Thoth in Action: Memory Management in Modern Data Analytics*

Mayuresh Kunjir
Duke University
mayuresh@cs.duke.edu

Shivnath Babu
Duke University
shivnath@cs.duke.edu

ABSTRACT

Allocation and usage of memory in modern data-processing platforms is based on an interplay of algorithms at multiple levels: (i) at the resource-management level across containers allocated by resource managers like Mesos and Yarn, (ii) at the container level among the OS and processes such as the Java Virtual Machine (JVM), (iii) at the framework level for caching, aggregation, data shuffles, and application data structures, and (iv) at the JVM level across various pools such as the Young and Old Generation as well as the heap versus off-heap. We use Thoth, a data-driven platform for multi-system cluster management, to build a deep understanding of different interplays in memory management options. Through multiple memory management *apps* built in Thoth, we demonstrate how Thoth can deal with multiple levels of memory management as well as multi-tenant nature of clusters.

1. INTRODUCTION

We are witnessing an explosion in the number of data-processing platforms: Hadoop, Spark, HBase, Cassandra, Kafka, Storm, Flink, Presto, and others. Some key observations about these platforms are:

- **JVM-based:** Most of these platforms run on the Java Virtual Machine (JVM) and are written in JVM-based languages like Java, Scala, and Clojure. The JVM is recognized industry-wide as a developer-friendly, stable, efficient, and secure system.
- **Container-friendly:** These platforms are invariably run in multi-tenant environments where resources are allocated and isolated using *containers*, e.g., using technologies like Yarn, Mesos, and Docker.
- **Memory-intensive:** Jim Gray is credited with predicting that "Memory is the new disk." In-memory data storage is increasingly the focus in these platforms.

*This research is supported by NSF grants CNS-1423128 and IIS-1423124.

Proceedings of the VLDB Endowment, Vol. 10, No. 12

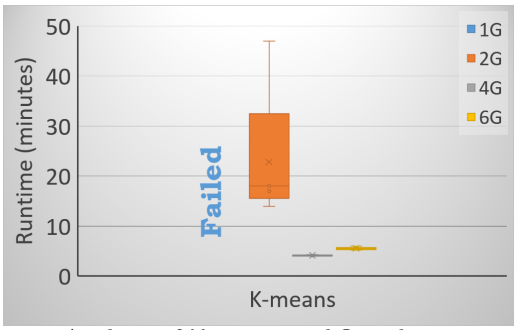

Figure 1: Analysis of K-means workflow showing interplay in different user goals by varying memory per container

The central premise of this paper is that modern data analytics will increasingly be done in memory on shared-nothing clusters using JVMs running inside containers. Given this, it is high time for a systematic empirical study to understand issues faced by these platforms such as:

1. The interplay of memory-management decisions made at multiple levels such as: (i) the resource-management level across various containers allocated on a cluster, (ii) within a container, (iii) at the application level for caching, aggregation, data shuffles, etc., and (iv) inside the JVM.
2. The impact of the JVM's *garbage collection (GC)* on response time, throughput and performance predictability.
3. The resource-usage overheads of serializing and deserializing data for use in the JVM heap, as well as the use of off-heap storage to possibly reduce these overheads.
4. The chances of failure due to "out-of-memory exceptions."
5. The impact of the many tuning knobs at various levels to control memory management, including the number and sizes of memory pools, how to serialize/deserialize, how to perform GC, and others.
6. Multi-tenant nature of the platforms necessitates guarantees on fair access to system resources, including memory. This requirement conflicts with memory-management decisions made for a single tenant.

Often times, cluster administrators simply over-allocate memory as a safeguard. An internal study carried out at a popular social networking company showed that memory is over-allocated by 350% in a production workload. This is partly due to developers using default resource allocation settings for lack of a better knowledge about actual memory requirements.

A simple example suffices to bring out the complexity that application developers and database administrators of these modern platforms face. Figure 1 plots runtimes of a sample K-means application workflow run on a 'Spark on Yarn' cluster of 10 nodes. The application is provisioned with one container on each node. Memory allocated to this container is varied across test configurations. The application caches a dataset of vertices in memory to be used across iterations. With only 1GB memory, containers run out of memory while caching this dataset. After multiple container failures, Yarn kills the application. When the memory is increased to 2GB, some containers run out of memory. But they are replaced with new containers which re-do the failed tasks and the application eventually succeeds. In this process though, not only does the performance suffer, but the predictability also takes a hit: The number of containers that fail due to insufficient memory is hard to predict due to data distribution and scheduling factors. A memory of 4GB proves sufficient for a predictable and a much better performance. In this setting, the entire data fits in the available memory on each node. If memory is over-provisioned, like in the case of 6GB memory in the graph, the runtime suffers since the JVM's memory allocation and de-allocation (GC) costs go up with the heap size.

This example showcases the interplay among four key performance metrics: (i) response time, (ii) efficiency of resource usage, (iii) reliability, and (iv) performance predictability. We do a systematic study of how different memory management options impact the performance metrics. Thoth is a data-driven cluster management platform we have developed [8]. It collects a wide variety of profiling data from a data analytics cluster, consolidates it to a system-agnostic format, and allows multiple visualization and system management *apps* over the data. This demonstration uses the Thoth platform to understand memory management interplays on a 'Spark on Yarn' cluster.

Further, Thoth is used in optimizing multi-tenant workloads sharing memory as a data cache in a *fair* manner. We have developed a multi-tenant cache management solution, ROBUS [7], which maximizes the throughput of a multi-tenant cluster using cache for performance speedups while being fair in terms of the speedups obtained by tenants.

Related Work. The community is either working on understanding time bottlenecks [9] or on a focussed analysis of a single level of memory management (e.g. VisualVM [5], Ganglia [3], Memory Pressure Interrupts [6]). We are doing a comprehensive study of memory management encompassing all levels which incorporates various interplays in tuning options as well as in performance metrics.

2. MEMORY MANAGEMENT

2.1 Memory Pools

Memory in modern data analytics clusters is managed at four different levels: Resource manager level, Container level, Application level, and JVM level. Each level maintains multiple memory pools, each with a specific purpose. While some of the pools are controlled by configuration options, the others are often unmanaged. Figure 2 depicts various memory pools on a node in a data analytics setup.

Resource Manager level. At the topmost level, the cluster memory is managed by a Resource Manager (RM) such as Yarn [10]. The RM deploys a Node Manager (NM) on each worker machine which allocates the physical memory to a set of *containers* as per application requirements (e.g., data locality) and multi-tenancy requirements (e.g., sharing resources fairly among tenants). Each container gets to use the resources assigned to it in isolation.

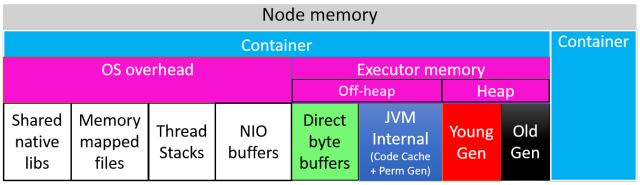

Figure 2: Logical organization of memory pools on a worker node. Executor memory is managed by JVM.

Container level. We focus on JVM-based application frameworks that run a JVM process, called *executor*, within a container. The process is launched with a heap configured with an upper bound on physical memory. Additionally, the application can use off-heap memory to allocate direct byte buffers [1] which is shown as part of the Executor Memory in Figure 2. On top of this, each executor process requires some amount of memory for various OS overheads. Here, the OS stores shared native libraries, thread stacks, Non-blocking I/O (NIO) buffers, and memory-mapped files.

While launching a container, we can configure the amount of memory for the application heap which is to be specified as part of JVM launch command. The off-heap space used for OS overheads, however, could be unbounded. Resource managers employ a utility to monitor this space and warn or kill the application if the value exceeds a set threshold [10].

Application level. An application platform needs to use memory for various purposes including: (a) data cache, (b) execution objects, (c) serialization/deserialization buffers, and (d) user code data structures. There are various choices for the application platform when it comes to managing these objects: Each object could be allocated from a single unified pool of application memory; Or the application memory could be broken into multiple pools for different types of objects. Sizing these pools is an important optimization choice in data analytics platforms [2]. Another important choice an application platform needs to make is regarding the format of objects in memory. While the objects on heap take more space due to Java's serialization overheads, JVM takes on the burden of managing the heap entirely. For an improved performance, application platforms use off-heap space for certain object types [4].

Since the modern analytics platforms are multi-tenant in nature, possibilities of work sharing and resource sharing are available [7]. Managing memory for such shared optimizations brings challenges of both efficiency and fairness.

JVM Process level. One of the salient features of the JVM is its memory management. It periodically runs a process of garbage collection (GC) which collects any unreferenced objects from application heap. The heap is organized into multiple pools of objects as shown in Figure 2. Any object serialized to heap resides in a pool determined by its *age* in heap. The number of pools and the size of each is determined by the GC policy configured at the time of process launch. On a high level, there are two categories of pools: *Young generation* and *Old generation*. As the names suggest, the

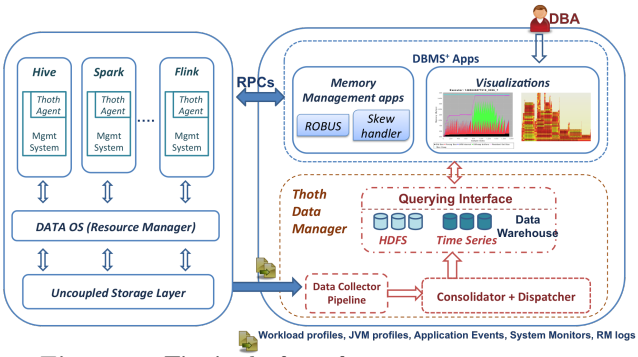

Figure 3: Thoth platform for memory management

young generation pool stores newly loaded objects and the old generation pool stores long living objects.

Interplays. Management of memory pools across levels can be inter-related. As a case in point, data cached in memory by an application is typically needed for a longer duration and is, therefore, expected to end up in old generation pool of heap. If the old generation pool is configured to a size lower than the application's cache data pool, some of the cache objects will end up in young generation pool increasing the frequency of garbage collection calls thereby adversely affecting performance. It is, therefore, necessary to tune application cache pool and GC pools together.

2.2 Thoth Platform

Thoth [8] provides a data-driven platform to build various cluster management *apps*. We make use of this platform in analyzing cluster memory. Figure 3 shows a Thoth prototype specifically catered to memory management. Data generated during an application life-cycle is collected by Thoth agents running in each node. This data is collected and consolidated on a central data warehouse. The key sources of profiled data we use as part of this demonstration are as follows:

- **System profiling using pidstat**: 'pidstat' is a linux utility that instruments a process for resource metrics such as CPU usage and Resident Set Size (RSS) memory.
- **JVM profiling using statsd**: We employ JVM profiling using statsd monitor on every executor JVM.
- **Application event profiles**: In order to understand application platform's management of memory, we track all memory-related events in the underlying application platform. Our prototype builds on the event logging infrastructure provided by Spark [4].
- **Error logs**: Error logs generated by the resource manager and application platforms are monitored for any Out-of-Memory errors.

As Thoth's data manager consolidates data from various sources together, it enables a deep analysis of impact of various memory management options. The platform also enables developing applications that optimize multi-tenant workloads together, ROBUS [7] being a case in point. ROBUS is an online *app* that manages a centralized *cache* to simultaneously optimize a workload from multiple tenants. The choice of data brought to cache guarantees a fair utility to each tenant which is achieved by developing accurate utility estimation models based on workload profiling in Thoth.

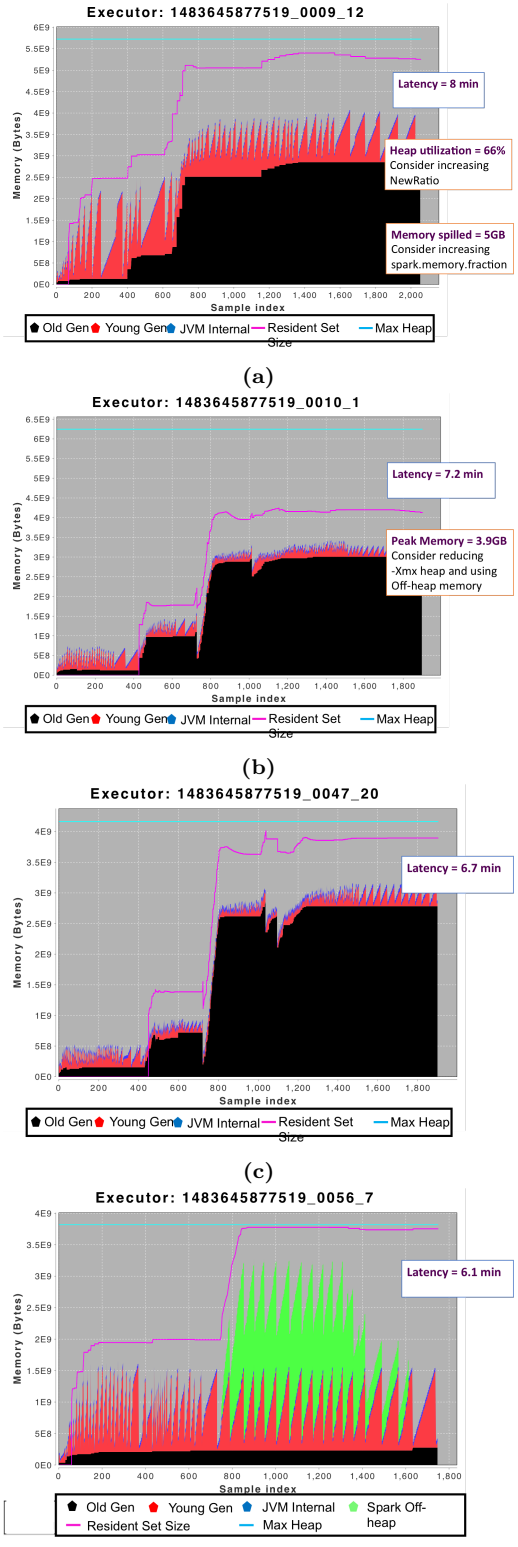

Figure 4: Memory usage for a representative executor running a SortByKey application under: (a) 6GB heap with default GC configurations, (b) 6GB heap with more aggressive GC, (c) 4GB heap with aggressive GC, and (d) 4GB heap with default GC configurations and up to 2GB off-heap buffer space

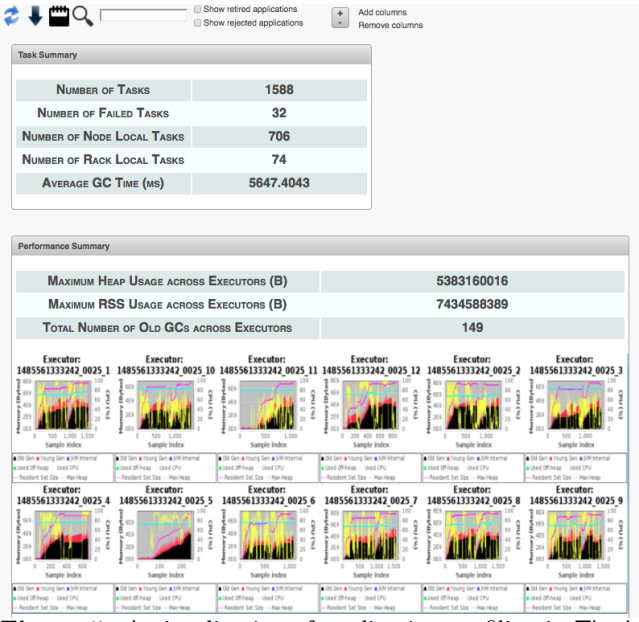

Figure 5: A visualization of application profiling in Thoth

3. DEMONSTRATION PLAN

We have developed a prototype of Thoth platform illustrated in Figure 3 for a 'Spark on Yarn' cluster. A web interface is built to analyze applications using profiling data collected from multiple sources as described in Section 2.2. Different summary views are provided along with visualizations allowing for a visual inspection of memory and performance-related inefficiencies. An example screenshot is presented in Figure 5.

Further, we have built a recommendation engine that provides guidelines on optimizing performance metrics such as latency, memory utilization, predictability of performance, and reliability. The engine works in two phases:

The first phase inspects data local to the application under investigation to find any sub-optimal configurations resulting in performance inefficiencies. For example, a low heap utilization across all executors of an application can trigger an alert to lower the maximum heap configuration; or, a big variance in memory usage of executors can be correlated to the skew in distribution of input data.

The second phase of the recommendation engine looks at profiles of previously run applications to learn better memory configuration settings. A profile matcher component first matches the applications based on a signature built on the input properties and the workflow graphs. All the possible matches are compared with the current application to find out any correlations between the configuration options and the performance metrics. This analysis can help tune applications running as part of scheduled workflows.

An example analysis is presented in Figure 4. It demonstrates how various tuning options at each of Yarn level, Spark level, and JVM level are progressively explored by our tool (namely: allocating right heap size, using off-heap buffers, and increasing frequency of GC). The tuning results in a configuration that not only reduces the memory provisioned by 33% but also improves the latency by 25%.

Through the case studies presented so far, it is clear that many factors ranging from interplays between different memory management options to the trade-offs in various performance metrics play a part in memory tuning. Thoth platform enables a deep understanding of these factors. The demonstration is intended to present the power of Thoth by means of various memory analysis apps. Some of the key analysis apps that will be demonstrated are the following:

- **Memory tuner**: Using the recommendation engine discussed above, we demonstrate how workflows that run periodically can be optimized for their resource usage and performance predictability.
- **Performance engineering**: Thoth supports visualizations showing memory usage of various memory pools during execution which can help pinpoint the resource-hungry stages in an user application. The data backing the visualizations consolidates important metrics from application event logs, such as average time spent in GC by different application phases, providing great value to application developers.
- **ROBUS**: ROBUS is representative of both a tool managing memory across multiple tenants as well as an app that is online in nature. Tenant workloads are optimized through caching data in a centralized pool of memory which simultaneously improves performance of tenants as well as guarantees a fair performance speedup to each tenant [7].

4. REFERENCES

[1] Byte buffers and non-heap memory. `http://goo.gl/9v5EyT`[Online; accessed 28-Feb-2017].

[2] Consolidate storage and execution memory management. `https://goo.gl/k93rHi`[Online; accessed 28-Feb-2017].

[3] Ganglia Monitoring System. `http://ganglia.sourceforge.net`[Online; accessed 28-Feb-2017].

[4] Spark-2.0.1 Docs. `http://spark.apache.org/docs/2.0.1/`[Online; accessed 28-Feb-2017].

[5] VisualVM. `https://visualvm.java.net/`[Online; accessed 28-Feb-2017].

[6] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 394–409, New York, NY, USA, 2015. ACM.

[7] M. Kunjir, B. Fain, K. Munagala, and S. Babu. ROBUS: Fair cache allocation for data-parallel workloads. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 219–234, New York, NY, USA, 2017. ACM.

[8] M. Kunjir, P. Kalmegh, and S. Babu. Thoth: Towards managing a multi-system cluster. *Proc. VLDB Endow.*, 7(13):1689–1692, Aug. 2014.

[9] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, Oakland, CA, May 2015. USENIX Association.

[10] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.