Say Goodbye to Off-heap Caches! On-heap Caches Using Memory-Mapped I/O

Iacovos G. Kolokasis[1], Anastasios Papagiannis[1], Foivos Zakkak[2], Polyvios Pratikakis[1], and Angelos Bilas[1]

[1]University of Crete & Foundation of Research and Technology Hellas (FORTH), Greece

[2]University of Manchester (Currently at Red Hat, Inc.)

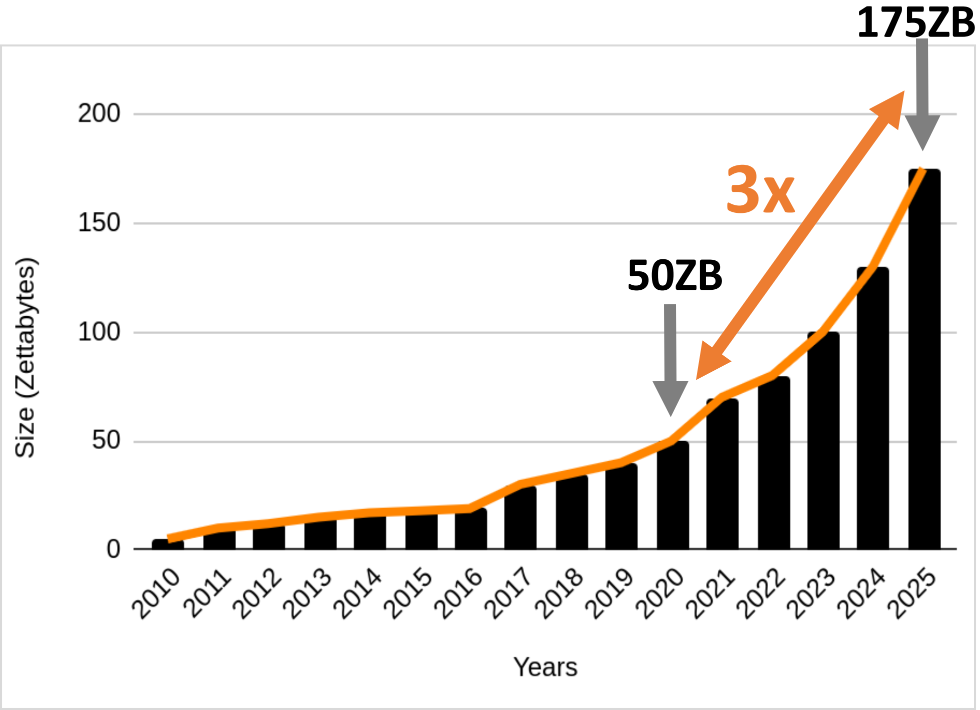
Outline

- Motivation
- TeraCache design for multiple heaps with different properties
  - How we reduce GC time?
  - How we grow TeraCache over a device?
- Evaluation
- Conclusions

Increasing Memory Demands!

- Big data systems cache large intermediate results in-memory
  - Speed-up iterative workloads

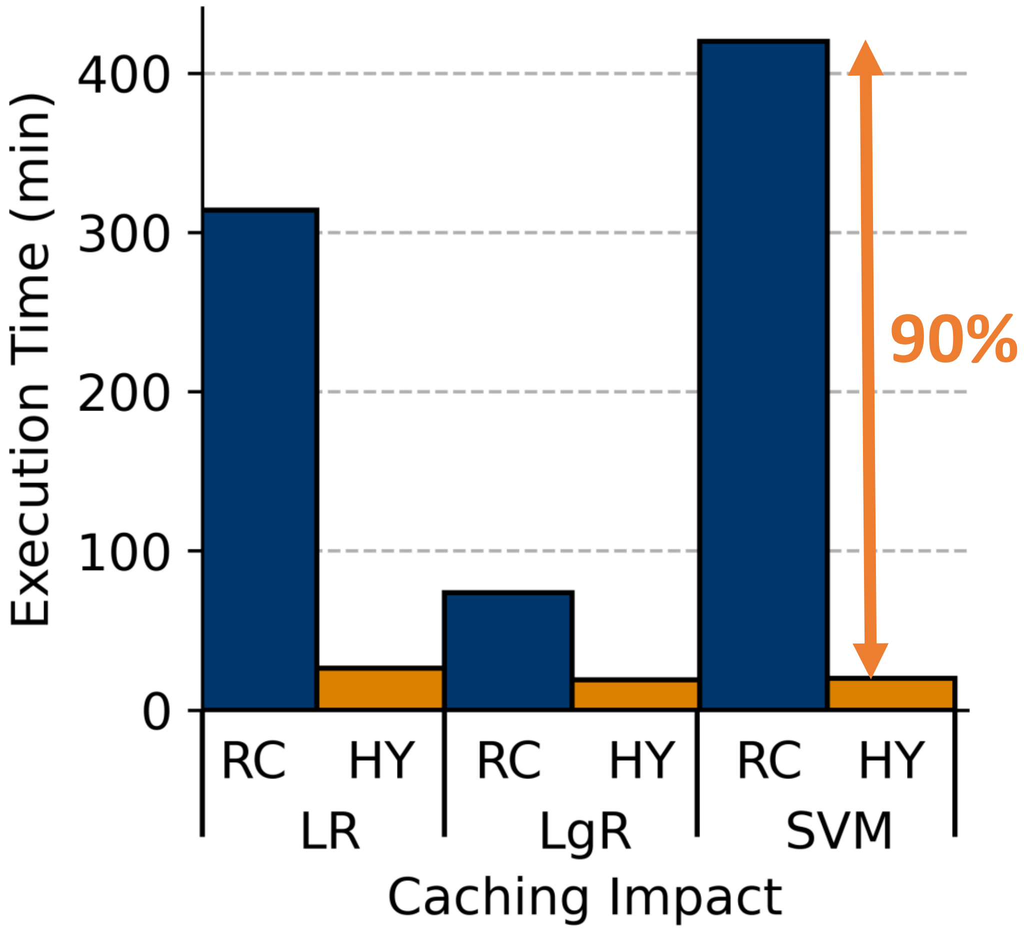
- Analytics datasets grow at a high rate
  - Today ~50ZB
  - By 2025 ~175ZB



[Source: www.seagate.com | Seagate]

- Big data systems request TBs of memory per server

Spark: Caching Impacts Performance

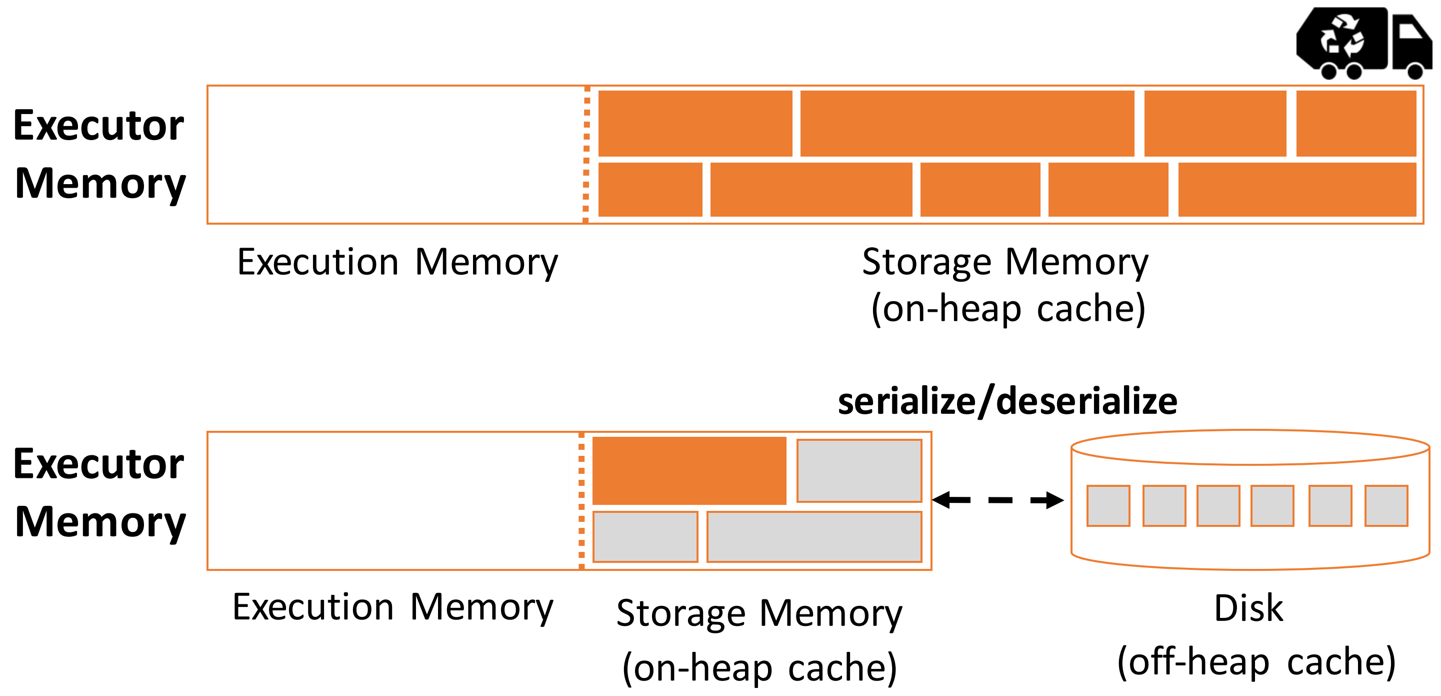


- Jobs cache intermediate data in memory
- Subsequent jobs reuse cached data
- Caching reduces execution time by orders of magnitude
- Naively, caching data needs large heaps which implies a lot of DRAM

Caching Beyond Physical DRAM

- DRAM capacity scaling reaches its limit [Mutlu-IMW 2013]
- DRAM scales to GB / DIMM
- DRAM capacity is limited by DIMM slots / servers
- NVMe SSDs scale to TBs / PCIe slot at lower cost
- **Already Today:** Spark uses off-heap store on fast devices

Between a Rock and a Hard Place! GC vs Serialization Overhead



	Pros	Cons
On-heap Cache	No Serialization	High GC
Off-heap Cache	Low GC	High Serialization

Merge the benefits from both worlds!

Outline

- Motivation
- TeraCache design for multiple heaps with different properties
  - How we reduce GC time?
  - How we grow TeraCache over a device?
- Evaluation
- Conclusions

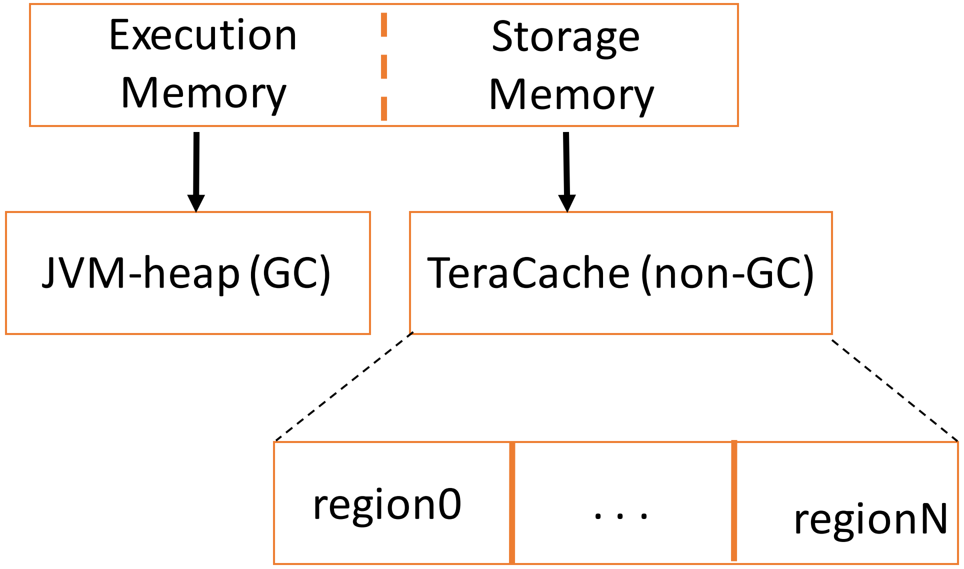
Different Heaps for Different Object Types

- Analytics computations generate mainly two types of objects
  - Short-lived, (**runtime managed**)
  - Long-lived, similar life-time, (**application managed**)

- JVM-heap on DRAM which is **garbage collected**
  - Locate short-lived objects
  - For computation usage (task memory usage)

- TeraCache-heap which is **never** garbage collected
  - Contains group of similar life-span objects (e.g., cached data)
  - Grow over a storage device **(no serialization)**

Split Executor Memory In Two Heaps

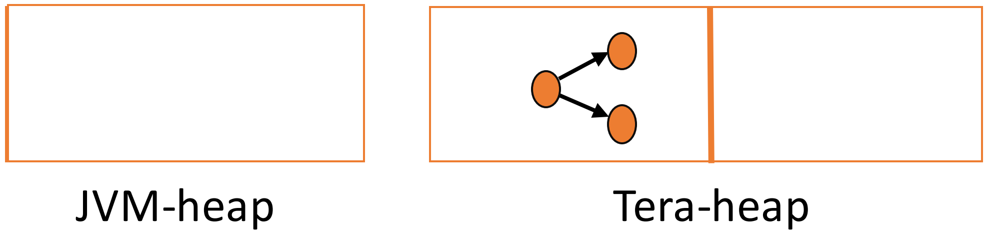


Executor Memory

- JVM-heap (**GC**)
- TeraCache (**non-GC**)

Organize TeraCache in regions

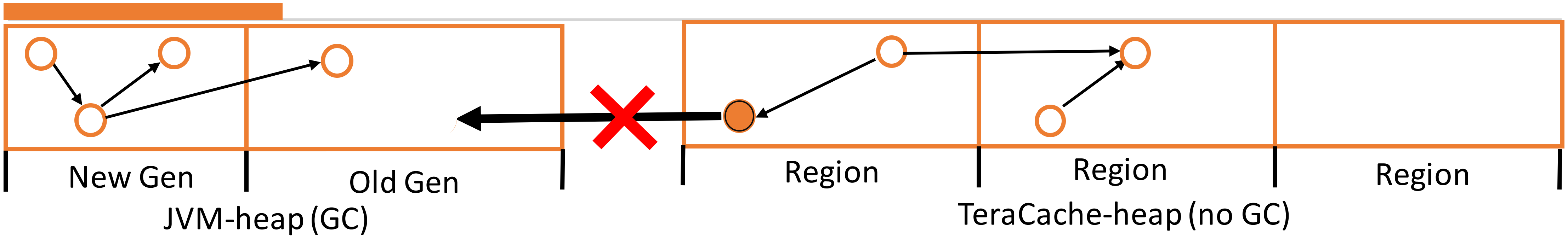
- **Bulk free:** Similar life-time objects into the same region
- Dynamic size



We make the JVM **aware** of cached data

- Spark notifies JVM
- Finds the transitive closure of the object
- Move and migrate object into a region

We Preserve JAVA Memory Safety

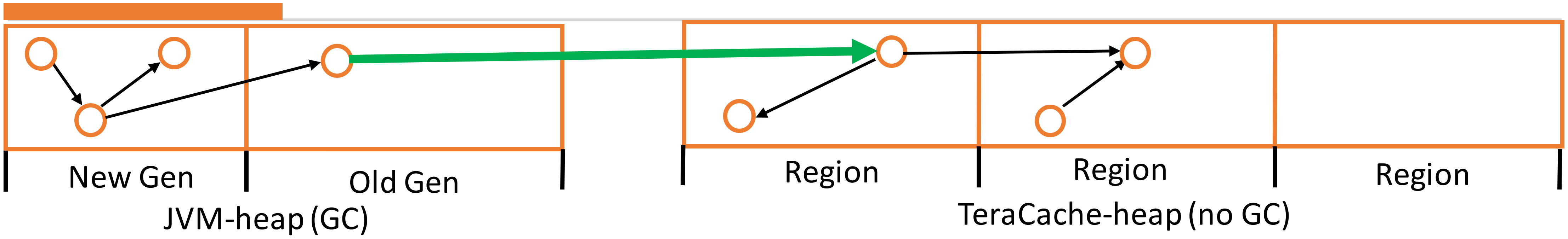


Avoid pointer corruption between objects in two heaps

No backward pointers: TeraCache → JVM-heap

- Stop GC to reclaim objects used by TeraCache objects
- Move transitive closure of the object

We Preserve JAVA Memory Safety



Avoid pointer corruption between objects in two heaps

No backward pointers: TeraCache → JVM-heap

- Stop GC to reclaim objects used by TeraCache objects
- Move transitive closure of the object

Allow forward pointers: JVM-heap → TeraCache

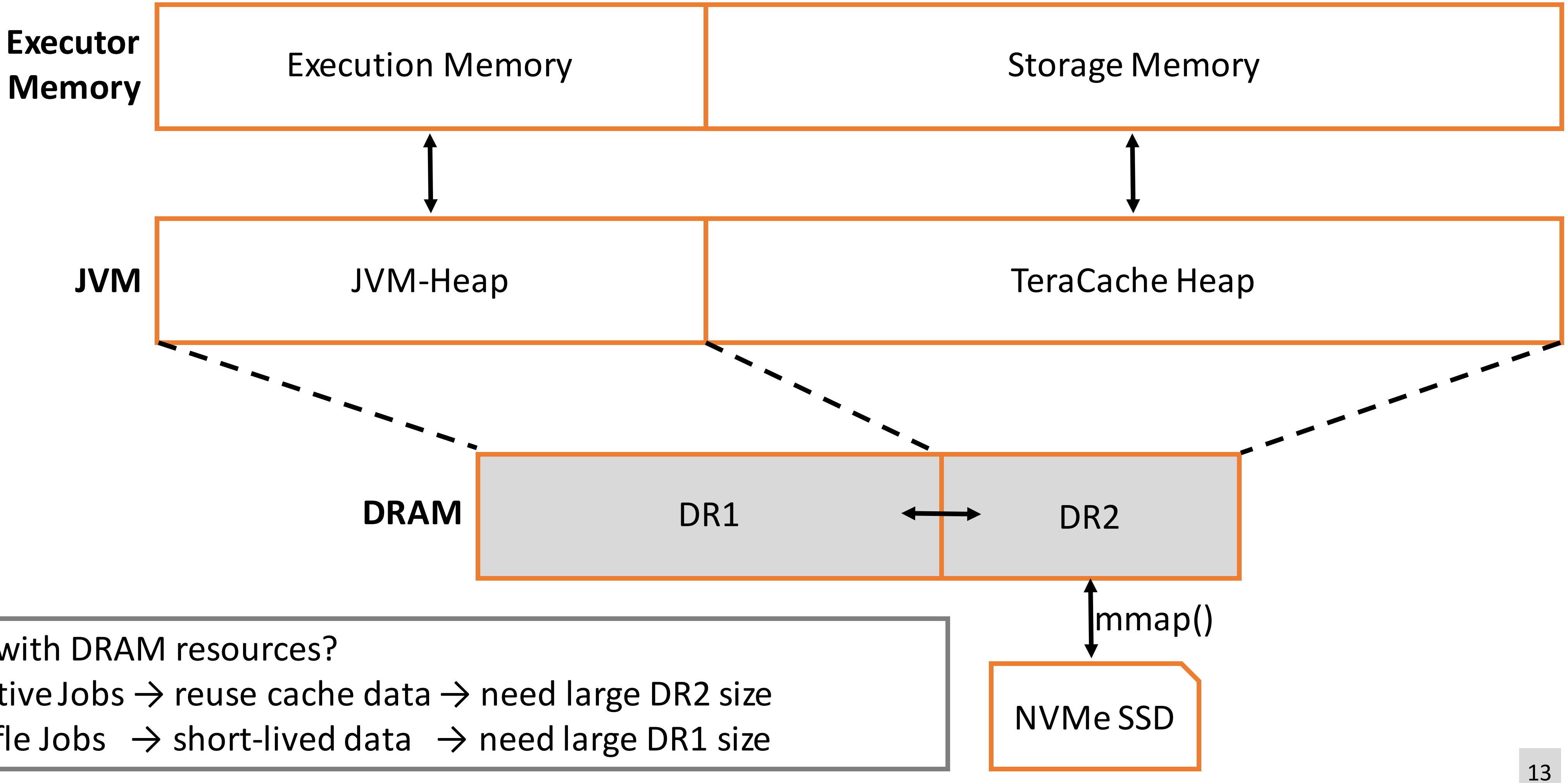
- But stop GC to traverse TeraCache

Allow internal pointers: TeraCache↔TeraCache

Outline

- Motivation
- TeraCache design for multiple heaps with different properties
  - How we reduce GC time?
  - How we grow TeraCache over a device?
- Evaluation
- Conclusions

Dividing DRAM Between Heaps

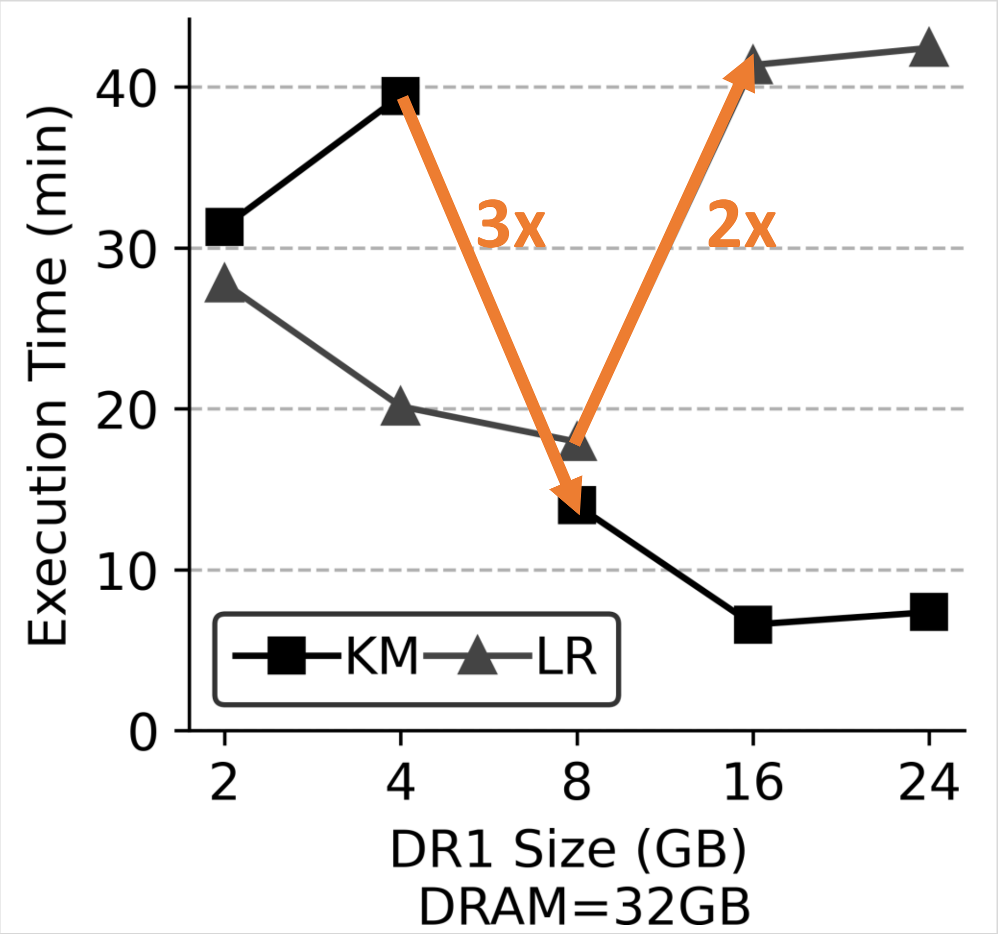


How to deal with DRAM resources?

- Iterative Jobs → reuse cache data → need large DR2 size
- Shuffle Jobs → short-lived data → need large DR1 size

Deal With DRAM Resources For Multi-Heaps

- KM-jobs produce more short-lived data
  - More **minor GCs/s** →more space for DR1
- LR-jobs reuse large size of cached data
  - More **page faults/s**→ more space for DR2
- We propose dynamic resizing of DR1, DR2
  - Based on page fault rate in MMIO
  - Based on Minor GCs



Outline

- Motivation
- TeraCache design for multiple heaps with different properties
  - How we reduce GC time?
  - How we grow TeraCache over a device?
- Evaluation
- Conclusions

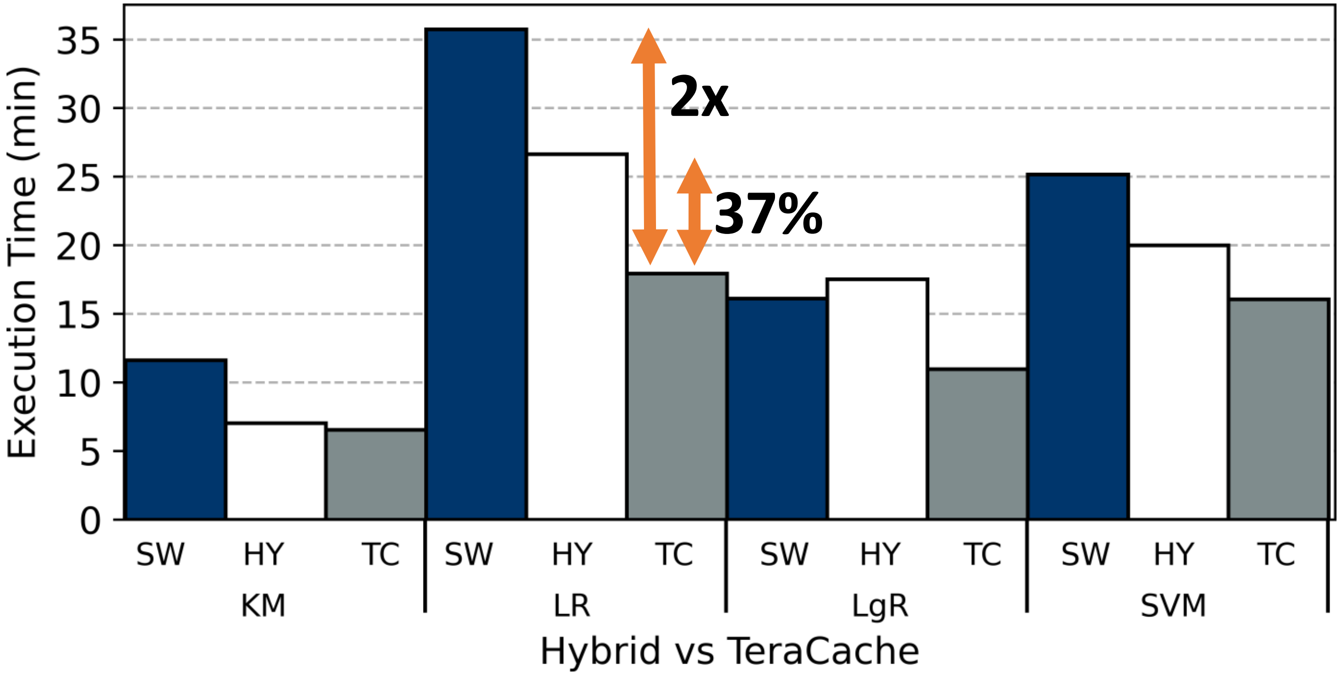
Prototype Implementation

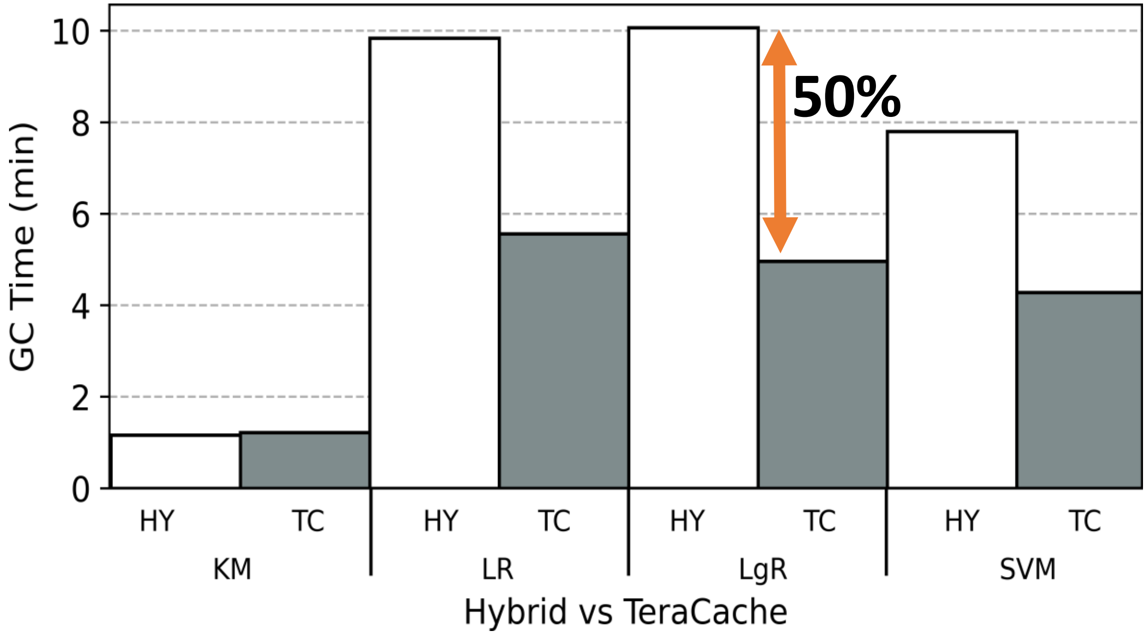
- We implement an early prototype of TeraCache based on ParallelGC
  - Place New generation on DRAM
  - Place Old generation on the fast storage device
  - Explicitly disable GC on Old generation

- Evaluate
  - GC overhead
  - Serialization overhead

- Not support for reclamation of cached RDDs and dynamic resizing

Preliminary Evaluation





- TC improves performance up to 37% LR (on average 25%)
- TC improves performance up to 2x compared to Linux swap (LR)

- TC improves GC up to 50% LGR (on average 46%)

Conclusions

- TeraCache: A JVM/Spark co-design
    - Able to support very large heaps
    - Reduces GC time using two heaps
    - Eliminates serialization-deserialization
- Dynamic sharing of DRAM resources across heaps
- Improves Spark ML workloads performance by 25% on average
- Applicable to other analytics runtimes

Contact

Iacovos G. Kolokasis

kolokasis@ics.forth.gr

www.csd.uoc.gr/~kolokasis

Institute of Computer Science (ICS)

Foundation of Research and Technology (FORTH) - Hellas

• • •

Department of Computer Science, University of Crete