MemTune: Dynamic Memory Management for In-memory Data Analytic Platforms

[†]Luna Xu, [‡]Min Li, [‡]Li Zhang, [†]Ali R. Butt, [‡]Yandong Wang, [⋆]Zane Zhenhua Hu
[†]*Virginia Tech*, [‡]*IBM T.J. Watson Research*, [⋆]*IBM Platform Computing, Toronto, CA*
{xuluna, butta}@cs.vt.edu, {minli, zhangli, yandong}@us.ibm.com, {zane}@ca.ibm.com

***Abstract*—Memory is a crucial resource for big data processing frameworks such as Spark and M3R, where the memory is used both for computation and for caching intermediate storage data. Consequently, optimizing memory is the key to extracting high performance. The extant approach is to statically split the memory for computation and caching based on workload profiling. This approach is unable to capture the varying workload characteristics and dynamic memory demands. Another factor that affects caching efficiency is the choice of data placement and eviction policy. The extant LRU policy is oblivious of task scheduling information from the analytic frameworks, and thus can lead to lost optimization opportunities.**

In this paper, we address the above issues by designing MemTune, a dynamic memory manager for in-memory data analytics. MemTune dynamically tunes computation/caching memory partitions at runtime based on workload memory demand and in-memory data cache needs. Moreover, if needed, the scheduling information from the analytic framework is leveraged to evict data that will not be needed in the near future. Finally, MemTune also supports task-level data prefetching with a configurable window size to more effectively overlap computation with I/O. Our experiments show that MemTune improves memory utilization, yields an overall performance gain of up to 46%, and achieves cache hit ratio of up to 41% compared to standard Spark.

I. Introduction

Emerging in-memory distributed processing frameworks such as Spark [35] and M3R [28] are experiencing rapid growth and adoption due to their use in big data analytics. A crucial reason for the success of these frameworks is their ability to persist intermediate data in memory between computation tasks, which eliminates significant amount of disk I/Os and reduces data processing times. Consequently, for iterative jobs, in-memory processing has been shown to outperform the well-established Hadoop [1] model by more than ten times [35]. Such performance boost has led to the establishment of in-memory processing as the enabling technology for different data analytic platforms. For instance, a comprehensive ecosystem with a rich set of features has been developed atop Spark, including SQL query [12], [33], machine learning [24], graph computing [32] and streaming [36].

The key resource enabling the performance acceleration of the above platforms is *memory*. Failing to persist the whole working set in memory causes disk I/O or re-computation that leads to performance degradation. However, not all memory can be used for caching; applications also require memory for processing and data shuffling. Thus, there are opposing demands of caching and processing on the memory, which are growing with larger data sets and complex analysis tasks. Reconciling these demands is non-trivial. The current approach adopted in Spark is to statically configure memory partitions based on user specifications. However, this entails that users have deep knowledge about their workloads including process working set size, input data size, and data dependency. Given that the frameworks are general purpose, such a "best configuration" differs significantly across workloads. Moreover, determining a best configuration is hard and cumbersome and often not even possible as a users may be simply employing a prepackaged analytics application and not intimately aware of its system-level characteristics. This is problematic, as we show in our evaluation that there is a large penalty for using mismatched configurations. Furthermore, we have observed that even in a single workload, the memory usage changes during execution. This is due to the change of data dependency and task working set size. Thus, a static configuration approach cannot capture such varying workload behavior, consequently leading to degraded performance.

Dynamic memory tuning at runtime can help improve memory resource utilization and reduce memory contention between data cache and process and shuffle memory. While promising, this is a challenging task due to two reasons. First, dynamic tuning requires accurate accounting information about both shuffle and tasks memory consumption, which is not available, and memory tuning can be counterproductive when such information is lacking. Second, it is difficult to decide which data to keep in memory and which data to evict, especially when multiple datasets, e.g., Resilient Distributed Dataset (RDD) [35], needed by the same processing stage cannot be fit into memory. Similarly, if consecutive stages use different RDDs, caching of data from a previous stage may be useless and unnecessarily increase the pressure on memory.

In this paper, we address the above problems and propose MemTune, an approach that uses dynamic Directed Acyclic Graph (DAG) [30]-aware memory tuning for DAG-based in-memory distributed processing platforms. The goal of MemTune is to improve overall memory utilization and reduce performance-degrading memory contention between data cache, process and shuffle memory. MemTune monitors the task memory consumption using statistics such as garbage collection duration and memory paging frequency, and uses the information to dynamically change the data cache size. We also exploit the DAG execution graph of tasks to prefetch

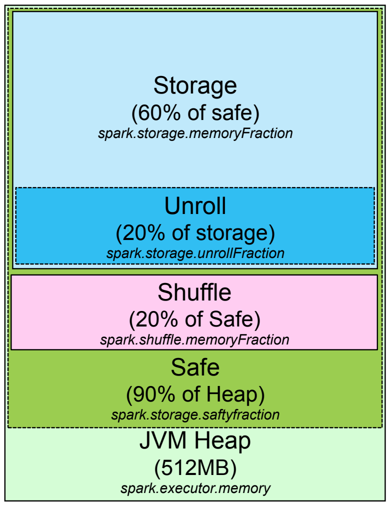

Fig. 1. Typical memory partitioning used in Spark.

data that would be needed by the next stages, thus overlapping the computation and I/O to improve performance.

Specifically, this paper makes the following contributions.

1) We empirically study and demonstrate the impact of memory contention, caching policy and data cache size on performance for workloads running on in-memory data analytic platforms.
2) We design and implement MEMTUNE, a dynamic memory manager atop Spark, which adjusts the cache size and cached data at runtime to capture varying workloads demands and enhance performance.
3) We design and implement an automated algorithm that uses monitored memory statistics and DAG execution flow to determine efficient data cache size and caching policy.
4) We evaluate MEMTUNE in Spark using representative and diverse workloads from SparkBench [21]. Our results demonstrate that compared to static configuration, MEMTUNE reduces workload execution time by up to 46%. Moreover, MEMTUNE effectively detects the desired memory demand of tasks in each stage and change the data cache size accordingly, thus improving memory hit ratio by up to 41%.

II. BACKGROUND AND MOTIVATION

In this section, we first discuss memory management used in Spark. Next, we motivate the need for our proposed solution through an empirical evaluation of Spark on a local cluster, SystemG [7].

A. Spark Memory Management

In Spark, data is managed as an easy-to-use memory abstraction called resilient distributed datasets (RDDs) [35]. An RDD is a collection of objects partitioned across a set of machines. Each machine retains several partitions (blocks) of an RDD in memory. An RDD block can be replicated across nodes for resiliency, and blocks can also be recomputed based on the associated dependencies if the data is lost due to machine failure. Computation is done in the form of RDD actions and transformations, which can be used to capture the lineage of a dataset as a DAG of RDDs, and help in RDD (re)creation as needed. Such DAGs of RDDs are maintained in a specialized component, DAGScheduler, which also schedules the tasks as needed.

Spark is deployed using a *driver program* running on a master node of a resource cluster and several *executors* running on worker nodes. Executors are launched as JAVA processes within which all tasks are executed. Each executor allocates its own heap memory space for caching RDDs. Figure 1 shows the memory partitions used by an executor. By default, Spark allocates a maximum of 90% of the heap memory size as *safe space* for RDD cache and shuffle sort operations, and reserves the remaining 10% of the heap for tasks processing. The figure shows how the safe space is further partitioned for RDD storage, shuffle sort operations, and RDD serialization/deserialization. If the assigned RDD cache is full, any remaining RDD blocks will either be re-computed or spilled to disk based on user specification, i.e., under Spark options `MEMORY_ONLY` and `MEMORY_AND_DISK`, respectively.

B. Empirical Study

The memory management adopted in Spark is static. However, as discussed earlier, workload variance implies that such fixed memory partitioning may not offer the best performance. To quantify the impact of memory management on workload performance, we conduct an empirical study using SparkBench [21], a comprehensive benchmark suite for Spark. We use Spark version 1.5, the latest release of Spark, with Hadoop version 2.6 providing the storage layer. All experiments are done on 6 nodes of our SystemG cluster. Each node has two 4-core 2.8 GHz Intel Xeon processors and 8 GB memory, and boosts a 1 $Gbps$ Ethernet interconnect. One of the nodes is configured to be the master node and the others as workers for both HDFS and Spark. We configure each worker node to have one executor with 6 GB memory, leaving the remaining memory for OS buffer and HDFS data node operations. Each executor has 8 task slots, one for each CPU core. We repeated each experiment 5 times, and in the following report the average results.

1) Memory Contention: In our first test, we study the performance under varying configurations such as persistence level and `spark.storage.memoryFraction` values. We study two persistence levels, the default `MEMORY_ONLY` and `MEMORY_AND_DISK`. For this test, we use the *Logistic Regression* workload. We ran the workload with an input data size of 20 GB, with 40 GB total system memory capacity. We change the configuration parameter from 0 to 1, i.e., from no memory to cache RDDs to all of the memory used for caching. We set the workload iteration limit to three to ensure that the experiment can finish in a reasonable amount of time without sacrificing the accuracy of the inferences drawn.

Figure 2 shows the overall execution time that includes compute time and garbage collection (GC) time of the work-

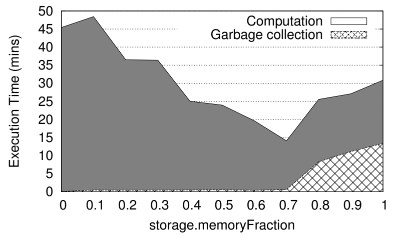

Fig. 2. Total execution time and garbage collection time of *Logistic Regression* under `MEMORY_ONLY`.

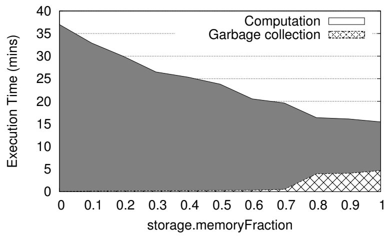

Fig. 3. Computation time and garbage collection time of *Logistic Regression* under `MEMORY_AND_DISK`.

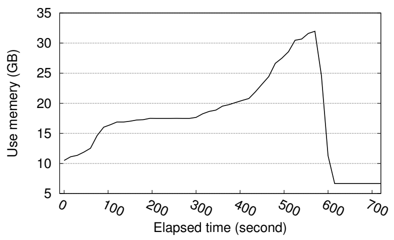

Fig. 4. Memory usage of *TeraSort*.

load under the default level of `MEMORY_ONLY` with increasing `spark.storage.memoryFraction`. We record the GC time in each executor during the workload execution and report the average here. We can see that the overall performance is the best when the parameter is configured to a value of 0.7. Lower values result in larger compute time. This is because there is not enough memory for caching RDDs, forcing needed RDDs to be recomputed. We also found that when the parameter value is configured to a higher value (from 0.8 to 1), the overall execution time is once again increased. Upon closer inspection, we find that the reason for this increase is that the GC time is now increased due to less memory being available to the JVM, which causes more frequent garbage collection. Thus, simply allocating more memory to RDD caching is not beneficial.

Note that the compute time includes RDD computation, task computation and framework overhead. Increasing the memory fraction for RDD increases contention with the executor's other tasks. This is because Spark utilizes part of the executor memory for RDD storage, meanwhile tasks are also launched inside the same executor. If too much memory is allocated for RDD caching, the executor will again incur a huge GC overhead, resulting in degraded performance.

Figure 3 shows that a similar behavior is observed under `MEMORY_AND_DISK` as well. With this persistence level, the GC overhead is not as pronounced as the default memory-only level. This is because spilling RDDs to disk avoids recomputation, which in turn decreases memory contention.

From this test, we see that to get the best performance, we need to balance the memory fraction between RDD cache and compute memory consumption. For the above case, the best fraction turned out to be 0.7, but this may not hold for other real workloads that exhibit different characteristics. For example, memory-intensive workloads such as *Logistic Regression* require more memory for tasks to execute, while I/O-intensive workloads such as *TeraSort* require less memory but have frequent accesses to data. Static configuration relies heavily on users' knowledge about the workload characteristics as well as the deployment setup to find the best configuration. To remedy this, the Spark community recommends using a value of 0.6 for `spark.storage.memoryFraction`, which works for general workloads with modest input data sizes and system set up. However, this approach fails to handle big input data sizes (`OutOfMemory` error was thrown during our test runs

Workload	Input size (GB)
Logistic Regression	20
Linear Regression	35
Page Rank	2
Connected Components	< 1 (16M nodes, 99M edges)
Shortest Path	6

TABLE I: Maximum input size under Spark with default configurations.

with big data sizes). The value is also not suitable for long running jobs with varying characteristics. To investigate this further, we ran several experiments with different workloads and input sizes and show the results in Table I. The table points out the maximum input data size that Spark was able to handle using the community-suggested parameter values without `OutOfMemory` errors. Note that for some workloads the problem started with as small an input as $1\ GB$, which is a worrisome observation for a big data processing framework.

2) Static Configuration: Once set, static configuration is effective throughout an application execution, which makes it hard to adapt to workloads with dynamic memory demands. For example, Figure 4 shows the memory use of *TeraSort*. We set the RDD cache size to 0 in order to observe the task memory consumption. We observe a burst in the memory usage after about 8 minutes. Under static configuration, a user would have to configure the RDD cache size to a small number throughout the execution to accommodate such a burst in task memory requirement, thus losing the opportunity to utilize the memory for RDD cache in earlier stages. An ideal dynamic approach, on the other hand, can start with more memory for RDD, reap the benefits of caching, and then reduce the RDD memory to accommodate the burst and so on. MEMTUNE aims to realize such dynamic management.

3) Memory Management Policy: Spark uses LRU [26] policy for evicting RDD blocks from memory. If a block is evicted, it is not brought back to memory again, and accessed directly from disk or re-computed in cases it is accessed again. The policy is effective, but does not consider an RDD's dependency and need for future stages, information that is available in the workflow DAG. For example, in the *Shortest Path* workload, there are 7 stages and 5 RDDs (RDD3, RDD16, RDD12, RDD14, and RDD22) need to be cached. Among the 7 stages, 5 stages have RDD dependencies. Table II shows the RDD dependencies and the total sizes of the RDDs

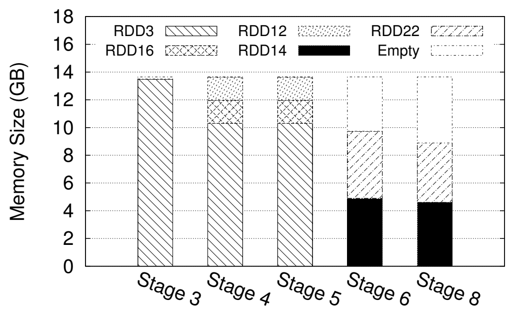

Fig. 5. RDD sizes in memory in different stages of *Shortest Path* under default configurations.

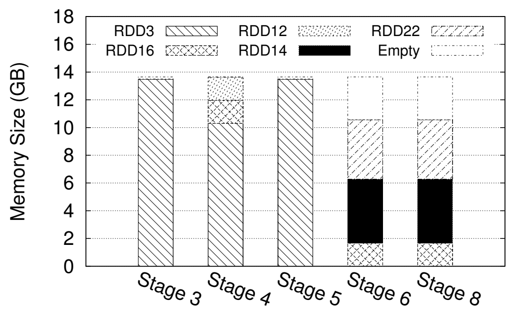

Fig. 6. Ideal RDD sizes in memory based on stage RDD dependencies of *Shortest Path*.

Stages	RDD3 (18.7G)	RDD16 (4.8G)	RDD12 (4.8G)	RDD14 (11.7G)	RDD22 (12.7G)
Stage 3	×	·	·	·	·
Stage 4	·	×	×	·	·
Stage 5	×	·	·	·	·
Stage 6	·	×	·	·	·
Stage 8	·	×	·	·	·

TABLE II: RDD dependencies for different stages of *Shortest Path*. '×' and '·' denote whether a stage is dependent on an RDD or not, respectively.

for this workload. Figure 5 shows the RDD sizes in the beginning of the 5 stages that have RDD dependencies. In contrast, Figure 6 shows the ideal RDD sizes that the stage needs. Note that with default configuration in our test cluster, we have 14 GB in total for RDD storage.

We can see that for stage 3 and stage 4, LRU works well. However, stage 5 is solely dependent on RDD3, but some RDD3 blocks are evicted in stage 4. Also, stage 6 and stage 8 are dependent on RDD16, while no RDD16 is cached in memory because it is completely evicted from memory after stage 5. Since the evicted RDDs will not be brought back to memory again, there is extra empty room left in both stages. This RDD placement policy leads to an inefficient use of memory resource. MEMTUNE aims to address this by designing better workload-aware memory management.

III. SYSTEM DESIGN

In this section, we first describe the architecture of MEMTUNE, followed by how we achieve dynamic memory tuning and RDD cache management.

A. Architecture Overview

Figure 7 shows the overall architecture of MEMTUNE. Although we have implemented MEMTUNE atop Spark, MEMTUNE can also work in multi-tenancy environments with other cluster resource managers such as YARN [29] and Mesos [18]. This is because MEMTUNE manages resources that are provided to it, and thus can naturally extend to containers supported by YARN or Mesos like systems.

MEMTUNE has two key centralized components, *controller* and *cache manager*, and a distributed component, *monitor* that is implemented within each executor on participating nodes. The controller implements the main logic flow of MEMTUNE, such as determining and setting the RDD cache size for each stage, the RDD eviction policy, and the prefetch window size. The cache manager implements the APIs described in Table III. The distributed monitors are responsible for gathering runtime statistics such as garbage collection time, memory swap, task execution time per stage, and input and output dataset sizes. The monitor is designed to be an extensible component so that additional information can be easily captured as needed.

After an application is submitted through spark-submit scripts, Spark launches a Spark driver program together with a *SparkContext* object. Within SparkContext, MEMTUNE's controller and cache manager are instantiated along with the *DAGscheduler* and *BlockManagerMaster*. Next, Spark launches its executor components on the participating nodes, which results in the MEMTUNE monitors being deployed on the cluster as well. The controller periodically gathers data from each monitor and uses the information to adjust RDD cache sizes on nodes, and if needed, selects blocks to evict or prefetch. The controller then communicates this information to the cache manager, which in turn invokes the BlockManager-Master requests to perform the needed operations and execute the commands on the working nodes.

To support dynamic memory configuration, we modify BlockManagerMaster to allow dynamically changing of RDD cache sizes and triggering RDD eviction if the cache is now smaller than the cached data. MEMTUNE supports a set of APIs for this purpose, as shown in Table III. Typically, MEMTUNE will use these APIs to manage RDD cache automatically. However, the APIs also allow users to explicitly control RDD cache ratios, RDD eviction policy and prefetch window during application execution. MEMTUNE automatically manages the RDD cache by efficient eviction and prefetching with a dynamically-adjusted prefetch window. By considering application I/O demands in controlling the prefetch window size—which determines how much data to be prefetched from disks to memory—MEMTUNE effectively overlaps computation with disk I/Os and avoids I/O contention.

B. Dynamically Tuning RDD Cache and JVM Heap Size

As shown in Figure 1, the JVM memory is used by task execution, shuffle sort operations, and RDD cache. It is crucial

API	Description
`double getRDDCache(AppID aid)`	Returns the current RDD cache ratio for the application with ID `aid`.
`void setRDDCache(AppID aid, double rddCacheRatio)`	Sets the RDD cache ratio to value `rddCacheRatio` for the application with ID `aid`.
`void setPrefetchWindow(AppID aid, double prefetchWindow)`	Sets the prefetch window with a value `prefetchWindow` for the application with ID `aid`.
`void setEvictionPolicy(AppID aid, EvictionPolicy ep)`	Sets the RDD eviction policy for the application with ID `aid`.

TABLE III: Key APIs provided by MEMTUNE.

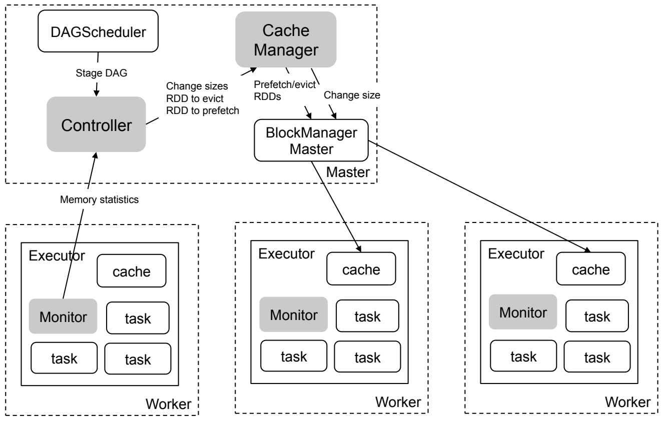

Fig. 7. System architecture of MEMTUNE.

to coordinate the different needs, especially, when there is a contention. Furthermore, node memory outside of JVM provides buffer space for shuffle reads and writes. If there is not enough space to buffer the shuffle data, significant disk I/O would occur, degrading performance. Thus, if the workload is not memory-intensive rather shuffle-intensive, we can enlarge such buffer space by shrinking the JVM heap size. The memory contention between such different needs is shown in Table IV. We observe that there is no contention between shuffle and task execution. This is because shuffle operations usually happen in the end (writes) or start (reads) of a stage. Spark determines a new stage based on whether there is a shuffle operation or not, and the new stage does not have RDD cache dependencies, rather the RDDs are obtained through shuffle reads. Although during shuffle writes, there are RDD cache dependencies, data can only be written when tasks are finished with their computation.

Moreover, a challenge here is that we need to ensure that applications can finish without errors, because task failures due to memory errors are not recoverable. Consider how such errors are handled in Spark. Upon getting a memory error, a user has to either reduce the RDD cache ratio or increase the task's parallelism, and then re-launch the entire application. However, this is a trial-and-error approach, and the process may have to be repeated until the out-of-memory errors disappear. This is cumbersome and frustrating especially if the application is a long running one.

We leverage the above observations in MEMTUNE to prioritize and first allocate sufficient task memory, then shuffle sort memory, shuffle buffer, and finally RDD cache. It is challenging to determine both the task and shuffle memory demands. Currently MEMTUNE adopts indicators of GC ratio and swap ratio, as well as three different thresholds, Th_GCup, Th_GCdown, and Th_sh to determine whether there is enough memory for task execution and shuffle operations. The indicators can be extended to other indicators with more accuracy such as task memory footprint in the future. Currently, the thresholds are set based on observations from our experimentation, but they can also be exposed to users in the form of parameters. For the RDD cache, we start with the maximum fraction of 1 instead of the default of 0.6, and adjust it dynamically as needed to accommodate other demands as follows. If the monitors detect a shuffle operation, it implies that there is currently no need to access the RDD cached data. In this case, we prioritize shuffle operations over RDD cache and reduce the cache size. Conversely, if the tasks are not in the shuffle phase, we allocate more memory for the RDD cache while also ensuring (using the GC etc. indicators described above) that the tasks have enough memory for execution.

Case#	Shuffle	Task	RDD	Contention	Action
0	N	N	N	N	N/A
1	N	N	Y	RDD	↑JVM, ↑cache
2	N	Y	N	Task	↑JVM
3	N	Y	Y	Task, RDD	↑JVM, ↓cache
4	Y	N	N	Shuffle, RDD	↓cache, ↓JVM

TABLE IV: Cases of memory contention and corresponding actions taken by MEMTUNE. Yes (Y) and No (N) under the Shuffle, Task and RDD columns denote whether there is memory contention detected for that usage.

Table IV classifies five cases of contention and the corresponding actions performed by MEMTUNE. There are two tuning knobs to mitigate the contention, namely the JVM size and the RDD cache size. By default, we set the JVM heap size to the maximum available memory of each physical node to maximize utilization. We tune the JVM size asymmetrically, in the sense that we prefer to only reduce the JVM size temporally when we detect shuffle contention. We always first increase JVM size whenever we detect task or RDD memory contention, if JVM heap size has been set to less than the maximum memory allocation in a prior epoch. If the JVM heap is already at its maximum value, we proceed as follows. If there is RDD contention only, we conservatively increase the

Algorithm 1: Controller workflow algorithm.

```
Input: $block\_size$, $RDD\_size$, $shuffle\_size$
begin
    $RDD\_list \leftarrow$ calculate dependent RDD list of the stage;
    if $RDD\_size <$ sizeof $(RDD\_list)$ then
        prefetch($window\_size$);
    while true do
        $\{gc, swap\} \leftarrow$ get GC and page swap information
        from monitor;
        $gc\_ratio \leftarrow$caculate_gc_ratio($gc$);
        $swap\_ratio \leftarrow$calculate_swap_ratio($swap$);
        if $gc\_ratio > Th\_GCup$ then
            $RDD\_size -= block\_size$;
            evict_rdd($block\_size$);

        if $swap\_ratio > Th\_sh$ then
            $\alpha_{sh} \leftarrow block\_size \times number\_of\_tasks$;
            $RDD\_size -= \alpha_{sh}$;
            evict_rdd($\alpha_{sh}$);
            $shuffle\_size += \alpha_{sh}$;
            $jvm\_size -= \alpha_{sh}$
        if $gc\_ratio < Th\_GCdown$ then
            $RDD\_size += block\_size$;
        sleep(5);
```

RDD cache size by one unit. If there are both Task and RDD contention, priority is given to Tasks and the RDD cache size is reduced by one unit. Finally, if there is shuffle contention, both the RDD cache and JVM heap size are reduced by the same amount α_{sh}, i.e., we give a portion of the memory allocated to RDD cache to shuffle (before giving up the memory allocated for tasks). Note that if there is no contention, MEMTUNE does not perform any actions in the current epoch.

The main loop (line 4 to line 20) of the Algorithm 1 shows the steps taken by the controller when we cannot simply increase the JVM heap size to mitigate memory contention. Using the periodically gathered runtime statistics (GC time and page swap amount) from the monitor, MEMTUNE calculates the GC ratio and swap ratio, and checks if the GC ratios exceeds Th_GCup, i.e., an upper threshold for GC ratio. If this is the case, MEMTUNE determines that there is a memory shortage for tasks and reduces the RDD cache size by one unit size. We choose one RDD block size as the unit size because this is the minimum amount of RDD cache size that we can evict to release memory. Next, if the swap ratio for an executor exceeds the Th_sh, it means that memory is needed by the N_s tasks that are performing shuffle in the executor. To provide the memory, we reduce the RDD cache by N_s units to ensure that none of the shuffle tasks suffer from swapping. We also increase the shuffle sort size and decrease the JVM heap size to give more memory for I/O buffers. Finally, if the GC ratio is too low (less than Th_GCdown) implying that the tasks are not using much memory, we increase the RDD cache size by one unit to give more memory to the RDD cache. We conservatively set Th_GCdown smaller than Th_GCup to give priority to task execution memory. Note that the controller

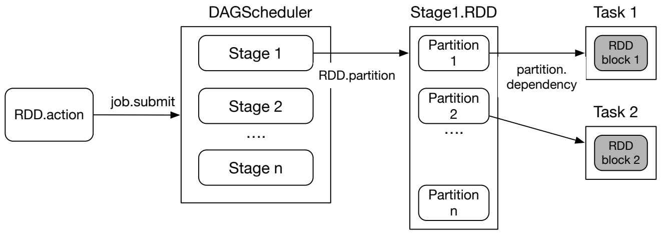

Fig. 8. Generation of tasks based on RDD dependency after an RDD action is submitted to DAGSchduler.

triggers these steps periodically so that the size can be changed gradually. Even if the controller makes a sub-optimal decision in the current epoch, it can be improved/corrected in the next epochs.

C. RDD Eviction Policy

Our automatic RDD cache manager may entail that some RDDs need to be evicted. For this purpose, we do not employ the default LRU policy for RDD cache. Instead, MEMTUNE leverages the DAG information generated by Spark task scheduler for selecting eviction candidates.

As seen in Figure 8, an RDD action triggers *SparkContext* to submit a job, which is then handled by the *DAGScheduler*. The scheduler divides a job into stages based on RDD dependencies, and submits the stages one by one. Each stage has a group of tasks that actually performs the computation. These tasks are generated based on RDD blocks that the tasks need to produce. Here, the controller calculates the RDD block dependency for each block (gray blocks in Figure 8) and associates this information with the tasks for supporting task-level prefetching later. Thus, in each stage of a job, we have a collection of tasks with their dependent RDD blocks. We refer to this group of RDD blocks as hot_list. Note that all RDD eviction and prefetching are within fine-grained block level. This enables the controller component to have the DAG information for each stage. Moreover, the controller can commence prefetching with a hot_list before the associated tasks are submitted. During task execution, finished tasks are also tracked to help eviction as needed by adding the RDD blocks of such tasks to a $finished\_list$.

Two scenarios can occur that can trigger evicting RDD blocks. The first is when the controller reduces the RDD cache size. In this case, the controller first scans the current in-memory cached RDD blocks ($memory\_list$) obtained from the cache manager. If a block is found that does not belong to the hot_list, that block is chosen for eviction. Otherwise, a block from the $finished\_list$ is evicted if it is not empty. If no eviction candidate is available so far, a block in $memory\_list$ with the highest partition (block) number is chosen since it is least likely to be used right away. The choice of evicting high-partition number blocks is based on the observation that Spark schedules tasks according to partition numbers in an ascending order, so we are evicting a block to be used farthest in the future, i.e., effectively an LRU policy. This eviction policy prevents eviction of current blocks that are in use. Finally, the controller uses the cache manager to evict the chosen block and reduce the RDD cache size.

The second eviction case arises when a new RDD block needs to be placed in memory, but the cache is full or does not have sufficient room. The default behavior in Spark is to first check if there are blocks of other RDDs in memory, if so, those are evicted. Otherwise, a warning message is generated to indicate that blocks from the same RDD are being evicted, and then LRU RDD blocks are spilled to disk. This is not desirable as blocks that may be needed soon may be evicted even though the cache contains blocks on the $finished\_list$ that are no longer needed. To remedy this, we changed the default policy to first evict $finished\_list$ blocks before spilling others. The goal of MEMTUNE eviction policy is to utilize RDD dependency information to achieve better caching performance. However, the users can still use the explicit control APIs of MEMTUNE to implement their own custom policies as needed.

D. Prefetch and Prefetch Window

The controller checks to see if all the dependent RDD blocks are cached at the beginning of each stage (Algorithm 1 line 1 to line 3) and triggers prefetching as needed. MEMTUNE goes further to also aggressively prefetch RDD blocks from disk to effectively overlap the task computation and I/O. The intuition is that since we know the task scheduling sequence and the task running status per machine, we have exact knowledge about which RDDs will be accessed in the next epoch and can prefetch them into memory. One exception is that when the tasks are determined to be I/O bound, indicating that there is little additional disk bandwidth for prefetching, in which case prefetching is not done.

To achieve prefetching, the cache manager creates a prefetch thread running on each executor. The thread continuously prefetches data as long as the prefetch window is not filled. MEMTUNE uses an initial prefetch window size that is twice the degree of task parallelism (number of tasks in each executor). This is because tasks are executing in parallel, and data are consumed in a wave (number of tasks). If the controller detects memory contention and decides to drop an RDD block from memory, the window size will be decreased by one wave. If no contention is detected for tasks and shuffle, the size is increased again to the maximum used as the initial value. This policy also ensures that memory priority is given to task execution.

MEMTUNE keeps a list of blocks to prefetch ($prefetch\_list$) and a list of blocks that are prefetched ($cached\_list$). Upon trigger, the prefetching thread scans the RDD blocks that are present in a node disk ($disk\_list$), finds any RDD blocks that are in the hot_list, and puts them on the $prefetch\_list$. The $prefetch\_list$ blocks are then read one by one in ascending order of partition numbers and placed on the $cached\_list$. The prefetching continues until the size of the $cached\_list$ is equal to the prefetch window size, which is a configurable parameter as indicated above. When a task is started, access to any block on the $cached\_list$ results in it being moved to the standard cached block ($memory\_list$). This makes room for further prefetching. The prefetching thread keeps track of the size of the $cached\_list$ and perform prefetching whenever the size is less than the prefetch window.

E. Discussion

In a multi-tenant environment where there are multiple applications running at the same time, we also need to consider other factors such as service level agreements (SLAs), job priority, and overall system utilization when deciding the memory allocation. MEMTUNE does not have a global system view, however, the underlying resource managers can instruct MEMTUNE by setting a hard limit of JVM size so that MEMTUNE will not expand its memory for an application beyond what is allowed. While inside this hard limit, MEMTUNE strives to best utilize the memory resource. This would ensure that MEMTUNE improves individual allocated memory utilization of each application.

IV. EVALUATION

In this section, we demonstrate the efficacy of MEMTUNE on our SystemG setup (described in Section II). We have implemented MEMTUNE in Spark by modifying about 20 classes. Mainly, we modified the Spark *DAGScheduler*, *BlockManagerMaster*, *BlockManager* classes to realize the controller, cache manager, and prefetcher components, respectively. We use the built-in function `dropFromMemory` to evict RDD blocks, and implemented a new helper function `loadFromDisk` to load RDD blocks from disk to memory.

A. Overall Performance of MEMTUNE

In our first test, we study the overall performance impact of MEMTUNE. For this purpose, we use five workloads from the SparkBench [21] suite as shown in Table I, with the maximum input sizes that can be run on Spark without errors. Among these five workloads, *Logistic Regression* and *Linear Regression* both have RDDs whose size is larger than the aggregated cluster RDD capacity. On the other hand, the graph computation workloads (*Page Rank*, *Shortest Path* and *Connected Components*) have smaller RDDs that can completely fit into the RDD cache under the default Spark configuration. However the graph workloads cannot complete successfully if we increase the input data size. Figure 9 shows the overall workload execution time under four scenarios: Spark with default configuration, MEMTUNE with dynamic memory tuning only, MEMTUNE with prefetch only, and MEMTUNE with both dynamic memory tuning and prefetching enabled. We can see that MEMTUNE performs comparable or faster than the default Spark (up to 46.5% improvement) for all workloads. Note that this performance improvement is compared against Spark with the default configuration (`storage.memoryFraction=0.6`). MEMTUNE performs better than the optimal configuration (`storage.memoryFraction=0.7`) shown in Figure 2. *Page Rank*, *Connected Components*, and *Shortest Path* do

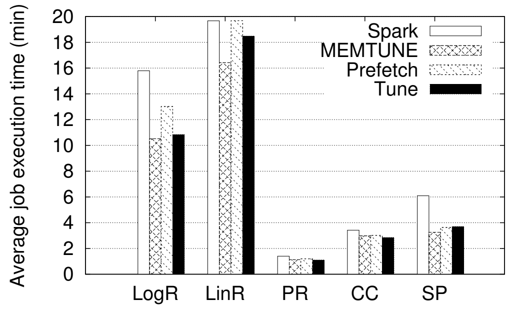

Fig. 9. Execution time of studied workloads under the default Spark, MEMTUNE, MEMTUNE with prefetch only, and MEMTUNE with tuning only. LogR: *Logistic Regression*, LinR: *Linear Regression*, PR: *Page Rank*, CC: *Connected Components*, SP: *Shortest Path*.

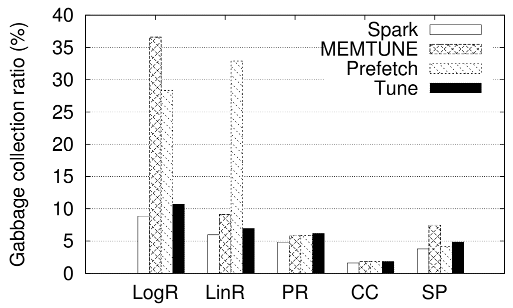

Fig. 10. Gabbage collection ratio of studied workloads under different scenarios.

not benefit much from MEMTUNE because the input data size is not big enough to exhaust the memory and no RDD block is spilled to disk under the default Spark setup. When we increased the data size, the default Spark emitted `OutOfMemory` errors and failed the execution, while MEMTUNE was able to finish execution without errors even with larger data set sizes. Moreover, except for *Shortest Path*, we see that most workloads benefit from dynamic configurations more than data prefetching. This is because most of these workloads are not CPU-intensive, resulting in short task execution times, and thus do no leave enough time to overlap I/O with computation through task-level prefetching. However, prefetching was able to help *Shortest Path* by overlapping task computation with I/O, and reduced the workload execution time by 46.5%. The overall average performance gain across the studied workloads achieved by MEMTUNE is 25.7% compared to the default Spark.

B. Impact of Garbage Collection

Next, we repeat the previous set of experiments and collect the total garbage collection (GC) time on each executor during the entire application execution. We report the average ratio of GC time to overall application execution time. Figure 10 shows that MEMTUNE imposes bigger GC ration than Spark with default configuration. This is because of two reasons: Spark does not exhibit a huge GC overhead under the default configuration of 0.6 as shown in Figure 3. Secondly, dynamic memory tuning of MEMTUNE tends to increase RDD storage size when GC is not significant. Finally, data prefetching provided by MEMTUNE tends to bring RDD blocks to memory, which also increases the memory utilization. For *Linear Regression*, MEMTUNE has lower GC ratio compared to the Prefetch only case since MEMTUNE decreases RDD cache sizes for task computation while RDD blocks are prefetched.

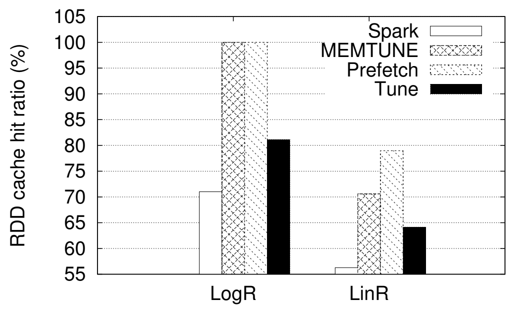

Fig. 11. RDD memory cache hit ratio of studied workloads under different scenarios.

C. Impact on Cache Hit Ratio

In our next test, we use *Logistic Regression* and *Linear Regression* to study the RDD cache hit ratio under MEMTUNE. We do not use the graph computing workloads as they fit in memory and have a 100% hit rate for the four studied scenarios. We can see in Figure 11 that prefetching under MEMTUNE results in the highest RDD memory cache hit ratio—up to 41% improvement compared to the default Spark. In contrast, dynamic tuning yields better hit ratios than the default Spark but not as higher as under the other two MEMTUNE scenarios. This is because dynamic memory tuning increases RDD storage sizes for both workloads, thus increasing the number of RDD blocks that are cached. Prefetching has the best cache hit ratio because it prefetches the RDD blocks into memory before the blocks are accessed. For *Logistic Regression*, MEMTUNE with both features enabled achieves the same cache hit ratio as prefetching. However, for *Linear Regression*, MEMTUNE with both features enabled achieves less than prefetching alone. This is because *Linear Regression* has a higher task memory consumption, thus when prefetching the data, dynamic memory tuning reduces the RDD cache size, thus reducing the amount of RDD blocks that are cached.

Discussion: Considering the above three experiments, we see that *Logistic Regression* shows less task memory consumption and benefits more from prefetching. On the other hand, *Linear Regression* shows more task memory contention, thus prefetching alone shows bigger GC overhead and benefits both from prefetching and dynamic tuning. The other three graph computation workloads have small input data sizes, thus show a modest resource consumption, and both the default Spark and MEMTUNE performs similarly. Yet, MEMTUNE is better as it can also process larger input data set sizes where the default Spark fails. MEMTUNE's effectiveness is heightened whenever there is memory contention. RDD prefetch aims at increasing cache hit ratio of RDD blocks in memory,

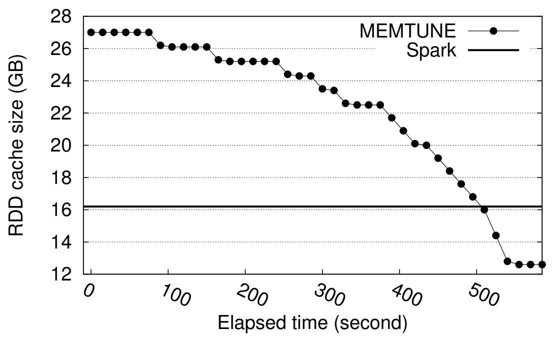

Fig. 12. Dynamic change in RDD cache size during execution.

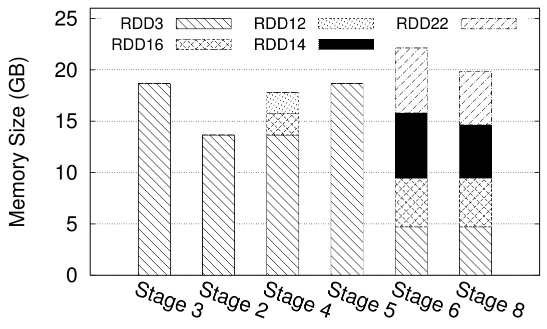

Fig. 13. RDD cache sizes in memory with the default configuration under different stages with MEMTUNE.

while dynamic memory tuning aims at increasing the memory usage while mitigating memory contention among different demands. Combining these two techniques, MEMTUNE increases the memory utilization and the efficiency of RDD cache by overlapping computation with I/O.

D. Dynamic RDD Cache Size Tuning

As discussed in Section II-B2, *TeraSort* exhibits a dynamic memory usage demand that cannot be fully satisfied by a static configuration. Another characteristic of *TeraSort* is that it is shuffle-intensive. Most of its stages involve heavy shuffle I/Os. In our next experiment, we run the *TeraSort* workload with MEMTUNE, and monitor the dynamic RDD size changes over the execution time. Figure 12 illustrates that MEMTUNE starts with a high RDD configuration in the beginning, and decreases gradually throughout the execution. Recall that in Figure 4, we observed a large memory usage burst in the final stage of *TeraSort*. Although the algorithm of MEMTUNE is conservative for catching such bursty memory consumption, by periodic tuning, MEMTUNE is able to keep dropping the RDD cache size until it detects the contention falls below the defined threshold. Increasing the checking and tuning frequency would enable MEMTUNE to react to memory contention more aggressively (though it can add monitoring overhead and may also cause thrashing, which underscores our current conservative approach to tuning).

E. DAG-Aware RDD Prefetching

As shown in Section II-B3, LRU RDD eviction policy works for some stages in a workload, but not for other stages that have RDD dependencies. In our next experiment, we run *Shortest Path* with $4\ GB$ input graph data size under MEMTUNE. We show the RDD memory size in the beginning of the 7 stages in Figure 13. Table II shows the RDD dependencies of the 5 stages that are also applicable here. We see that unlike the default Spark (Figure 5), MEMTUNE brings RDD3 back to memory in stage 5 because MEMTUNE detects that stage 5 is dependent on RDD3. Likewise, MEMTUNE also brings RDD16 to memory in both stage 6 and stage 8. Moreover, on average RDD sizes in memory are also more than the default Spark, and there is no empty space left in the RDD cache. This is because MEMTUNE dynamically changes the RDD cache size and increases it compared to the default 0.6. This is also the reason for the changes in total RDD memory sizes.

V. RELATED WORKS

Memory management is a well studied topic. LRU is widely used in caches [10], [26] and also the default in Spark, however, it does not factor in available dependency information as in MEMTUNE. A number of distributed memory cache systems have also been designed for data intensive parallel processing frameworks [3], [5], [11], [13], [16], [25]. MEMTUNE learns from these projects, but goes further to obtain tasking scheduling orders from the data analytics frameworks, and uses the information to prefetch and evict blocks as needed, and manage memory across the different system components.Systematic approaches such as iTask [15] resolve memory pressure by suspending and resuming running tasks. In contrast, MEMTUNE focuses on memory resizing instead of managing the degree of parallelism and thus avoids the issues of task interference.

Memcached [3] and Redis [5] are highly available distributed key value stores, which are used to cache data in memory for large scale web applications [6], [8]. Megastore [13] offers a distributed storage system with strong consistency guarantees and high availablility for interactive online applications. These systems are complementary to MEMTUNE but are designed for specific application domains and not general purpose framework such as Spark. Grappa [25] provides a distributed shared memory allowing programmers to use the aggregated cluster resources as a large single machine. This is orthogonal to MEMTUNE as it does not use memory cache, and iterative applications still use disk for intermediate results.

Dynamic JVM heap management [14], [34] enables multiple JVMs to concurrently run on the same machine with reduced contention. MEMTUNE also dynamically changes the JVM heap size, but uses different JVM metrics as the contention indicator. Moreover, MEMTUNE's goal is to reduce contention between different application tasks and operation within a JVM to improve overall performance.

Parameter and memory tuning for large-scale data parallel computation have also been explored [9], [17], [19], [22], [23],

[27], [31]. These systems and performance tuning guides propose either automatically change workload configurations or suggest configuration heuristics for MapReduce frameworks, and are orthogonal to MEMTUNE design and aims.

Tachyon [20] offers a reliable in-memory distributed caching layer that caches intermediate data across multiple frameworks. The key idea is using lineage to recompute lost data in case of failures. In contrast, MEMTUNE focuses on dynamically adjusting the memory allocation of task, shuffle and data caching based on application characteristics. Project Tungsten [4] has been proposed to move the memory management from JVM heap to off heap management. However, even when the memory is allocated off heap, the issue of deciding how much memory to allocate to tasks, shuffle and intermediate data remains. MEMTUNE is orthogonal and can be applied on top of project Tungsten.

The Spark open source community [2] has proposed to unify memory allocation within Spark. However, the focus is on how to reallocate RDD cache capicity for shuffle usage but not vice versa, and RDD cache management is left as is. In contrast, MEMTUNE exploits DAG task scheduling information, and offers a comprehensive solution for memory management of in-memory data analytic frameworks.

VI. CONCLUSION

In this paper, we design MEMTUNE, a dynamic memory management approach for in-memory data analytic platforms. MEMTUNE detects memory contention at runtime and dynamically adjusts the memory partitions between in-memory data cache, task execution, and in-memory shuffle sort operations. By leveraging workload DAG information, MEMTUNE proactively evicts and prefetches data with a configurable prefetch window and also employ task-level prefetching. Experiments with an implementation of MEMTUNE in the popular Spark framework shows an overall performance improvement of up to 46% for representative workloads. In our future work, we plan to improve memory usage estimations of MEMTUNE, and expand our system to multi-tenant environments.

VII. ACKNOWLEDGMENT

This work is sponsored in part by the NSF under the grants: CNS-1405697 and CNS-1422788.

REFERENCES

[1] Apache. Hadoop. http://hadoop.apache.org/.

[2] Consolidate storage and execution memory management. https://issues.apache.org/jira/browse/SPARK-10000.

[3] Memcached. http://memcached.org.

[4] Project Tungsten. https://issues.apache.org/jira/browse/SPARK-7075.

[5] Redis. http://redis.io.

[6] Scaling memcached at Facebook. https://www.facebook.com/notes/facebook-engineering/scaling-memcached-at-facebook/39391378919?comment_id=26807469&offset=0&total_comments=159&comment_tracking=%7B%22tn%22%3A%22R9%22%7D.

[7] System G. http://www.cs.vt.edu/facilities/systemg.

[8] Twemcache. https://github.com/twitter/twemcache.

[9] Hadoop performance tuning. https://hadoop-toolkit.googlecode.com/files/Whitepaper-HadoopPerformanceTuning.pdf, 2012.

[10] E. Abrossimov, M. Rozier, and M. Shapiro. Generic virtual memory management for operating system kernels. *ACM SIGOPS*, 23(5), 1989.

[11] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: coordinated memory caching for parallel jobs. In *Proc. of the 9th USENIX NSDI*, 2012.

[12] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proc. of 2015 ACM SIGMOD*, 2015.

[13] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, 2011.

[14] N. Bobroff, P. Westerink, and L. Fong. Active control of memory for java virtual machines and applications. In *Proc. of 11th ICAC 14*, Philadelphia, PA, June 2014.

[15] 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 *Proc. of the 25th SOSP*, pages 394–409. ACM, 2015.

[16] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proc. of OSDI*, 2010.

[17] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *VLDB Endowment*, 2011.

[18] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of NSDI*, 2011.

[19] P. Lama and X. Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proc. ACM ICAC*, 2012.

[20] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. of the ACM SoCC*, 2014.

[21] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proc. of the 12th ACM ICCF*, 2015.

[22] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller. Mronline: Mapreduce online performance tuning. In *Proc. of the 23rd HPDC*, 2014.

[23] G. Liao, K. Datta, and T. L. Willke. Gunther: Search-based auto-tuning of mapreduce. In *Proc. Springer Euro-Par*, 2013.

[24] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *arXiv preprint*, 2015.

[25] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *Proc. of USENIX ATC 15*, July 2015.

[26] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proc. of 93 ACM SIGMOD*.

[27] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang. Mrtuner: a toolkit to enable holistic optimization for mapreduce jobs. *VLDB Endowment*, 2014.

[28] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta. M3r: Increased performance for in-memory hadoop jobs. *Proc. VLDB Endow.*, Aug. 2012.

[29] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proc. 4th SOCC*, 2013.

[30] L. Wang. Directed acyclic graph. In *Encyclopedia of Systems Biology*. Springer, 2013.

[31] T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2012.

[32] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. *arXiv preprint*, 2014.

[33] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proc. 2013 ACM SIGMOD*.

[34] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Cramm: Virtual memory support for garbage-collected applications. In *Proc. of the 7th OSDI*, 2006.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. 2nd USENIX HotCloud*, 2010.

[36] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. of the 24th ACM SOSP*, 2013.