TeraCache: Efficient Caching over Fast Storage Devices

Iacovos G. Kolokasis\textsuperscript{1,2}, Anastasios Papagiannis\textsuperscript{1,2}, Foivos Zakkak\textsuperscript{3}, Shoaib Akram\textsuperscript{4}, Christos Kozanitis\textsuperscript{2}, Polyvios Pratikakis\textsuperscript{1,2}, and Angelos Bilas\textsuperscript{1,2}

\textsuperscript{1}University of Crete
\textsuperscript{2}Foundation of Research and Technology Hellas (FORTH), Greece
\textsuperscript{3}Red Hat, Inc.
\textsuperscript{4}Australian National University
Spark Caching Mechanism

- Stores the result of an RDD
- Essential when an RDD is used across multiple Spark jobs
- Caching avoids recomputation and reduces execution time
- Effective for iterative workloads (e.g., ML, graph processing)
- How much data do we need to cache?

<table>
<thead>
<tr>
<th>Storage Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_ONLY</td>
</tr>
<tr>
<td>MEMORY_AND_DISK</td>
</tr>
<tr>
<td>DISK_ONLY</td>
</tr>
<tr>
<td>OFF_HEAP</td>
</tr>
</tbody>
</table>

Source: https://spark.apache.org/docs/latest/rdd-programming-guide.html
Increasing Memory Demands!

- Analytics datasets grow at high rate
  - Today ~50ZB
  - By 2025 ~175ZB

- Typical deployments use roughly as much DRAM as the input dataset

- Typically cached data is even larger than the input dataset

Source: Seagate – The Digitization of the World
Cached Data Size Matters

- In-memory caching needs a lot of DRAM
- DRAM density difficult to increase
- Fast storage (NVMe) scales to TBs/device
- Spark already uses fast storage for cached data – However, at high cost
# Dilemma: On-heap vs Off-heap NVMe Caching

## Executor Memory
- **Execution Memory**
- **Storage Memory**

### Serialization / Deserialization

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-heap Cache</td>
<td>No Serialization</td>
</tr>
<tr>
<td>Off-heap Cache</td>
<td>Low GC</td>
</tr>
</tbody>
</table>

Can we avoid Serialization and reduce GC?
Cached Objects Behave Differently

Spark App

Dataset

Create RDD

Persist

Operations

Unpersist

Java Heap

GC
Cached Objects Behave Differently

Spark App

Dataset → Create RDDs → Persist

Operations → Unpersist

GC

Java Heap
Cached Objects Behave Differently

- GC between **persist-unpersist** extremely wasteful
- GC scans all objects in the heap
Cached Objects Behave Differently

Spark App

Dataset

Create RDDs

Persist

Operations

Unpersist

Java Heap

- GC reclaim cached RDDs after unpersist
Our Approach: Treat Cached Objects Differently

- Objects in JAVA follow generational hypothesis

- Opportunity: Nomadic hypothesis observation

- Spark cached objects are
  - Long-lived: Used across multiple Spark jobs (cache)
  - Intermittently-accessed: Long intervals without access (NVMe)
  - Grouped life-times: RDD objects leave the cache at the same time (unpersist)

- Place cached objects in a special heap
TeraCache: Introduce a Second JVM heap on NVMe

- Execution Heap remains as a garbage collected heap
  - Maintains the JVM heap for execution purposes

- The second TeraCache heap has two significant advantages

- No GC: Use persist/unpersist semantics to avoid GC

- No Serialization/Deserialization: Use memory-mapped I/O
TeraCache Design Overview
TeraCache: Design Overview

- **Spark Executor**: Execution Memory, Storage Memory
- **JVM**: JVM heap, TeraCache
- **DRAM**: DR1, DR2
- **NVMe SSD**: mmap()
Spark Knocks on the JVM Door

- Spark notifies JVM for RDD caching
  - At persist/unpersist operations
- Add new TeraFlag word in JVM objects
- JVM creates new object, sets TeraFlag

Spark Application

```
rdd.persist()
```

Spark Runtime

- Store RDD to Storage Memory
- Notify JVM to mark RDD object

JVM

```
JVM heap
```

TeraCache
Spark Knocks on the JVM Door

Spark Application
- `rdd.persist()`

Spark Runtime
- Store RDD to Storage Memory
- Notify JVM to mark RDD object

JVM
- Move to TeraCache during next full GC

- Spark notifies JVM for RDD caching
  - At persist/unpersist operations
- Add new TeraFlag word in JVM objects
- JVM creates new object, sets TeraFlag
TeraCache Design: Avoid GC
How to Avoid GC in TeraCache?

- **Disallow** backward pointers to Heap
- **Move** transitive closure in TeraCache
How To Avoid GC in TeraCache?

- **Disallow** backward pointers to Heap
- Move **transitive closure** in TeraCache
- **Allow** forward pointers from Heap
- Objects in TeraCache **do not move**
- **Fence GC** from following forward pointers
Organize TeraCache in Regions

- Objects that belong to the same RDD have similar life-time
- Organize TeraCache in regions
  - Place objects in regions based on life-time
  - Dynamic size of regions
- Bulk free
  - Reclaim entire region
Bulk Free Regions

- To provide **correct** and **bulk** free
  - **Allow only** pointers within regions
  - Merge regions with crossing pointers when objects move to TeraCache

- Keep a bit map with live regions
  - Track reachable regions from JVM heap in every GC

- During GC marking phase identify active regions
  - Mark the bit array if there is a pointer from the JVM heap to a TeraCache region
TeraCache Design: Avoid Serialization
No Serialization→Memory Mapped I/O

- MMIO allows **same data format** on memory and device
- No explicit device I/O - Only accesses using load/store
- Linux Kernel already supports required mechanisms for MMIO
- We use FastMap [USENIX ATC'20]: Optimize scalability of Linux MMIO
Competition for DRAM Resource

- Execution Memory must reside in DRAM
  - A lot of short-lived data
  - We need large DRAM

- Cached objects are accessed as well
  - E.g., Iterative jobs reuse cached data
  - We need large DRAM

- Can we statically divide DRAM between the heaps?
Dividing DRAM between Heaps

- KMeans (KM)-jobs produce more short-lived data
  - More minor GCs
  - More space for DR1

- Linear Regression (LR)-jobs reuse more cached data
  - More page faults/s
  - More space for DR2

- Dynamic Resizing of DR1, DR2
  - Based on page-fault rate in MMIO
  - Based on minor GCs
Preliminary Evaluation
Early Prototype Implementation

- We implement a prototype of TeraCache based on ParallelGC
  - Place New Generation on DRAM
  - Place Old Generation on fast storage device
  - Explicitly disable GC on Old Generation

- Remaining to be implemented
  - Cached RDDs reclamation
  - Dynamic DR1/DR2 resizing

- Evaluation
  - GC overhead
  - Serialization overhead
TeraCache Improves Performance by 25%

- Compared to Serialization: **TC better up to 37%** (on average 25%)
- Compared to GC + Linux swap: **TC better up to 2x**
TeraCache Reduces GC Time by up to 50%
Conclusions
TeraCache: Efficient Caching over Fast Storage

- Spark incurs high overhead for caching RDDs

- We observe: Spark cached data follow a **nomadic hypothesis**

- We introduce TeraCache which both reduces GC and eliminates serialization by using two heaps (**generational, nomadic**)

- We improve performance of Spark ML workloads by 25% (avg)

- Currently we are working on the full prototype
Thank you for your attention

This work is supported by the EU Horizon 2020 Evolve project (#825061)
Anastasios Papagiannis is supported by Facebook Graduate Fellowship

Iacovos G. Kolokasis
kolokasis@ics.forth.gr
www.csd.uoc.gr/~kolokasis
Feedback

Your feedback is important to us.
Don’t forget to rate and review the sessions.