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

Iacovos G. Kolokasis,* Anastasios Papagiannis,* Polyvios Pratikakis,* *and* Angelos Bilas*
Institute of Computer Science (ICS), Foundation for Research and Technology – Hellas (FORTH), Greece
{kolokasis, apapag, polyvios, bilas}@ics.forth.gr

Foivos Zakkak†
Red Hat, Inc.
fzakkak@redhat.com

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-memory caching of intermediate results. However, off-heap caching requires the serialization and deserialization (*serdes*) of data, which add significant overhead especially with growing datasets.

This paper 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.

1 Introduction

Analytics applications often use ML [16] 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 [35] caches large intermediate results of complex computation pipelines and reuses them in each step. Intermediate results are stored as Resilient Distributed Datasets (RDDs) [34] in an LRU cache.

Figure 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 (we report the average of three runs; deviation was minimal and is omitted). Caching RDDs in both memory and disk (hybrid)

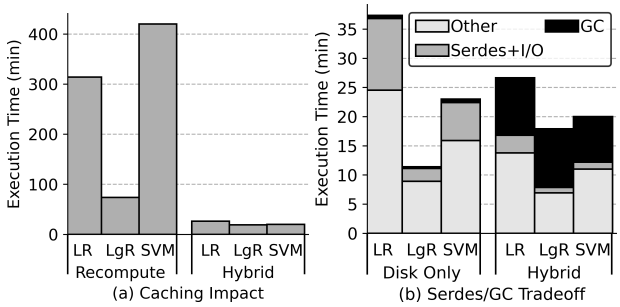

Figure 1: (a) Hybrid caching outperforms on-demand recomputation. (b) Disk-only caching incurs high *serdes* overhead and lower GC time, whereas hybrid caching exhibits the reverse behavior.

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 [26, 27], while DRAM scaling reaches its limits [13, 17, 20]. Cached data increasingly exceed physical DRAM size, making workloads prone to recomputing intermediate results at each step. Therefore, current practice is to use fast storage devices to increase Spark's effective cache size for intermediate RDDs [37]. NAND flash storage devices such as SSD and NVMe block devices [1], as well as NVMs, have higher density and capacity than DRAM [21, 25]. SSD and NVMe devices scale to terabytes per PCIe slot [19] 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 few μs and perform hundreds of thousands IOPS for both reads and writes.

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

*Also with the Department of Computer Science, University of Crete.
†Work performed while employed by The University of Manchester.

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 incurs high *serdes* overhead. According to Zhang *et al.* [36], *serdes* rather than disk I/O dominates overhead. Figure 1(b) shows the performance impact of off-heap RDD caches on LR, LgR, and SVM, when storing RDDs only off-heap (disk). 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(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 [9, 32] 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 an 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, prevents GC on cached data, and is inline with device technology trends and server power limitations.

Next, 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 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.

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. The main benefits of *TeraCache* 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.

2 *TeraCache*: Caching Over a Device Heap

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. Figure 2(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 disk.

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 2(b)): (1) a JVM Heap (H1) allocated exclusively on DRAM (DR1) and which can be divided into generations [14], 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 garbage collector, as described below.

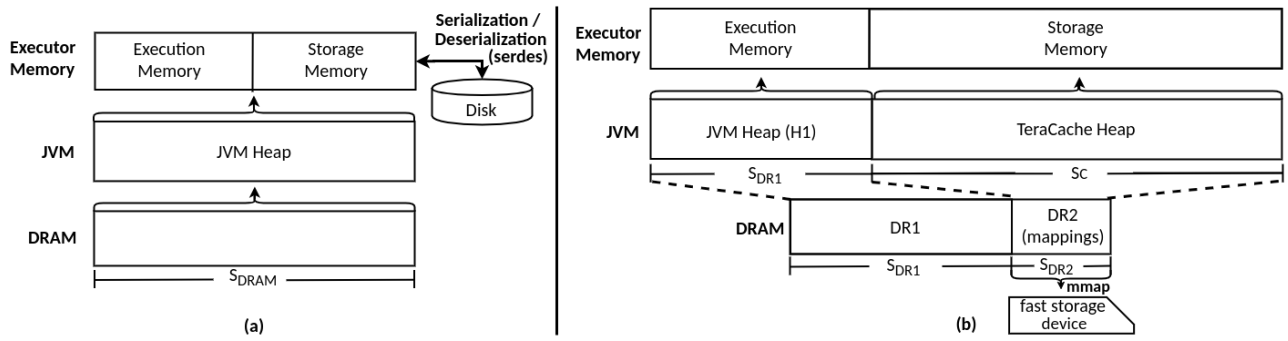

Figure 2: (a) Off-memory caching via *serdes*. (b) *TeraCache*, on-heap RDD cache over a memory-mapped fast storage device $S_{DR1} + S_{DR2} = S_{DRAM}$ ($\mathbf{S_{DRAM}}$=DRAM size, $\mathbf{S_{DR1}}$=DR1 size, $\mathbf{S_{DR2}}$=DR2 size, $\mathbf{S_C}$=Capacity of fast storage device).

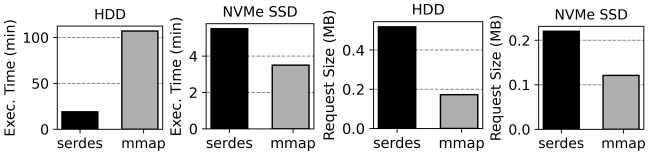

Figure 3: LgR on HDD vs NVMe SSD storage devices.

2.1 Design Challenges

***TeraCache* heap allocation using *mmap* on NVMe is practical:** 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* [10]. 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. Figure 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 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 [22], resulting in *mmap* performing 36% better compared to *serdes*. We believe that using an optimized mmio path, such as FastMap [23], 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 use the NVMe as swap space [10]. 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(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.

***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 2(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 garbage collector 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 the cached objects' life-time is clearly 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 garbage collector 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 [24] and conforms with the Java memory model [18]. Specifically, we organize *TeraCache* into regions corresponding to RDDs. A region is a collection of pages of memory and contains objects of the same RDD. Consequently, *TeraCache* can free batches of objects with identical lifetimes allocated in the same region at once, similarly to Broom [12]. 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 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.

Caching RDDs in *TeraCache*: Since RDD caching is explicitly managed by developers through the Spark RDD API [5], we introduce two Java annotations, `@cache` and `@uncache`, to annotate the corresponding code points in the Spark Block Manager. The Block Manager is the Spark component within the executor that manages caching, serialization, data transfers, etc. The annotations we introduce are syntactic metadata that communicate to the JVM that an RDD is cached or uncached by Spark. At a `@cache` annotation, which implies the caching of an RDD, *TeraCache* performs a traversal of the RDD data similar to the marking phase of a mark-sweep GC, marking all objects that can be reached from it and migrating them to an appropriate *TeraCache* region. We move the data from H1 to *TeraCache* instead of directly allocating them there, as RDD objects will have already been created when the application requests the Spark Block Manager to cache them. Respectively, at an `@uncache` annotation, the JVM can reclaim the RDD block and its space from *TeraCache*. In addition, to annotating the user facing API we also annotate, with `@uncache` the Spark Block Manager function that handles eviction of RDDs –based on an LRU policy when the Storage memory becomes full– to reclaim RDDs when *TeraCache* reaches its capacity limit.

Division of DRAM between DR1 and DR2: Figure 2(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 the 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.

We use two metrics to determine whether such resizing is needed. For DR1, we measure the frequency of *minor GCs/s*, since a higher frequency indicates that the application needs more heap space (DR1). Similarly, for DR2 we measure the rate of *pagefaults* for a single memory-map since a high frequency of page faults indicates that more space needed for memory mappings. Overall, if the rate of *pagefaults* increases and minor GC frequency is low, then we decrease the size of DR1 and increase the size of DR2 and vice versa. Section 3 shows that dynamic resizing of DR1 and DR2 is necessary.

2.2 Prototype Implementation

We implement an early prototype of *TeraCache* based on the ParallelGC [14]. 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 reach a predefined tenuring threshold are further moved to the OldGen.

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 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.

In our prototype, we place the NewGen in DRAM (H1) and *mmap* OldGen onto 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 garbage collector 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 reclamation of cached RDDs during uncache operations. 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*.

3 Evaluation

Using the *TeraCache* prototype implementation we perform a set of experiments to estimate: (1) the overall performance benefit using *TeraCache* compared to hybrid (baseline), (2) the impact of using *TeraCache* on GC overhead, and (3) the effect of DRAM division between DR1 and DR2.

Evaluation 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

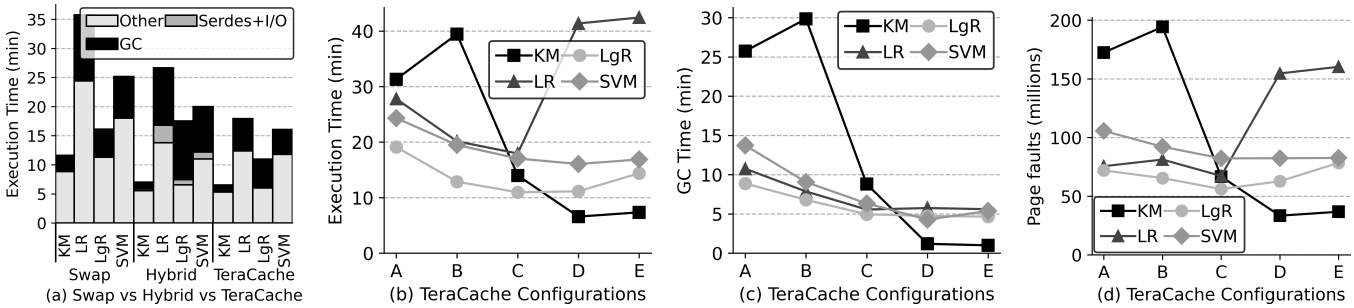

Figure 4: (a) Applications performance using Linux swap, hybrid and *TeraCache*. (b) Execution Time, (c) Total GC time, and (d) Total number of page faults for each *TeraCache* configuration with 64GB dataset.

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 Spark-Bench Suite [15]. Each workload ran for 100 iterations on a 64GB dataset. Also, for *TeraCache* we ran each workload with the different configurations shown in Table 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.

Overall Performance Benefits Using *TeraCache*: Figure 4(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 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 2, Figure 4(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 (c) configurations A and B), increases GC pressure and causes frequent collections. Finally, note that even in E (Figure 4(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 2, the DRAM division between DR1 and DR2 affects the overall performance of the applications. Figures 4(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 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.

4 Discussion

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

- They create objects that can be grouped in sets of similar and preferably long lifetimes. For example, Spark workflows with iterative computations use exist long-lived objects, such as accumulated records and shuffle data. Such groups of objects allow *TeraCache* to free regions in its *TeraCache* heap without the need for extensive garbage collection. Our early *TeraCache* prototype 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 H1, 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 [28, 29] 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 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.

5 Related Work

Caching in Spark: Neutrino [30] 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. LCS and LRC [11, 33] 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 [31] 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.* [36] 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.

JVM heaps over byte-addressable NVM: Espresso [29] takes advantage of the larger capacity of NVM by introducing a new programming model to persist long-lived objects. Espresso does not provide a GC policy to avoid frequent traversals of long-lived objects, increasing GC overhead, especially in big data analytics frameworks. Panthera [28] places YoungGen in DRAM and divides OldGen into DRAM and NVM. Panthera is integrated in ParallelGC and traverses all cached RDDs on every major GC resulting in significant performance degradation. Both Espresso and Panthera remove *serdes* overhead by increasing heap size at the expense of GC overhead. However, they are agnostic to application specific characteristics, such as caching. Panthera statically divides DRAM between the two heaps, regardless of job requirements. *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 garbage collector.

Region-based memory management for big data systems: Broom [12] show that big-data systems generate objects with the predefined life-times. They use region-based memory management to locate in objects in shared regions improving GC time. The downside of Broom is that add an additional complexity to the end user to be aware of regions, while *TeraCache* leverages the cache architecture of the framework.

6 Conclusions

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*. We expect that *TeraCache* can also improve performance for other frameworks making use of very large immutable object caches (e.g., Apache Flink).

7 Acknowledgments

We thankfully acknowledge the support of the Evolve Project (Grant Agreement N^o 825061), funded by the European Union Horizon 2020 Research and Innovation Programme. Anastasios Papagiannis is also supported by the Facebook Graduate Fellowship. Finally, we thank Yannis Sfakianakis, Giorgos Xanthakis, Fotis Nikolaides, and the anonymous reviewers for their insightful comments and constructive feedback.

References

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

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.