache* in the next full GC cycle. We extend the object header using an extra word that shows which objects are marked to be moved in *TeraCache*. We move the data from H1 to TeraCache instead of directly allocating them because RDD objects will have already been created when the application requests the Spark block manager to cache them.

3.6 Batch Transfer of Objects to *TeraCache* during Full GC

We move the transitive closure of a cached object in the JVM-heap as a result to preserve the Java Memory Safety. H1 is treated as a standard JVM heap and is collected using standard GC algorithms. To maintain memory safety, we do not allow references from *TeraCache* to H1. We achieve this by migrating each

RDD object and all object reachable from it to identify live-objects in H1. RDDs are immutable and always safe to move to *TeraCache* without corrupting other objects.

To find out the incoming and outgoing references of each cached RDD object we need to traverse all the objects in the heap. This introduces extra overhead because we have to stop the application threads from taking over one-third of total application time at maximum parallelism. To avoid these overheads, we leverage the marking phase of the full GC to identify the transitive closure of the cached objects.

Full GC uses a four-phase mark-compact algorithm. In the first phase, the GC threads mark in parallel the live objects starting from the roots. Roots provide an entry point to the graph of live objects. GC threads perform a breadth-first search traversal (BFT) of the graph of live objects. During this phase, we use the *TeraCache* word from the header of each object to identify the cached objects and all their references. In the second phase, GC threads compact in parallel live objects by sliding them in a compacted area. In the case of cached data objects, we move these objects into an individual region in *TeraCache*. In the third phase the GC recursively update the pointers to the new addresses and in the fourth phase, the GC moves the live objects to the new location on the heap.

Because *TeraCache* is not garbage collected, we avoid having to mark&sweep *TeraCache* by ensuring there are no pointers across regions. For this reason, we avoid having objects that reside in *TeraCache* that point to other regions in *TeraCache* by unifying regions. During the precompaction and compaction phases, all objects marked to be moved to *TeraCache* will be allocated in a new region. Moreover, if any pointers to existing regions were discovered during the mark phase, all involved regions will be unioned. Note that region-merge is an $O(1)$ operation, as it only involves a single pointer in the region metadata structures. We merge regions to enforce region lifetimes to match the lifetime of the longest-living object in the region.

3.7 Reclamation of Cached Data Objects

TeraCache needs to reclaim space for objects that are not used anymore by the program. We should note that the cache capacity in *TeraCache* is defined by the size of the storage device, therefore, it cost-effective to instantiate huge caches over a few, high density NVMe devices in each server. Nevertheless, for long-running programs it is important to reclaim space and ensure that the cache is occupied by used objects. Similar to all other objects in Java, objects in *TeraCache* are free when there are no pointers from other objects that reside in the heap or in *TeraCache*. However, GC does not traverse the *TeraCache* to reclaim dead objects, thus *TeraCache* performs a self managed reclamation, as follows.

One approach to reclaim space is to return objects to the regular heap (H1) at unpersist operations: When the program performs an unpersist operation, this

is an indication that this object is not required any more. This is not guaranteed because the unpersist operation is an application hint and the object might still be referenced by other objects. By returning the object to H1, the GC will reclaim the corresponding space in a safe manner. This approach, however, incurs moving the object back to H1 and increases GC overhead in an unpredictable manner, depending on the lifetime and behavior of the user program.

Specifically, there are two options to re-allocate objects back to the JVM heap: (1) Allocate objects in the young generation, thus incurs extra GC overhead in the case where the young generation is full of objects, and (2) direct allocation of cached data objects to the old generation. If the old generation is full, then we need to enforce a full GC to free space in the heap for the cached data objects.

The approach used by *TeraCache* is to place objects in regions and free full regions in a safe manner as follows. First, we argue that objects in regions have similar lifetimes because of the way Spark and analytics frameworks operate, by typically dividing the dataset in segments (partitions of RDDs in Spark) and operating in a segment as a whole. Segments typically have sizes of several 10s of MBytes, e.g. 64 MBytes, and therefore, we expect that regions of size of a few MBytes will often contain objects that are freed at the same time. However, this is complicated by the fact that *TeraCache* places in a region the transitive closure of each object during persist operations.

To free a region, *TeraCache* needs to know that all objects in the region are not used (referenced). To identify such regions, *TeraCache* identifies references from H1 to H2 objects as follows.

For H1 references, *TeraCache* uses one "USED" bit per region. These bits are cleared at the beginning of each GC phase in H1. Whenever GC reaches a pointer to H2, stops traversal of this pointer to fence the cache from GC. At the same time, *TeraCache* sets the bit of the region pointed by this pointer. In this manner, the "USED" bit for each region captures all references from H1.

At the end of the GC, any *TeraCache* region not marked as live is certain to not be reachable from any object in the heap, nor from any object in other regions in *TeraCache*, and can thus be freed. This operation is also $O(1)$, as it only involves appending the list of pages in the freed region to the list of free pages in *TeraCache*, and freeing its metadata object.

At the end of each GC phase in H1, *TeraCache* can free all regions that have their "USED" bit set to 0. Freeing a region is also $O(1)$, as it only involves appending the list of pages in the freed region to the list of free pages in *TeraCache*, and freeing its metadata object. In addition, the overhead of maintaining "FREE" bits for the cache regions is low, given that *TeraCache* does not introduce additional passes over the data in H1 or H2. Therefore, *TeraCache* is able to reclaim cache space, relying on the characteristics of the objects placed in the cache, with practically no overhead in the common path.

3.8 Dealing with Class Loading, Finalizers

In Java, there are objects that implement finalizers. For these objects, before they are garbage collected, the Java runtime system gives them a chance to perform clean up operation. This step is known as finalization and is achieved through a call to the object's finalize method. Then these objects are reclaimed in the next GC cycle.

For this reason, we keep in every region a metadata list that contains which objects implement a finalize method. When we are going to reclaim the objects in these regions we traverse the finalizer list and enqueue these objects to the finalize enqueue. Then, we allocate these objects back to the JVM heap and finally reclaim the region by zeroing regions metadata.

Additionally, the class of each object is loaded in the method area during the allocation process of a new object in the JVM heap. The method area is placed outside of the JVM heap. Class gets unloaded when all the references to the class(and their instances) are removed and the class loader used is garbage collected. During the marking phase of the major GC, the garbage collector marks which Classes are alive and which need to be unloaded from the method area.

For objects that reside in *TeraCache*, we should keep their classes loaded in the method area. For this purpose, we keep in every regions metadata a class list that locates a pointer to each object class. We traverse for every region the class list and mark which classes are alive. Thus, we prevent the garbage collector from unloading classes that are used by objects in *TeraCache*.

3.9 Division of DRAM between DR1 and DR2

Figure 3.1(b) shows that *TeraCache* divides the physical DRAM (bottom layer) into two regions: (1) DR1 used for H1, and (2) DR2 used as a cache for the memory mappings of *TeraCache*. To reduce the time spent in GC during task execution, DR1 needs to be large enough to accommodate as much of the newly created objects as possible. At the same time, the size of DR2 determines the number of page faults causing *mmap* I/O, which will have a direct effect on the average access times for cached data. Ideally, we need sufficient space for H1 when tasks create new objects and sufficient space for *mmap* when tasks use cached data. However, since DR1 and DR2 share the physical DRAM, the larger the size of DR1 the smaller the size of DR2 and vice versa. Thus, we propose a mechanism that dynamically resizes DR1 and DR2.

In any case, the objective for H1 is to minimize the time we spend in GC by resizing H1 and the corresponding DRAM. Resizing H1 happens in the JVM, using an adaptive policy with existing resize operations. These operations can be called after a major GC when all threads have stopped running. They can also be called at other points during JVM operation, however, they interact with the memory allocators of individual threads and result in synchronization overheads. Resizing

dynamically the young and old generations need to stop the application thread from generating new objects and then the size of Eden will be recalculated. In our evaluation, we perform resize operations only after major GC steps. Future work should examine more fine-grain alternatives.

Resizing H2 requires merely issuing resize operations to FastMap. These are system calls, therefore, incur the corresponding overhead of a few 10s μs per call. FastMap typically reclaims clean pages, which does not imply I/O to the device. Fetching each new page will occur in the first page access and is not part of the resize operation. Therefore, in H2, the dominating metric is the amount of time spent for managing the mmio cache in DRAM, where data transfer is the largest part. We measure this overhead, which we call I/O time, in FastMap and make it available to the user runtime system for the purposes of optimizing DRAM allocation.

TeraCache needs to allocate DRAM between H1 and H2 in a manner that minimizes the sum of the two dominating metrics: GC time for H1 and I/O time for H2. We perform this optimization as follows.

TeraCache has the required metrics available by the corresponding subsystems and accessible at negligible overhead (reading counters and variables). We divide time into intervals of duration T. Although there is no clear value for T, T should correspond to a granularity that is required for adjusting DRAM between H1 and H2. In our case, we use T of 1s and we adjust DRAM in granularity of 10s of seconds, also corresponding to major GC intervals.

TeraCache calculates for each past interval the objective function (sum of GC time and I/O time). At “trigger” points it performs a binary search to identify how to divide DRAM as follows. It starts by allocating all DRAM H1 (technically the majority of DRAM to H1, since H2 needs a minimum amount of DRAM as well). In the next interval, it moves the DRAM boundary to the left (divide by two). If the objective function improves, it keeps moving to the left allocating more memory to H2. If the objective function becomes worse, it moves the boundary to the right allocating more memory to H1. The main issue, is whether this process will converge. Given that program behavior with the amount of memory is fairly monotonic, we believe that this process will typically converge quickly.

“Trigger” points are determined by *TeraCache* as follows. There is a trigger at the start of execution. Then, whenever one of the two components (GC time, I/O time) of the objective function becomes worse, *TeraCache* starts another trigger point. However, this criterion does not capture the case, where application behavior has changed and although there is room for further improvement, the components of the objective function do not change. Therefore, *TeraCache* also observes two additional metrics: the amount of memory freed during the last minor GC step and the number of items removed from H2. If any of these numbers is substantial (affect the overall performance) then *TeraCache* assumes that there is room for improvement, either by reducing the amount of memory used by H1 or by H2, and triggers the DRAM re-assignment process. Our evaluation shows that these heuristics for identifying trigger points are effective, however, we believe that

future work should examine alternative heuristics as well.

Overall, *TeraCache* dynamically observes program behavior at a granularity of T and re-partitions DRAM between H1 and H2 to minimize the dominating overheads for each of the two heaps: GC time for the compute heap H1 and I/O time for the customized cache heap H2.

3.10 Prototype Implementation

We implement a prototype of *TeraCache* based on the ParallelGC [19]. ParallelGC splits the JVM heap into two generations: (1) NewGen for keeping short-lived objects, and (2) OldGen for keeping long-lived objects. NewGen is divided into an Eden space and two equally divided survivor spaces. New objects are allocated in the Eden space, while objects that survive a minor GC get moved to the survivor spaces. Finally, objects surviving enough GCs and reaching a predefined tenure threshold are further moved to the OldGen.

In our prototype, we place the NewGen in DRAM (H1) and *mmap* OldGen on an NVMe device. This implementation uses OldGen as cache, containing all long-lived objects, including cached data. The Spark block manager does not notify the JVM when a cache operation is performed, however, the GC promotes cached data objects in OldGen after several minor GCs. To avoid GC on cached data we explicitly disable GC in OldGen.

Our prototype targets only caching and does not support the reclamation of cached RDDs during uncache operations and dynamic division of DRAM. Note that our prototype, allows for long-lived data other than the cached ones to slip in the Old Gen as well. To ensure that OldGen will primarily contain cached data and only a low number of non-cached data we set the tenured threshold of the GC to 25. Using this threshold we avoid in OldGen allocation of long-lived objects which are not related to cached data. As a result, only 5% of the allocated objects in OldGen are irrelevant long-lived data. This allows us to evaluate the GC time and I/O traffic, as they would be achieved by a more complete prototype of *TeraCache*.

Chapter 4

Evaluation

Using the *TeraCache* prototype implementation we perform a set of experiments to estimate:

1. The distribution of cached RDDs lifetime,
2. The overall performance benefit using *TeraCache* compared to MEMORY_AND_DISK storage level of Spark (hybrid),
3. The impact of using *TeraCache* on GC overhead, and
4. The effect of DRAM division between DR1 and DR2

Experimental Setup: We ran all experiments on a dual-socket server with two Intel(R) Xeon(R) E5-2630 v3 CPUs at 2.4GHz, with 8 physical cores and 16 hyper-threads each (32 total hyper-threads), 32 GB of DDR4 DRAM and CentOS v7.3, with Linux kernel 4.14.72. As storage device we use a PCIe-attached Samsung SSD 970 PRO with 500GB capacity. In our experiments, we use OpenJDK v8u250-b70 and Spark v2.3.0, using one Spark executor with 30 threads. We evaluate *TeraCache* against hybrid using KMeans (KM), LR, LgR, and SVM workloads from the

Config.	DR1 (GB)	DR2 (GB)	Virtual Memory TeraCache (GB)	Total Virtual Heap (GB)
A	2	30	420	422
B	4	28	420	424
C	8	24	420	428
D	16	16	420	436
E	24	8	420	444

Table 4.1: H1 size is equal to DR1. DR2 is the memory-mappings for *TeraCache* in DRAM. Total DRAM is the sum of DR1 and DR2. Total Virtual Heap is the sum of H1 and the *TeraCache* heap size.

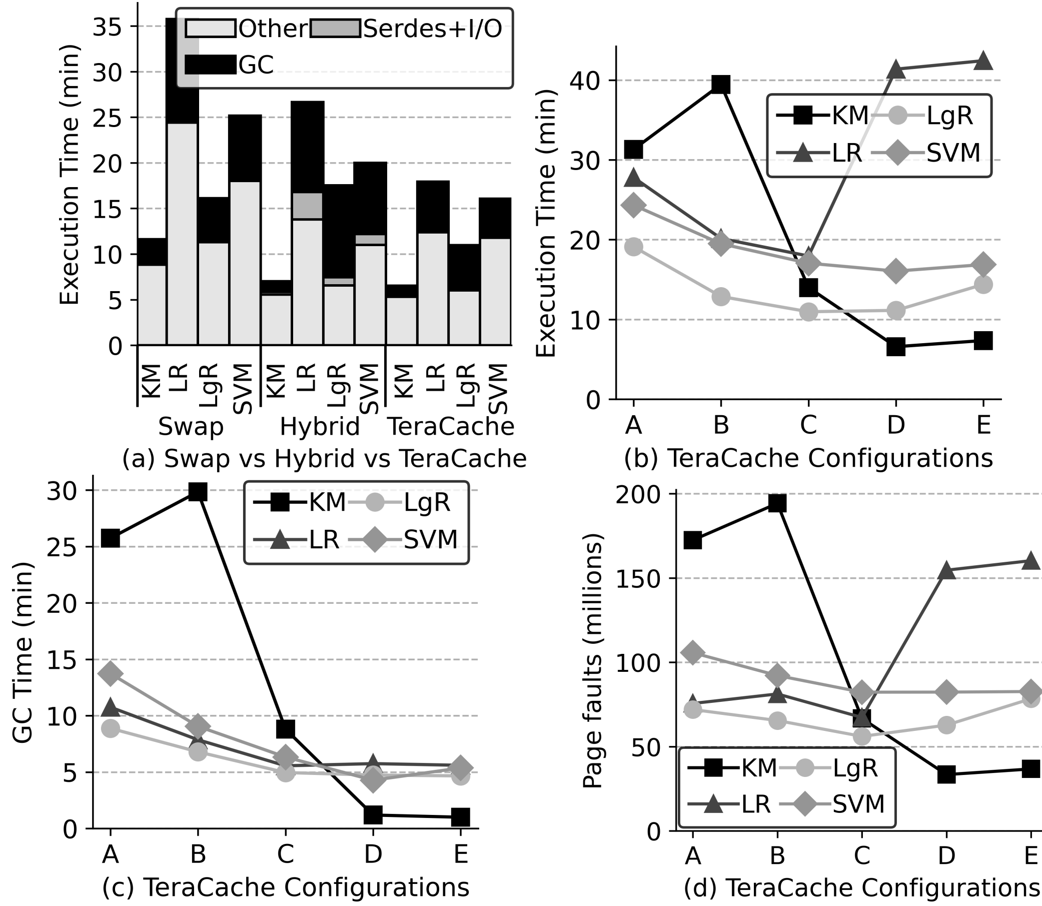


Figure 4.1: (a) Applications performance using Linux swap, hybrid and *TeraCache*. (b) Execution Time, (c) Total GC time, and (d) Total number of page faults for each *TeraCache* configuration with 64GB dataset.

Spark-Bench Suite [20]. Each workload ran for 100 iterations on a 64GB dataset. Also, for *TeraCache* we run each workload with the different configurations shown in Table 4.1, while for hybrid we use a 32GB heap that leverages 60% of the heap total heap space for on-heap cache and the full storage device as the off-heap RDD cache.

Heap demographics and cached RDDs lifetime: Table 4.2 shows the total allocation volume in TeraBytes (TB), the survival rate, and the volume of cached objects for three machine learning workloads using two datasets, 32 and 64 GB. Note that applications allocate rapidly and up to 2 TB of heap objects. This volume includes a large number of intermediate RDDs that are not persisted across iterations and some are never materialized. Thus young generation survival rates are low and less than 10%. This confirms prior findings that observe high mortality

Benchmark	Dataset (GB)	Allocation (TB)	Survival Rate (%)	Cached (GB)
LR		1.40	7.14	90
LgR	32	1.50	6.67	88
SVM		2.11	4.27	98
LR		3.13	5.75	182
LgR	64	2.58	5.81	160
SVM		3.79	4.75	188

Table 4.2: Spark heap demographics for popular machine learning benchmarks. *Allocation volumes are high and intermediate RDDs are less than 10% of total allocation but they need large amount of DRAM.*

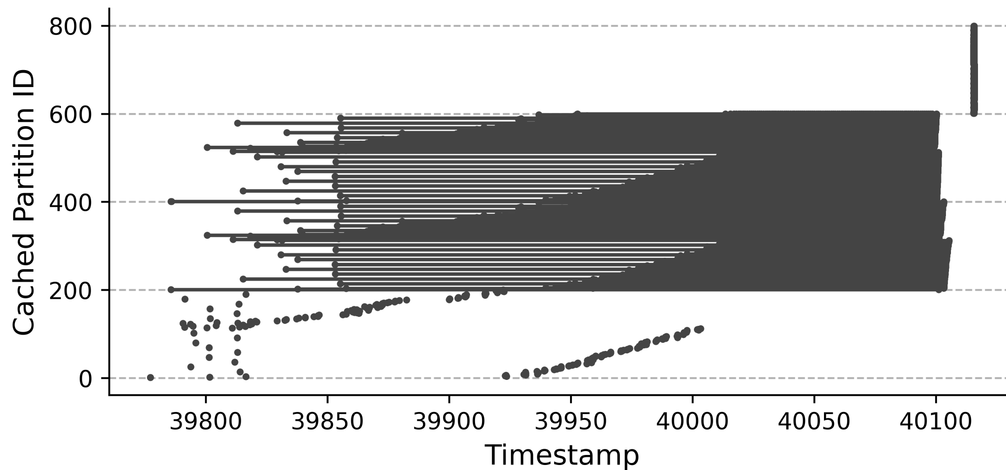


Figure 4.2: Cached data objects demographics

in the young generation. The biggest portion of the survived RDD volume is cached. As shown in the table, the volume of cached RDDs is up to 188 GB. This volume is high especially given current DRAM scaling trends and the size of the input dataset. Thus, running a Spark application could require 6× the size of a DRAM DIMM. This analysis shows that *TeraCache* could exploit the large capacity of fast storage devices by efficient caching RDDs while still avoiding the overheads of serialization and garbage collection.

Figure 4.2 shows the (active) lifetime of the persisted RDD partitions of SVM during the workload's execution. The Y-axis shows the RDD partition IDs and the X-axis shows the execution time. Each horizontal line signifies the life span of a single RDD partition. The left-most point on the horizontal line denotes the time the persist operation is called on this RDD partition, while the right most point denotes the (un)persist time. The rest of the points between the persist and the unpersist of the RDD partition indicate the accesses performed

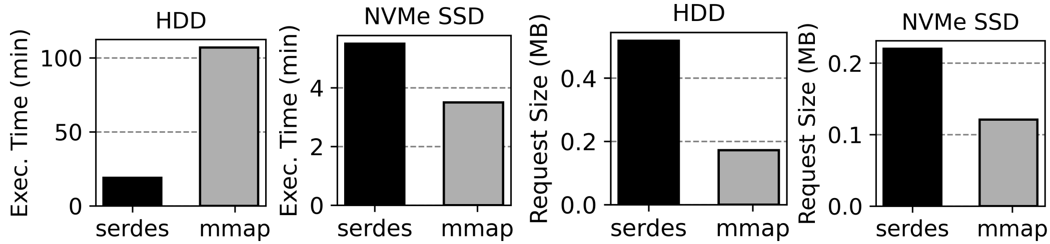


Figure 4.3: LgR on HDD vs NVMe SSD storage devices.

to it. We observe that the life spans of about half of the RDDs is approximately equal to the entire program duration, while the other half seem to be short-lived. Additionally, we observe that the long-lived RDDs are regularly accessed, with the accesses becoming more frequent close to the end of the program. This observation demonstrates why dynamically adjusting the distribution of the DRAM between the execution memory and the memory mapped part of the heap is important. In the beginning of application execution, many new short-lived objects are created while the cache is not accessed that much. After a while, cache accesses are becoming more frequent. As a result, giving more DRAM to the execution memory in the beginning and decreasing it after a while of the execution is expected to benefit the application.

Memory-Mapped I/O over fast and slow storage devices Figure 4.3 (left) shows the performance of LgR on HDD [6] and NVMe [3], for both *serdes* and *mmap*. In both cases we use a 18GB dataset and a single Spark executor using 8GB DRAM and 30 cores. The actual working set that needs to be cached is 10x the DRAM cache size. *mmap* produces small —due to the small (4KB) page size— and relatively random I/Os compared to *serdes*, as shown in Figure 4.3 (right), which shows the average request size. HDDs do not perform well for this access pattern. *Serdes* with 3x larger request size is more than 3x better than *mmap*, despite the high *serdes* CPU overhead. However, the NVMe achieves high throughput and low latency for small request sizes regardless of the access pattern [30], resulting in *mmap* performing 36% better compared to *serdes*. As a result, fast SSDs and NVMe devices, as opposed to HDDs, are amenable to mmio, due to the characteristics of the device and the access patterns produced.

Overall performance benefits using *TeraCache*: Figure 4.1(a) shows the total execution time of each benchmark when using hybrid (middle) and *TeraCache* (right). For *TeraCache* we plot the best performing configurations in Table 4.1 for the corresponding workloads i.e., configuration C for LR, LgR and configuration D for KM and SVM. For hybrid, we use the maximum heap size to allocate more RDDs on-heap to reduce the number of evictions in the storage device. We observe that *TeraCache* improves the overall performance by 7%, 32%, 37%, and 20% for KM, LR, LgR, and SVM, respectively compared to hybrid. To better understand

the source of the performance improvement we break down the execution time in Other, Serdes+I/O, and GC time.

Impact of *TeraCache* on GC overhead: As discussed in Section 3, Figure 4.1(a) shows that GC time decreases by 43%, 50%, and 45% for LR, LgR, and SVM respectively in hybrid. Part of the reduction is attributed to *TeraCache* not having to mark cached RDDs, while another part is attributed to *TeraCache* using the DRAM heap only for ephemeral objects thus performing fewer collections. In contrast, the original Spark keeps both short-lived objects and RDDs in the DRAM heap and only evicts serialized RDDs to the device. Specifically for KM, we observe a 5% increase in GC time. We attribute this increase to the fact that, in KM, tasks do not access large cached data, but instead create large amounts of shuffle data (short-lived objects) in H1. This, for smaller sizes of H1 (Figure 4.1 (c) configurations A and B), increases GC pressure and causes frequent collections. Finally, note that even in E (Figure 4.1(c)), where GC time is minimal, execution time is not necessarily minimum, as accessing *TeraCache* produces a large number of page faults (e.g., for LR), as discussed below.

Effect of DRAM division between DR1 and DR2: As discussed in Section 3, the DRAM division between DR1 and DR2 affects the overall performance of the applications. Figures 4.1(b), (c), and (d) show the execution time, the GC time, and the total number of page faults respectively for each workload, for configurations A-E in Table 4.1. KM has the minimum number of page faults using D but the GC time is higher by 16% than E. KM generates shuffle data between stages, requiring more space for the execution memory, as shown by the great improvement in total execution time between B and C. SVM has the minimum number of page faults and the lowest GC and execution time using D, while LR and LgR using C. Conversely, LR and LgR access the cached data frequently, and hence respond better to increasing the DRAM available for *mmap* pages. In general, variability in the execution time between configurations can be attributed to the application pattern. Benchmark stages with more accesses to cached data benefit more from more *mmap* pages in DR2, while stages that create many temporary objects benefit more from increasing H1. Thus, resizing DR1 and DR2 at runtime between stages is beneficial.

Chapter 5

Related Work

Related work falls in three categories: (1) hybrid memories for Java Heaps, (2) caching mechanism in Spark, and (3) region-based memory management for big data systems.

5.1 Hybrid Memory for Java Heap

Wang et *al.* propose Panthera [37], a semantic-aware GC for Big Data analytics working over hybrid memories. This work address two main challenges: (1) design a GC aware of hybrid memories and (2) design a GC aware of application sophisticated memory subsystems that execute different memory management tasks at the application level and is implemented for Spark framework. Panthera uses a hybrid Java Heap where allocates the new generation explicitly in DRAM and splits the old generation into a DRAM component and an NVM component. The placement of data objects is based on a static analysis where Panthera selects to allocate them on DRAM or NVM. Their evaluation shows that Panthera reduces the energy consumption by 32 - 52% incurring 1-9% extra time overhead. *TeraCache* is a co-design approach that uses application knowledge to explicitly and transparently manage cached objects and avoid unnecessary traversals over cached data on every GC cycle, significantly reducing CPU overhead. Additionally, *TeraCache* dynamically resizes the DRAM space used for memory-mapped I/O and execution memory in Spark according to job requirements, improving GC time and cache access time. Finally, the design of *TeraCache* is generic; it can be implemented on top of different GC.

Wu et *al.* [38] propose a new programming model for the developers to allocate objects directly in persistent memory. They introduce a persistent Java heap design and a crash consistent garbage collector to exploit Non-Volatile memories for persistence with higher performance. Java developers could manipulate persistent objects as if they were stored in a normal Java Heap However, real-world java developers will complain to re-implement their java application systems from scratch using Espresso API. Additionally, their work does not provide any solution

to avoid large GC pauses when the application uses very large heap sizes.

YakGC [27] identifies objects that are defined by the application that have same life-span. In their work tries to avoid unnecessary object copying with a region-based algorithm and avoid to traverse these data objects on every garbage collection cycle. However, YakGC is limited by the DRAM capacity and can not work with very large address space beyond physical DRAM.

NumaGiC [16] is a new GC for big data analytic frameworks on large memory NUMA machines. It considers data placement when performing object allocation or reclamation. However, as a generational GC, NumaGiC shares with modern GCs the problem of traversing long-lived data objects on every collection.

Deca [22] proposes lifetime-based memory management on top of Spark by analyzing variability of object sizes. Deca transparently decomposes and groups objects with similar lifetimes into byte arrays and reclaim their spaces in batch mode when their lifetime comes to an end. However, Deca needs to be re-implemented in order to be deployed in other analytics runtimes.

Xu et. *al.* [41] provide an experimental evaluation of GC on big data analytic frameworks. Popular big data analytics frameworks (*e.g.*, Apache Spark and Apache Flink) rely on garbage-collected languages, such as Java and Scala. Through their work, they use three different popular garbage collectors *i.e.*, Parallel, CMS, G1GC using four representative Spark workloads. Through their thorough evaluation analysis, the authors provide insights to researchers. We apply some of these insights to *TeraCache* design.

NG2C [11] creates multiple garbage collected dynamic generations. They provide an API to the user to locate group of similar lifetime objects in the same generation. As a result of that, the authors avoid object copying and heap fragmentation. However, *TeraCache* transparently locates cached data with similar lifetime into regions in an individual heap that is not garbage collected, reducing GC pause time.

5.2 Caching Mechanism in Spark

Neutrino [39] proposes a fine-grain, off-heap caching mechanism that performs *serdes* for blocks that belong to the same RDD, based on executor available memory at runtime. The authors use a dynamic programming algorithm to select the best caching strategy for each RDD partition. Neutrino requires a trace of the application execution in order to make these decisions, which we in our approach we avoid. Also, the authors support that the trace can be obtained from running the same application using a smaller input dataset. However, in some iterative Spark applications, the jobs are data-dependent since loop termination conditions can depend on data values.

LCS and LRC [15, 42] improve the management of on-heap memory caching, evicting RDD blocks that lead to minimum recomputation time in subsequent stages, without dealing with GC overhead. MemTune [40] dynamically tunes

executor caching space at runtime, based on data center workloads. MemTune provides task-level data prefetching with a configurable window size to overlap computation with *serdes* operation. Zhang *et al.* [46] modify the re-caching algorithm to avoid moving blocks from memory back to disk at the end of each task, reducing *serdes* cost. Tungsten [4] uses off-heap computation to eliminate *serdes*. However, Tungsten applies to known object schema (e.g. Spark SQL) while *TeraCache* can be employed for arbitrary schema object data. These works reduce *serdes* cost by managing cached data in-memory while paying significant GC cost to traverse cached data on every GC cycle. Instead, *TeraCache* completely removes *serdes*, providing direct access to the cached data and enables caches that exceed DRAM size, over memory-mapped fast storage devices. Finally, *TeraCache* reduces GC overhead by preventing GC traversals of cached data.

Coi et *al.* [12] explore the hypothesis that high-performance SSDs would reduce the average memory access latency by alleviating the saturated memory bandwidth. Iterative applications generate large amount of accumulated intermediate data in each iteration. They showed that utilizing high-performance SSDs improve the performance of data intensive applications due to their relatively high-bandwidth and capacity. Their experimental results show that the overall performance can improve by 23% on average using high-performance SSDs compare to the memory only approach.

5.3 Region-based memory management for big data systems

Broom [17] shows that big-data systems generate objects with the predefined lifetimes. Broom uses region-based memory management to locate in objects in shared regions improving GC time. The downside of Broom is that it adds an additional complexity to the end user to be aware of region management, while *TeraCache* leverages the cache architecture of the framework.

Burger *et al.* [9] propose that reclaiming individual objects in region-based memory allocators can be implemented efficiently. However, locating similar lifetime objects in regions *TeraCache* could reclaim entire objects by zeroing region metadata without freeing individual objects.

Facade [28] highlights that big data applications suffer from long GC cycles. It demonstrates a program transformation that divides application objects into objects stored on the garbage collected heap and stored off-heap in regions that are reclaimed at the end of every iteration. While Facade does not require changes to big data applications, developers must identify “boundary classes” and annotate their code which introduces extra overhead to the developers. On the other hand, *TeraCache* benefits from the Spark RDD API that implements *persist* and *unpersist* function calls to identify which objects are cached or uncached.

Chapter 6

Conclusions and Future Work

Spark applications often cache intermediate data, especially when performing iterative computations. However, the repeated serialization/deserialization of Spark RDDs creates significant CPU overhead that cannot currently be reduced without increasing GC overhead. We believe such overheads can be eliminated by extending the JVM heap over fast storage devices. We propose *TeraCache*, a co-design of the JVM and Spark that uses an on-heap RDD cache, memory-mapped over a fast storage device. *TeraCache* provides direct access over cached RDDs, removing both *serdes* and GC overheads for cached objects. Our preliminary results show that ML workload performance improves by up to 37% using *TeraCache* compared to *serdes*.

Our future work includes implementing the full and stable implementation of the *TeraCache*. Also, we would like to port *TeraCache* to other garbage collectors in OpenJDK, such as G1GC and ZGC. Additionally, we would like to extend the design of *TeraCache* to provide a persistent heap for the Spark executor. Spark executors could then load the cached RDDs when a failure is happened by loading the *TeraCache* heap. As a result of that, we could recover complex and long-running computations without the need to recompute cached RDDs from scratch. Finally, we expect that *TeraCache* can also improve performance for other frameworks that make use of large immutable object caches, such as Apache Flink.

Bibliography

[1] Intel Optane SSD DC P4800X Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-ssd-series/optane-dc-p4800x-series.html. Accessed: March 15, 2020.

[2] Kryo. https://github.com/EsotericSoftware/kryo. Accessed: March 15, 2020.

[3] SAMSUNG 970 EVO Plus NVMe M.2 SSD 500GB. https://www.cnet.com/products/wd-caviar-se-80gb/. Accessed: March 15, 2020.

[4] Project tungsten: Bringing apache spark closer to bare metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringingspark-closer-to-bare-metal.html, 2015. Accessed: May 25, 2020.

[5] RDD Programming Guide. https://spark.apache.org/docs/2.3.0/rdd-programming-guide.html, 2018. Accessed: March 15, 2020.

[6] Western Digital Caviar SE WD800JD. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-970-evo-plus-nvme-m-2-500gb-mz-v7s500b-am/, 2018. Accessed: March 15, 2020.

[7] Intel memory drive technology. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/intel-mdt-setup-guide.pdf, 2019. Accessed: March 15, 2020.

[8] NVMe SSD 960 PRO/EVO. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/ssd960/, 2019. Accessed: March 15, 2020.

[9] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, page 1–12, New York, NY, USA, 2002. Association for Computing Machinery.

[10] Rodrigo Bruno and Paulo Ferreira. A study on garbage collection algorithms for big data environments. *ACM Comput. Surv.*, 51(1), January 2018.

[11] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. Ng2c: Pretenuring garbage collection with dynamic generations for hotspot big data applications. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, page 2–13, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Yang W. Choi, I. Stephen and Kee Y. Early experience with optimizing I/O performance using high-performance SSDs for in-memory cluster computing. In *2015 IEEE International Conference on Big Data*, Big Data '15, pages 1073–1083, Washington, DC, USA, 2015. IEEE.

[13] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc., 2005.

[14] Apache Software Foundation. Apache hadoop, 2020.

[15] Yuanzhen Geng, Xuanhua Shi, Cheng Pei, Hai Jin, and Wenbin Jiang. Lcs: An efficient data eviction strategy for spark. *Int. J. Parallel Program.*, 45(6):1285–1297, December 2017.

[16] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 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, page 661–673, New York, NY, USA, 2015. Association for Computing Machinery.

[17] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingan, Derek Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, page 2, USA, 2015. USENIX Association.

[18] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management.* CRC Press, 2016.

[20] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on*

Computing Frontiers, CF '15, New York, NY, USA, May 2015. Association for Computing Machinery.

[21] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[22] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. Lifetime-based memory management for distributed data processing systems. *Proc. VLDB Endow.*, 9(12):936–947, August 2016.

[23] Alexander Makarov, V Sverdlov, and Siegfried Selberherr. Emerging Memory Technologies: Trends, Challenges, and Modeling Methods. *Microelectronics Reliability*, 52(4):628–634, April 2012.

[24] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. *SIGPLAN Not.*, 40(1):378–391, January 2005.

[25] Chris Mellor. Samsung drops 128TB SSD and kinetic-type flash drive bombshells. `https://www.theregister.co.uk/2017/08/09/samsungs_128tb_ssd_bombshell`, August 2017. Accessed: March 15, 2020.

[26] Onur Mutlu. Memory scaling: A systems architecture perspective. In *2013 5th IEEE International Memory Workshop, IMW 2013*, pages 21–25, May 2013.

[27] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 349–365, Savannah, GA, November 2016. USENIX Association.

[28] 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 *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 675–690, New York, NY, USA, 2015. Association for Computing Machinery.

[29] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, July 2011.

[30] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 537–550, Denver, CO, June 2016. USENIX Association.

[31] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing Memory-mapped I/O for Fast Storage Devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '20, USA, July 2020. USENIX Association.

[32] Nikolaos Papakonstantinou, Foivos S Zakkak, and Polyvios Pratikakis. Hierarchical parallel dynamic dependence analysis for recursively task-parallel programs. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 933–942. IEEE, 2016.

[33] K. Parat and A. Goda. Scaling trends in nand flash. In *2018 IEEE International Electron Devices Meeting (IEDM)*, pages 2.1.1–2.1.4, December 2018.

[34] David Reinsel, John Gantz, and John Rydning. DataAge 2025 - The Evolution of Data to Life-Critical. Seagate, November 2018.

[35] Shubhanshi Singhal, Pooja Sharma, Rajesh Kumar Aggarwal, and Vishal Passricha. A global survey on data deduplication. *Int. J. Grid High Perform. Comput.*, 10(4):43–66, October 2018.

[36] Apache Spark. Rdd programming guide, 2020.

[37] 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 *the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, June 2019.

[38] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 70–83, New York, NY, USA, 2018. ACM.

[39] Erci Xu, Mohit Saxena, and Lawrence Chiu. Neutrino: Revisiting memory caching for iterative data analytics. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'16, pages 16–20, Berkeley, CA, USA, 2016. USENIX Association.

[40] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu. Memtune: Dynamic memory management for in-memory data analytic platforms. In *2016*

IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 383–392, May 2016.

[41] Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. An experimental evaluation of garbage collectors on big data applications. *Proc. VLDB Endow.*, 12(5):570–583, January 2019.

[42] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. Lrc: Dependency-aware cache management for data analytics clusters. *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[43] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, April 2012. USENIX Association.

[44] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[45] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10. USENIX Association, June 2010.

[46] K. Zhang, Y. Tanimura, H. Nakada, and H. Ogawa. Understanding and improving disk-based intermediate data caching in spark. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2508–2517, December 2017.

[47] Xuechen Zhang, Ujjwal Khanal, Xinghui Zhao, and Stephen Ficklin. Making sense of performance in in-memory computing frameworks for scientific data analysis: A case study of the spark system. *Journal of Parallel and Distributed Computing*, 120:369–382, 2018.