Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling

Călin Iorgulescu and Florin Dinu, ***EPFL;*** **Aunn Raza,** ***NUST Pakistan;***
Wajih Ul Hassan, ***UIUC;*** **Willy Zwaenepoel,** ***EPFL***

https://www.usenix.org/conference/atc17/technical-sessions/presentation/iorgulescu

This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Open access to the Proceedings of the 2017 USENIX Annual Technical Conference is sponsored by USENIX.

Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling

Călin Iorgulescu[*], Florin Dinu[*], Aunn Raza[‡], Wajih Ul Hassan[†], and Willy Zwaenepoel[*]

[*]EPFL [‡]NUST Pakistan [†]UIUC

Abstract

Understanding the performance of data-parallel workloads when resource-constrained has significant practical importance but unfortunately has received only limited attention. This paper identifies, quantifies and demonstrates memory elasticity, an intrinsic property of data-parallel tasks. Memory elasticity allows tasks to run with significantly less memory than they would ideally need while only paying a moderate performance penalty. For example, we find that given as little as 10% of ideal memory, PageRank and NutchIndexing Hadoop reducers become only 1.2x/1.75x and 1.08x slower. We show that memory elasticity is prevalent in the Hadoop, Spark, Tez and Flink frameworks. We also show that memory elasticity is predictable in nature by building simple models for Hadoop and extending them to Tez and Spark.

To demonstrate the potential benefits of leveraging memory elasticity, this paper further explores its application to cluster scheduling. In this setting, we observe that the resource vs. time trade-off enabled by memory elasticity becomes a task queuing time vs. task runtime trade-off. Tasks may complete faster when scheduled with less memory because their waiting time is reduced. We show that a scheduler can turn this task-level trade-off into improved job completion time and cluster-wide memory utilization. We have integrated memory elasticity into Apache YARN. We show gains of up to 60% in average job completion time on a 50-node Hadoop cluster. Extensive simulations show similar improvements over a large number of scenarios.

1 Introduction

The recent proliferation of data-parallel workloads [27, 10, 24] has made efficient resource management [22, 26, 7] a top priority in today's computing clusters. A popular approach is to better estimate workload resource needs to avoid resource wastage due to user-driven over-estimations [26, 12, 21]. Another is over-committing server resources to cope with the variability of workload resource usage [26, 7, 11]. Unfortunately, only a few efforts [12] have touched on the malleability of data-parallel workloads when resource-constrained. The study of malleability is complementary to solutions for over-estimations and variability. While the latter two attempt to accurately track the actual workload resource usage, the former is about allocating to applications fewer server resources than they would ideally need. A thorough understanding of the trade-offs involved in resource malleability is useful in many contexts ranging from improving cluster-wide resource efficiency and provisioning to reservation sizing in public clouds, disaster and failure recovery, and cluster scheduling.

The main contribution of this paper is identifying, quantifying and demonstrating memory elasticity, an intrinsic property of data-parallel workloads. We define memory elasticity as the property of a data-parallel task to execute with only a moderate performance penalty when memory-constrained. Memory elasticity pertains to tasks involved in data shuffling operations. Data shuffling is ubiquitous [6, 20, 28]. It is used by a large number of data-parallel applications across all data-parallel frameworks. Most tasks are involved in shuffling and show memory elasticity: mappers and reducers in MapReduce, joins and by-key transformations (reduce, sort, group) in Spark, and mappers, intermediate and final reducers in Tez.

Despite significant differences in the designs of popular data-parallel frameworks, shuffling operations across these frameworks share a common, tried-and-tested foundation in the use of merge-sort algorithms that may also use secondary storage [3]. The memory allocated to a task involved in shuffling has a part for shuffling and a part for execution. The best task runtime is obtained when the shuffle memory is sized such that all shuffle data fits in it. This allows the shuffle to perform an effi-

†‡ Work done while authors were interns at EPFL.

cient in-memory-only merge-sort. If the shuffle memory is insufficient, an external merge-sort algorithm is used.

The key insight behind memory elasticity is that under-sizing shuffle memory can lead to considerable reductions in task memory allocations at the expense of only moderate increases in task runtime. Two factors contribute to the sizeable memory reductions. First, shuffle memory is usually a very large portion of the task memory allocation (70% by default in Hadoop). Second, external merge-sort algorithms can run with very little memory because they can compensate by using secondary storage. A couple of factors also explain why the task runtime increases only moderately when shuffle memory is under-sized. First, a data-parallel task couples shuffling with CPU-intensive processing thus making far less relevant the performance gap between external and in-memory merge-sort. Second, disk accesses are efficient as the disk is accessed sequentially. Third, the performance of external merge-sort algorithms remains stable despite significant reductions in shuffle memory (a k-way merge is logarithmic in k).

Thus, memory elasticity presents an interesting resource vs. time trade-off. This paper quantifies this trade-off and its implications using extensive experimental studies. We find that memory elasticity is prevalent across the Hadoop, Spark, Tez and Flink frameworks and across several popular workloads. In all cases, the performance penalty of memory elasticity was moderate despite sizeable reductions in task memory allocations. Let M be the task memory allocation that minimizes task runtime by ensuring that all shuffle data fits in shuffle memory. Given as little as 10% of M, PageRank and NutchIndexing Hadoop reducers become only 1.22x/1.75x and 1.08x slower. For Hadoop mappers the largest encountered penalty is only 1.5x. For Spark, Tez and Flink the penalties were similar to Hadoop. Furthermore, we show the predictable nature of memory elasticity which is key to leveraging it in practice. We build simple models for Hadoop that can accurately describe the resource vs. time trade-off. With only small changes, the same models apply to Spark and Tez.

To demonstrate the potential benefits of leveraging memory elasticity, this paper further explores its application to cluster scheduling. Current clusters host concurrently a multitude of jobs each running a multitude of tasks. In this setting, we observe that the resource vs. time trade-off of memory elasticity becomes a task queueing time vs. task runtime trade-off. A task normally has to wait until enough memory becomes available for it but if it is willing to execute using less memory it might have to wait much less or not at all. Since the completion time of a task is the sum of waiting time plus runtime, a significant decrease in waiting time may outweigh an increase in runtime due to elasticity and overall lead to faster task completion times. We show that a scheduler can turn this task-level trade-off into improved job completion time and improved cluster-wide memory utilization by better packing tasks on nodes with respect to memory. Scheduling using memory elasticity is an NP-hard problem because it contains as a special case NP-hard variants of the RCPSP problem [8], a well-known problem in operations research. We propose a simple heuristic and show it can yield important benefits: the tasks in a job can leverage memory elasticity only if that does not lead to a degradation in job completion time.

We have integrated the concepts of memory elasticity into Apache YARN. On a 50-node Hadoop cluster, leveraging memory elasticity results in up to 60% improvement in average job completion time compared to stock YARN. Extensive simulations show similar improvements over a large number of scenarios.

2 Memory elasticity in real workloads

This section presents an extensive study of memory elasticity. We make three key points. First, memory elasticity is generally applicable to several frameworks and workloads. Our measurements put emphasis on Hadoop but also show that elasticity applies to Apache Spark, Tez and Flink. Second, memory elasticity costs little. The performance degradation due to using elasticity was moderate in all experiments. Third, elasticity has a predictable nature and thus can be readily modeled. We provide a model for Hadoop and with only simple changes apply it to Tez and a Spark Terasort job. We also detail the causes and implications of memory elasticity.

We use the term *spilling to disk* to refer to the usage of secondary storage by the external merge-sort algorithms. We call a task *under-sized* if its memory allocation is insufficient to avoid spilling to disk during shuffling. We call a task *well-sized* otherwise. We call *ideal memory* the minimum memory allocation that makes a task well-sized and *ideal runtime* the task runtime when allocated ideal memory. The term *penalty* refers to the performance penalty caused by memory elasticity in under-sized tasks.

2.1 Measurement methodology

For Hadoop we profiled 18 jobs across 8 different applications, most belonging to the popular HiBench big-data benchmarking suite [4]. The jobs range from graph processing (Connected Components, PageRank) to web-indexing (Nutch), machine learning (Bayesian Classification, Item-Based Recommender), database queries (TPC-DS) and simple jobs (WordCount, TeraSort). For Spark we profiled TeraSort and WordCount, for Tez we profiled WordCount and SortMerge Join and for Flink

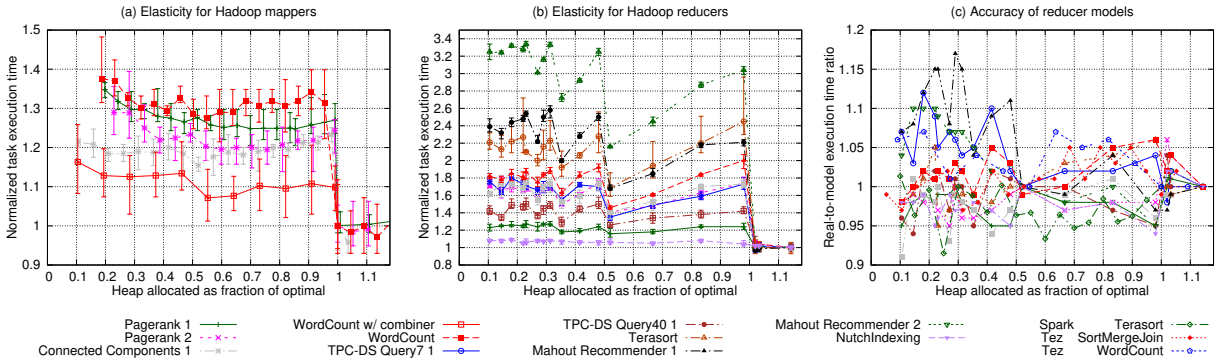


Figure 1: Memory elasticity profiles for Hadoop mappers (a) and reducers (b). Accuracy of our model (c). The reducers (b) exhibit a sawtooth-like pattern determined by the input-to-buffer ratio. Local minimum/maximum penalty values occur right before and after the points where this ratio is an integer (further detailed in §2.3).

we profiled WordCount. We used Hadoop 2.6.3, Spark 2.0.0, Tez 0.7.0 and Flink 1.0.2. However, the same behavior appears in Spark versions prior to 2.0.0 and Hadoop versions at least as old as 2.4.1 (June 2014).

For accurate profiling we made sure that the profiled task is not collocated with any other task. To measure the worst case penalties for under-sized tasks we ensure that disk I/O operations for spills actually go to the drive and not to the OS buffer cache. For this, we ran each task in a separate Linux cgroups container. We minimize the amount of buffer cache available to a task by setting the cgroups limits as close to the JVM heap size as possible. As an alternative, we also modified Hadoop to perform disk spills using direct I/O thus bypassing completely the OS buffer cache. The two approaches gave consistently similar results.

2.2 Memory elasticity for Hadoop mappers

Elasticity for mappers occurs on their output side. The key-value pairs output by map function calls are written to an in-memory buffer. If the mapper is well-sized then the buffer never fills up. In this case, when the mapper finishes processing its input, the buffer contents are written to disk into one sorted and partitioned file (one partition per reducer). If the mapper is under-sized, the buffer fills up while the mapper is still executing map function calls. The buffer contents are spilled to disk and the buffer is reused. For under-sized mappers, at the end there is an extra merge phase that merges together all existing spills. If combiners are defined then they are applied before spills.

The impact of elasticity on mapper runtime Fig. 1a shows the mapping between normalized mapper runtime (y-axis) and allocated heap size (x-axis) for several Hadoop mappers. We call this mapping the *memory elasticity profile*. The penalties are moderate. For example, an under-sized WordCount mapper is about 1.35x slower than when well-sized. If the same mapper uses a combiner, then the penalty is further reduced (1.15x) because less data is written to disk. The maximum encountered penalty across all mappers is 1.5x.

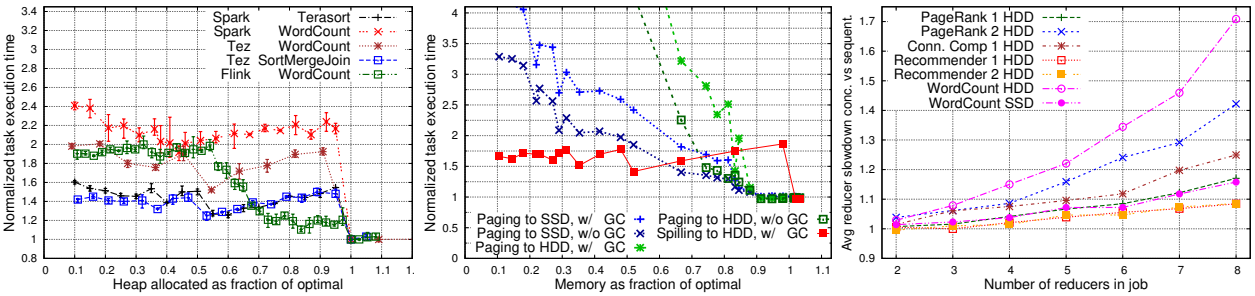
Why penalties are not larger As explained in the introduction, three factors limit the penalties. First, the mapper couples shuffling with CPU-intensive work done by map function calls. Second, disk accesses are efficient as the disk is accessed sequentially. Third, the performance of external merge-sort algorithms remains stable despite significant reductions in shuffle memory.

The shape of the memory elasticity profile The elasticity profile of a mapper resembles a step function. The reason is that under-sized mappers perform an extra merge phase which takes a similar amount of time for many different under-sized allocations.

Modeling memory elasticity for mappers A step function is thus a simple and good approximation for modeling memory elasticity for Hadoop mappers. To build this model two profiling runs are needed, one with an under-sized mapper and one with a well-sized mapper. The runtime of the under-sized mapper can then be used to approximate the mapper runtime for all other under-sized memory allocations.

2.3 Memory elasticity for Hadoop reducers

Elasticity for reducers appears on their input side. Reducers need to read all map outputs before starting the first call to the reduce function. Map outputs are first buffered in memory. For a well-sized reducer this buffer never fills up and it is never spilled. These in-memory map outputs are then merged directly into the reduce functions. For an under-sized reducer the buffer fills up while map outputs are being copied. In this case, the buffer is spilled and reused. The in-memory and on-disk data is then merged and fed to the reduce function of the under-sized reducer.



(a) Elasticity for Spark, Tez and Flink reducers (b) Spilling vs. paging (c) Impact of contention on Hadoop reducers

Figure 2: Memory elasticity for Spark, Tez and Flink jobs (a). Spilling vs. paging (b). Impact of disk contention (c).

The impact of elasticity on reducer runtime Fig. 1b shows the memory elasticity profiles for several Hadoop reducers. The reducer input size ranged from 5GB to 18GB with an average of 10GB. In addition to the jobs in Fig. 1b we also profiled the vector creation part of HiBench's Bayesian Classification. Because this application has many jobs we could not obtain the full elasticity profile for each individual job. Instead, we inferred the maximum penalty for each job using the model described at the end of this subsection. For the 8 distinct jobs we encountered, the maximum penalties are: 1.8x, 1.67x, 2.65x, 1.42x, 3.32x, 1.37x, 1.75x and 1.42x.

Two main insights arise from the results in Fig. 1b. Most importantly, under-sized reducers incur only moderate penalties. Given as little as 10% of ideal memory, 7 of the 10 reducers are between 1.1x and 1.85x slower than ideal. Second, the penalties are comparable for a wide range of under-sized memory allocations. For the WordCount reducer, for 83%, 41% and 10% of ideal memory the penalties are: 1.84x, 1.83x and 1.82x.

Why the penalty varies among reducers We found that the penalty correlates with the complexity of the reduce function. More complex reducers are more CPU-intensive and thus are influenced less by reading inputs from disk. A TeraSort reducer is very simple and shows one of the highest penalties. On the other extreme, the NutchIndexing reducer is complex and shows almost no penalty. To further analyze how reducer complexity influences the penalty we added to the WordCount reduce function a number of floating point operations (FPops) between two randomly generated numbers. Adding 10, 50 and 100 FPops per reduce function call decreased the maximum penalty from 2x to 1.87x, 1.65x and 1.46x.

We also found that the penalty correlates with the number of values corresponding to each key processed by a reducer. A large number of values per key leads to increased penalties because the read-ahead performed by the OS becomes insufficient to bring all keys in memory and thus some reduce calls will read on-disk data. The Mahout recommender uses on average 1500 keys per value for job1 and 15000 keys per value for job2. This explains their larger penalty.

Why penalties are not larger The explanations for reducers are the same as the ones provided for mappers.

Modeling Hadoop reducer behavior An accurate reducer model can be obtained from just two profiling runs: one with under-sized reducers and one with well-sized ones. Given these, the model can infer the penalty for all other under-sized memory allocations. While the profiling runs are application-specific, the model we describe is generally applicable to any Hadoop reducer.

Our model is based on three insights: (a) disk-spilling creates additional penalty on top of the ideal runtime, (b) the penalty is proportional to the amount of data spilled, and (c) the disk rate for reading and writing spills remains constant for a reducer regardless of its memory allocation. We summarize this in the following equation:

$$T(notId) = T + spilledBytes(notId)/diskRate$$

$T(notId)$ is the reducer runtime for an under-sized reducer given $notId$ (not ideal) memory. T is the runtime when that reducer is well-sized. $spilledBytes(notId)$ is the amount of data spilled when being allocated $notId$ memory. Finally, $diskRate$ is the average rate at which the under-sized reducer uses the disk when spilling.

The two profiling runs provide $T(notId)$ and T. Next, $spilledBytes(notId)$ can be computed numerically from the reducer input size, the value of $notId$, and a few Hadoop configuration parameters, thus yielding $diskRate$. Any other $T(notId')$ can be obtained numerically by computing the corresponding $spilledBytes(notId')$ and plugging it in the equation.

Fig. 1c shows the accuracy of our model for the Hadoop reducers profiled in Fig. 1b. The value of $notId$ chosen for the under-sized profiling run was 52% of optimal. Any other under-sized amount would have sufficed. The accuracy of the model is within 5% for most cases.

The shape of the memory elasticity profile We now explain the sawtooth-like shape of the memory elasticity profiles from Fig. 1b. The key insight is that the penalty

is proportional to the amount of spilled data.

The sawtooth peaks are caused by the input-to-buffer ratio of a reducer. When the input is close to a multiple of the buffer, almost all data gets spilled to disk. For example, given a 2.01GB input, two reducers with 500MB and 2GB shuffle buffers, respectively, will each spill 2GB.

There are several cases in which decreasing the memory allocation also decreases the penalty (e.g., WordCount with 52% vs. 83% of ideal memory). This is caused by a decrease in the amount of spilled data. Given a 2GB shuffle buffer and a 2.01GB input size, a reducer spills 2GB to disk but given a 1.5GB shuffle buffer it spills only 1.5GB to disk and keeps 510MB in memory.

One may argue that the static threshold used by Hadoop for spilling is inefficient and that Hadoop should strive to spill as little as possible. In this argument, the reducer with 2.01GB input and a 2GB shuffle buffer would spill 10MB only. Such a proposal actually strengthens the case for memory elasticity as the penalties decrease (due to less spilled data) and can be modeled similarly.

2.4 Elasticity for Spark, Tez and Flink

Fig. 2a shows that memory elasticity also applies to Spark. The reducer input size ranged from 2GB to 22GB with an average of 10GB. For Spark we profiled a task performing a sortByKey operation (TeraSort) and one performing a reduceByKey operation (WordCount). Internally, Spark treats the two cases differently. A buffer is used to store input data for sortByKey and a hashmap for reduceByKey. Both data structures are spilled to disk when a size threshold is reached. Despite the differences both tasks show elasticity.

Fig. 2a shows that the elasticity profile for Spark resembles that of Hadoop reducers. Given the similarities we were able to extend our Hadoop reducer model to Spark sortByKey tasks (TeraSort). The difference between the Hadoop and Spark TeraSort model is that for Spark we also learn an expansion factor from the under-sized profiling run. This is because Spark de-serializes data when adding it to the shuffle buffers. Fig. 1c shows that the accuracy of the Spark model is well within 10%, matching that of Hadoop.

Fig. 2a also shows the memory elasticity profiles for two Tez reducers. The elasticity profile for Tez is similar to those for Spark and Hadoop. We extended our Hadoop reducer model to Tez reducers by accounting for the fact that in Tez, map outputs stored on the same node as the reducer are not added to shuffle memory but are instead read directly from disk. Fig. 1c shows that the accuracy of our Tez model is equally good.

Finally, Fig. 2a also shows one Flink reducer. Flink stands out with its low penalty between 70% and 99% of optimal memory which suggests a different model is needed. We plan to pursue this as part of our future work.

2.5 Spilling vs. paging

Why do frameworks implement spilling mechanisms and do not rely on tried-and-tested OS paging mechanisms for under-sized tasks? To answer, we provisioned Hadoop with enough memory to avoid spilling but configured cgroups such that part of the memory is available by paging to a swapfile. Fig. 2b shows the results for the Hadoop Wordcount reducer. Paging wins when a task gets allocated close to ideal memory (0.7 or more on the x-axis) because it only writes to disk the minimum necessary while Hadoop spills more than necessary. However, spilling beats paging for smaller memory allocations because the task's access pattern does not match the LRU order used by paging. Fig. 2b also shows that paging greatly increases garbage collection (GC) times because the GC touches memory pages in a paging-oblivious manner. We also see that the SSD significantly outperforms the HDD due to more efficient page-sized (4k) reads and writes. Using 2MB transparent huge pages (THP) did not improve results for either the SSD or HDD since THP is meant to alleviate TLB bottlenecks not improve IO throughput.

2.6 Memory elasticity and disk contention

Since memory elasticity leverages secondary storage, it is interesting to understand the impact of disk contention when several under-sized tasks are collocated.

The impact of disk contention depends on how well provisioned the local storage is on nodes relative to computing power. The ratio of cores to disks can give a sense of how many under-sized tasks can compete, in the worst case, for the same disk (a task usually requires at least one core). In current data centers the ratio is low. In [23], the authors mention ratios between 4:3 and 1:3 for a Facebook 2010 cluster. Public provider offerings also have low core to disk ratios. The list of high-end Nutanix hardware platforms [2] shows plenty of offerings with a ratio of less than 2.5:1 and as low as 0.66:1. Nutanix has more than two thousand small and medium size clusters at various enterprises [9].

Nevertheless, not all clusters are equally well provisioned. Thus, we analyzed the degree to which memory elasticity can produce disk contention by varying the number of under-sized Hadoop reducers that spill concurrently to the same disk. We start between 2 and 8 under-sized reducers each within 1 second of the previous. This is akin to analyzing disk contention on nodes with a core to disk ratio ranging from 2:1 to 8:1. We focused on reducers because they spill more data than the mappers (GBs is common).

We measured the slowdown in average reducer runtime when all reducers run concurrently compared to the case where they run sequentially. Fig. 2c shows the results. Several reducers (PageRank job1, Recommender

job1,2) show minimal slowdown (at most 15% degradation for 8 concurrently under-sized reducers). In the other extreme, running 8 under-sized WordCount reducers concurrently leads to a 70% degradation when an HDD is used but that is reduced to just 15% when moving the spills to SSD. In conclusion, disk contention is a manageable potential side effect but should nevertheless be taken into consideration when leveraging elasticity.

2.7 Final considerations

Does elasticity cause increased GC? For Hadoop and Tez reducers, GC times remain constant when tasks are under-sized. For Hadoop mappers, GC times slowly increase as the memory allocation decreases but they remain small in all cases. Overall, Hadoop does a good job of mitigating GC overheads by keeping data serialized as much as possible. For Spark, GC times increase sub-linearly with an increase in task runtime. Interestingly, GC times are a larger portion of task runtime for well-sized tasks because spill episodes limit the amount of data that needs to be analyzed for GC.

Feasibility of modeling Our models for Hadoop, Tez and Spark are based on two profiling runs, one under-sized and one well-sized. Related work shows that a large fraction of jobs in current data centers are recurring, have predictable resource requirements and compute on similar data [14, 20, 5]. Thus, instead of profiling runs, one can use prior runs of recurring jobs. Alternatively, if no prior runs are available, the two profiling runs can be performed efficiently on a sample of the job's input data.

Sensitivity to configuration changes We repeated our experiments on two different hardware platforms (Dual Intel Xeon E5-2630v3 + 40Gb NIC, Dual AMD Opteron 6212 + 10GB NIC), two OSes (RHEL 7, Ubuntu 14.04), three different disk configurations (HDD, SDD, 2*HDD in RAID 0), three IO schedulers (CFS, deadline, noop) and three JVMs (HotSpot 1.7, 1.8, OpenJDK 1.7). The changes did not impact the memory elasticity profiles or the accuracy of our model.

Elasticity of interactive workloads While interactive applications are far less tolerant to increases in runtime, they may still benefit from memory elasticity (e.g., by reducing queuing times in saturated clusters). Furthermore, it can be ensured that latency-sensitive jobs are not impacted negatively by providing additional job constraints (such as execution deadlines).

3 Applying elasticity to cluster scheduling. Case study: Apache YARN

In this section, we explore the benefits that memory elasticity can provide in cluster scheduling by integrating memory elasticity into Apache YARN [25]. We chose YARN because it is very popular and provides a common resource management layer for all popular frameworks tested in §2. Moreover, several recent research efforts from large Internet companies were validated with implementations on top of YARN [11, 21, 17, 18]. In addition, we also discuss how the elasticity principles can be adopted to the Mesos [19] resource manager.

Scheduling using memory elasticity is an NP-hard problem because it contains as a special case NP-hard variants of the RCPSP problem [8], a well-known problem in operations research. Nevertheless, we show that the benefits of memory elasticity can be unveiled even using a simple heuristic.

3.1 Overview

YARN distributes cluster resources to the jobs submitted for execution. A typical job may contain multiple tasks with specific resource requests. In YARN, each task is assigned to a single node, and multiple tasks may run concurrently on each node, depending on resource availability. The scheduler has a global view of resources and orders incoming jobs according to cluster policy (e.g., fair sharing with respect to resource usage).

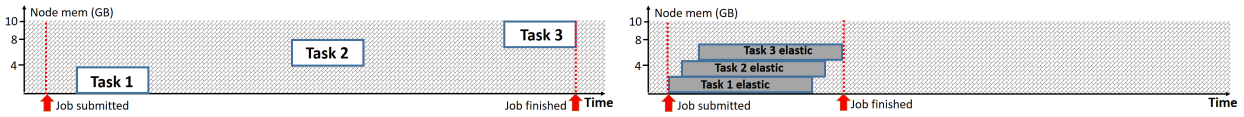
Notation We use the term *regular* to refer to the memory allocation and runtime of well-sized tasks and the term *elastic* for under-sized tasks. We further refer to our elasticity-aware implementation as YARN-ME.

Benefits As previously discussed, memory elasticity trades-off *task execution time* for *task memory allocation*. When applied to cluster scheduling it becomes a trade-off between *task queuing time* and *task completion time*. A task normally has to wait until enough memory becomes available for it but executing it with less memory may reduce or eliminate its waiting time. Since the completion time of a task is the sum of waiting time plus runtime, a significant decrease in waiting time may outweigh an increase in runtime due to elasticity and overall lead to faster task completion times. YARN-ME turns this task-level trade-off into improved job completion time and improved cluster-wide memory utilization.

Fig. 3 illustrates how memory elasticity benefits scheduling using a simple example of a 3-task job scheduled on a single, highly utilized node. Fig. 3a presents a timeline of task execution for vanilla YARN. Tasks 2 and 3 incur queuing times much larger than their execution times. In Fig. 3b, using memory elasticity, the scheduler launches all tasks soon after job submission, resulting in the job completing in less than 30% of its original time, despite its tasks now taking twice as long to execute.

3.2 System design

Two main additions are needed to leverage memory elasticity in YARN.



(a) All tasks *regularly* allocated.

(b) All tasks *elastically* allocated (with less memory).

Figure 3: Example of job completion time reduction for a simple 3-task job and one highly utilized node. In (a) the tasks wait for resources, while in (b) they start almost immediately. The job finishes faster, despite the longer tasks.

Metadata regarding task memory elasticity Reasoning about memory elasticity at the scheduler level leverages additional knowledge about the submitted tasks. The scheduler uses estimates for the regular execution time of a task (ideal duration), the minimal amount of memory for a regular allocation (ideal memory) and the performance penalty for under-sized memory allocations. This metadata is obtained using the profiling techniques from §2.7.

The timeline generator YARN-ME uses a timeline generator to provide an estimate of a job's evolution (the completion times of its tasks and of the whole job). In doing this, it accounts for the expected memory availability in the cluster. The generator simply iterates over all the nodes, adding up the task duration estimates of the executing and queued tasks. Effectively, the generator builds simple timelines for each node, which it then merges to obtain information about each job's timeline. The generator runs periodically, every heartbeat interval, since during such a period all healthy nodes report their status changes. It also runs when a new job arrives or an existing one is prematurely canceled.

3.3 Scheduler decision process

Algorithm 1 presents the decision process of YARN-ME. Lines 8-10 implement the main heuristic underlying YARN-ME (described below). Line 7 also implements disk contention awareness. Additionally, lines 7 and 9 always consider the minimum amount of memory that yields the lowest possible execution time, leveraging the behavior of elasticity described in §2.

Main heuristic YARN-ME aims to reduce job completion time by leveraging memory elasticity. As such, an elastic task cannot be allowed to become a straggler for its respective job. Therefore, *an elastic allocation is made for a task that cannot be scheduled regularly iff its expected completion time does not exceed the current estimated completion time of its job*.

Disk contention awareness As shown in §2.6 scheduling too many elastic tasks concurrently on a node may lead to disk contention. YARN-ME incorporates disk contention awareness. As shown in §2.6, obtaining the task metadata involves computing the disk bandwidth required by an elastic task. YARN-ME allocates a portion of each node's disk bandwidth for elastic tasks.

Algorithm 1 YARN-ME decision process pseudocode.

```
1:  while JOB QUEUE is not empty do
2:      for all N in NODES do
3:          J ← N's reserved job or next job in JOB QUEUE
4:          T ← next task of J
5:          if T regularly fits on N then
6:              allocate T on N, regular
7:          else if T elastically fits on N then
8:              get TIMELINE GENERATOR info for J
9:              if T elastically finishes before J then
10:                 allocate T on N, elastic
11:         if T was allocated then
12:             unreserve N if reserved
13:             resort the JOB QUEUE
14:         else
15:             reserve N for J, if not already reserved
```

It conservatively prohibits the scheduling of new elastic tasks on nodes where this portion would be exceeded.

Node reservations In YARN, if a node has insufficient resources to satisfy the job at the head of the queue, no allocation is performed, and the node is reserved for that job. While this job still has pending tasks, no other jobs can schedule tasks on the reserved node. This helps mitigate resource starvation by ensuring that the tasks of jobs with large memory requirements also get the chance to be scheduled. To account for this, we adjusted the *timeline generator* to take reservations into account when building its estimates. Additionally, YARN-ME allows tasks of other jobs to be allocated on a reserved node, but only if this does not hinder the tasks of the reserved job.

Additional constraints Schedulers may set additional constraints for their jobs, such as running on a data-local node only, or forcing certain tasks to start only after others have completed. Our design is orthogonal to this and only requires tweaking of the *timeline generator*.

4 Discussion: Mesos

Other schedulers beyond YARN can also be extended to use memory elasticity. We next review the main differences between Mesos [19] and YARN and argue that they do not preclude leveraging memory elasticity in Mesos.

Queuing policy Mesos uses Dominant Resource Fairness (DRF) [15], a multi-resource policy, to ensure fairness. Thus, the job queue may be sorted differently compared to YARN's policies. This does not restrict

memory elasticity as it only dictates job serving order.

Decision process Mesos decouples scheduling decisions from node heartbeats. Thus, a job may be offered resources from several nodes at the same time. This does not restrict memory elasticity since the job needs to consider each node from the offer separately (a task can only run on one node), so memory elasticity can be applied for every node in the offer.

Global vs. local decisions Mesos gives jobs the ability to accept or reject resource offers while YARN decides itself what each job receives. Thus, in Mesos, jobs can decide individually whether to use elasticity or not. If a decision based on global cluster information (like in YARN) is desired, jobs can express constraints (locality, machine configuration) with Mesos filters that can be evaluated by Mesos before making resource offers.

5 Cluster experiments

We next showcase the benefits of memory elasticity by comparing YARN-ME to YARN.

5.1 Methodology

Setup We use a 51-node cluster (50 workers and 1 master), limiting the scheduler to 14 cores per node (out of 16 cores we reserve 2 for the YARN NodeManager and for the OS) and 10GB of RAM. The exact amount of RAM chosen is not important (we could have chosen any other value). What is important is the ratio of ideal task memory requirements to node memory. Each node has one 2 TB SATA HDD. YARN-ME was implemented on top of Apache YARN 2.6.3 [25]. Disk spills use Direct I/O so that the OS buffer cache does not mask performance penalties due to elasticity.

We ran WordCount, PageRank and Mahout Item Recommender Hadoop applications. We chose them because they represent small, medium and large penalties encountered for reducers in §2 (mapper penalties span a much smaller range than reducers and are lower). We configured the jobs as described in Table 1. We executed each type of application separately (homogeneous workload) and all applications simultaneously (heterogeneous workload). For the homogeneous workloads, we varied the number of concurrent runs for each type of application. The start of each run is offset by the inter-arrival (IA) time indicated in Table 1. The IA time is chosen proportionally to application duration such that map and reduce phases from different jobs can overlap.

For each application we first perform one profiling run using ideal memory to obtain the ideal task runtime. We multiply this by the penalties measured in §2 to obtain task runtimes for various under-sized allocations.

Metrics We compare average job runtime, trace makespan and average duration of *map* and *reduce* phases. By *job runtime* we mean the time between job submission and the end of the last task in the job. Similarly, a map or reduce phase represents the time elapsed between the first request to launch such a task and the finish time of the last task in the phase. Each experiment is run for 3 iterations, and we report the average, minimum and maximum values.

5.2 Experiments

Benefits for memory utilization Fig. 4a shows the benefits of using memory elasticity on both cluster utilization and makespan, for an execution of 5 Pagerank runs. YARN-ME successfully makes use of idle memory, bringing total memory utilization from 77% to 95%, on average, and achieving a 39% reduction in makespan. By comparing the results in Figs. 4a and 4c we also see that the gain in job runtime is proportionally much higher than the amount of memory reclaimed. Fig. 4b shows how YARN-ME assigns the memory slack to tasks.

Benefits for homogeneous workloads We next show that YARN-ME can provide benefits for the jobs in Table 1. Figs. 4c, 5a and 5b show the improvement of YARN-ME compared to YARN vs. the number of runs. YARN-ME's benefits hold for all jobs.

We find that the Recommender, which has the highest penalties we have observed for reducers, achieves up to 48% improvement. We also find that mappers always benefit noticeably from elasticity, a direct consequence of their modest penalties. Pagerank's lower-penalty reducers yield an improvement of 30% even for a single concurrent run, peaking at 39% with 5 runs. Wordcount achieves a peak improvement of 41%, despite reducer gains being lower due to higher penalties. The reduction in average job runtime steadily increases across runs. For 3 concurrent Wordcount runs, the number of reducers leads only one out of the 3 jobs to be improved, but the map phase still reaches improvements of 46%.

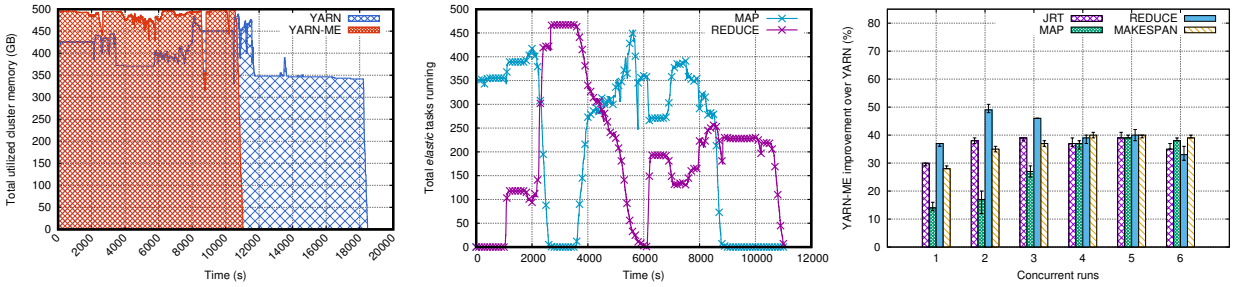
Benefits for heterogeneous workloads YARN-ME achieves considerable gains even under a heterogeneous workload composed of all the jobs from Table 1. We start 5 jobs at the same time (1 Pagerank, 1 Recommender and 3 Wordcount) and then submit a new job every 5 minutes, until we reach a total of 14 jobs (3 Pagerank, 3 Recommender and 8 Wordcount). Each job is configured according to Table 1. Fig. 5c shows overall improvement and breakdown by job type. YARN-ME improves average job runtime by 60% compared to YARN. The *map* phase duration is reduced by 67% on average overall, and by up to 72% for all Recommender jobs.

6 Simulation experiments

We use simulations to evaluate YARN-ME's benefits and its robustness to mis-estimations on a wide range of workload configurations.

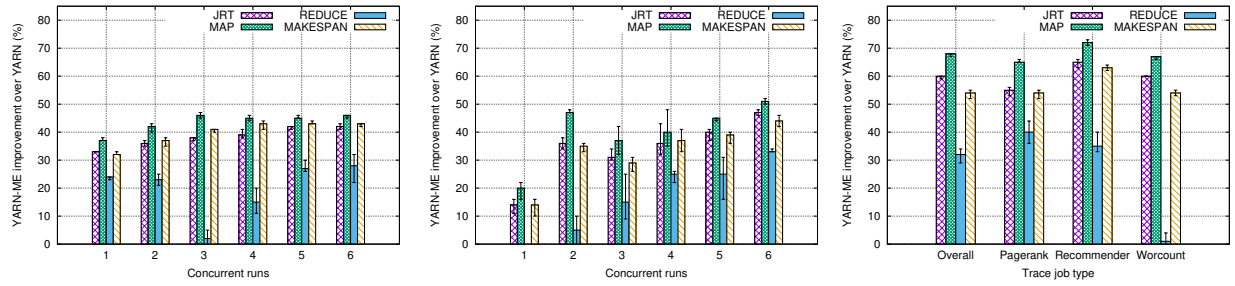
Application	Jobs	Input GB	maps	reduces	Penalties				Memory GB				Inter-arrival (IA) time
					1st job		2nd job		1st job		2nd job		
					map	reduce	map	reduce	map	reduce	map	reduce	
Pagerank	2	550	1381 / 1925	275	1.3	1.22	1.25	1.75	1.7	3.7	1.5	6.8	120s
WordCount	1	540	2130	75	1.35	1.9	-	-	1.7	5.4	-	-	30s
Recommender	2	250	505 / 505	100	1.3	2.6	1.3	3.3	2.4	2.8	2.4	3.8	120s

Table 1: Characteristics of the evaluated applications.



(a) Cluster memory utilization – 5 runs (b) *elastic* tasks – 5 runs (c) Improvement over YARN

Figure 4: YARN-ME vs. YARN for running Pagerank on 50 nodes. Fig. 4a shows the timeline of cluster memory utilization. Fig. 4b shows the timeline of tasks scheduled *elastically*. Fig. 4c reports improvement w.r.t. *average job runtime* (JRT), *average job phase time* (map, reduce), and *makespan*.



(a) WordCount (b) Recommender (c) Mixed job trace – 14 jobs

Figure 5: Improvement of YARN-ME over YARN for various applications on 50 nodes w.r.t. *average job runtime* (JRT), *average job phase runtime* (map, reduce), and *makespan*. We report average, min. and max. over 3 iterations. Fig. 5c reports results for a mixed trace of jobs: 3x Pagerank, 3x Recommender, 8x Wordcount.

6.1 Simulation Methodology

Simulator We built DSS (Discrete Scheduler Simulator), a discrete-time simulator for YARN, and we made the code publicly available [1]. In DSS, simulated tasks do not perform computation or I/O. The tasks are simulated using task start and task finish events. We simulate a cluster with 16 cores and 10GB of RAM per node. Memory is assigned to tasks with a granularity of 100MB. Jobs are ordered according to YARN's fair-scheduling policy [1]. Each task uses 1 core. The minimum amount of memory allocatable to a task is set to 10% of its ideal requirement. We use a 100-node cluster to perform a parameter sweep but also show results for up to 3000 nodes.

Simulation traces A trace contains for each job: the job submission time, the number of tasks, the ideal amount of memory for a task, and task duration. Each job has one parallel phase. Job arrivals are uniformly random between 0 and 1000s. The other parameters are varied according to either a uniform or an exponential random distribution. We use 100-job traces but also show results for up to 3000 jobs.

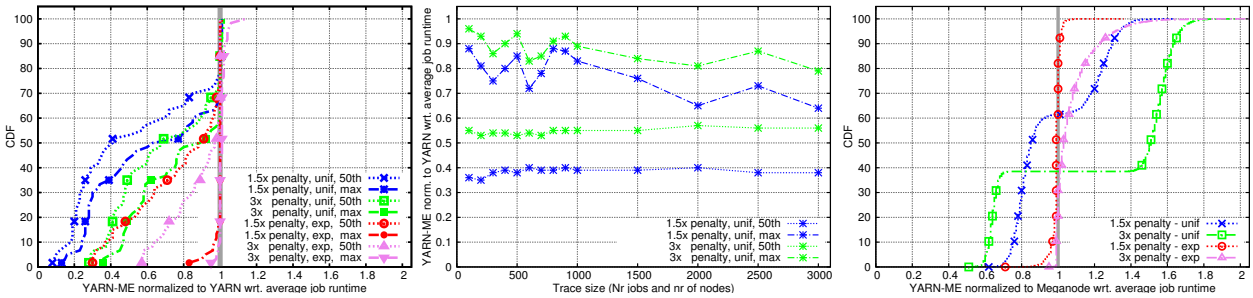
Modeling elasticity Since the simulated jobs have only a single phase we only use the reducer penalty model from §2. We show results for 1.5x and 3x penalties, to cover the range of penalties measured in §2.

Metrics We use average job runtime to compare the different approaches.

6.2 Simulation experiments

YARN-ME vs. YARN We perform a parameter sweep on 3 trace parameters: memory per task, tasks per job and task duration. Table 2 shows the different parameter ranges. We keep the min constant and vary the max within an interval to perform the sweep. This gives us a

[1] https://github.com/epfl-labos/DSS



(a) YARN-ME vs. YARN (avg job runtime) (b) Impact of scaling trace size (c) YARN-ME vs. Meganode (avg job runtime)

Figure 6: CDFs of average job runtime in YARN-ME vs. YARN (a) and vs. the idealized Meganode (c). YARN's benefits hold for traces of varying size (b).

range for each parameter, which is then varied independently of the others. We draw the values from a uniform or exponential random distribution. We perform 100 runs for each combination of 3 ranges (one for each parameter) and show the median and the maximum (worst-case) results for normalizing YARN-ME to YARN in Fig. 6a. We use 100-job traces on 100 nodes.

dist	tasks / job		task mem (GB)		task duration (s)	
	min	max	min	max	min	max
unif	1	[200,400]	1	[2,10]	1	[200,500]
exp	1	[20,220]	1	[2,10]	50	[100,500]

Table 2: Trace parameter sweep ranges.

The uniform distribution yields bigger benefits because it leads to more memory fragmentation in YARN. As expected, YARN-ME's improvements are larger if penalties are lower. The cases in which YARN-ME does not improve on YARN are either when the cluster utilization is very low or when most tasks have very small memory requirements. In such cases, memory elasticity is less beneficial. Nevertheless, for 3x penalty and a uniform distribution, 40% of the configurations have a ratio of YARN-ME to YARN of at most 0.7.

Fig. 6b shows the behavior of one uniform trace in a weak scaling experiment. We scale the trace and cluster size simultaneously from 100 to 3000. The benefits of YARN-ME hold despite the scaling.

The need for elasticity (YARN-ME vs. Meganode) We next show that YARN-ME yields improvements beyond the reach of current elasticity-agnostic schedulers. We compare against an idealized scenario (called Meganode) which serves as an upper-bound for elasticity-agnostic solutions that improve average job runtime. The Meganode pools all cluster resources into one large node with a memory and core capacity equal to the aggregate cluster-wide core and memory capacity available for YARN-ME. Thus, the Meganode does away with machine-level memory fragmentation. Meganode uses a shortest remaining job first (SRJF) scheduling policy because that is known to improve average job runtime. However, in using SRJF, YARN-ME obeys the existing fairness policy whereas Meganode does not.

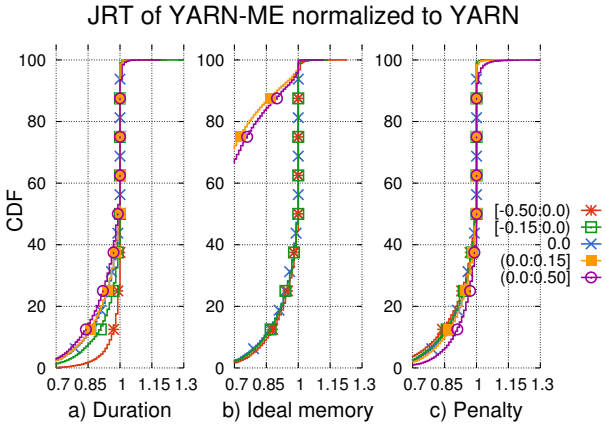


Figure 7: Sensitivity to mis-estimations. 3x penalty.

Fig. 6c compares the averge job runtime for Meganode and YARN-ME on 20.000, 100-job traces on 100 nodes. While it is expected that Meganode wins in many cases, YARN-ME beats Meganode for 40%-60% of the cases for the uniform trace and for 20% of the exponential trace for 1.5x penalty. YARN-ME gains because it turns even small amounts of memory fragmentation into an opportunity by scheduling elastic tasks.

Sensitivity to mis-estimations Further, we show that YARN-ME is robust to mis-estimations. We generate 20,000 traces with each of the trace parameters (memory per task, tasks per job, and task duration) following an exponential random distribution, within bounds of $[0.1,10]$ GBs, $[1,100]$ tasks, and $[50,500]$ seconds. We simulate mis-estimations by altering the duration, ideal memory, and performance penalty of tasks for both regular and elastic allocations. This forces the scheduler to make decisions based on imperfect information. We change each parameter fractionally by a uniformly random factor in the intervals of $(0,0.15]$, and $(0,0.5]$ (0.15 represents a 15% higher value). The former interval represents the worst-case deviation of our model in Fig. 1c,

while the latter is an extreme example chosen to stress YARN-ME. We present both positive and negative mis-estimations. Fig. 7 presents the ratio between average job completion time with YARN-ME and YARN, for a 3x penalty – one of the highest penalties measured in §2.

Sensitivity to task duration mis-estimation YARN-ME is robust to task duration mis-estimation, which can occur due to system induced stragglers or data locality. The timeline generator of the simulator bases its information on task durations from the trace. We alter each actual task runtime by a different factor.

For $[-0.15, 0.5]$, YARN-ME achieves gains similar to the scenario without mis-estimations on all traces. Even for the very large $[-0.5, 0)$ mis-estimations, the gains are still comparable, with only $\sim$35% of the traces reporting at most 10% lower gains. This is due to tasks being shorter than the timeline generator expects, resulting in a few elastic tasks exceeding the estimated job finish time.

Sensitivity to model mis-estimations YARN-ME is also robust to model mis-estimations, which may occur during profiling. We change task memory (Fig. 7b) and penalty (Fig. 7c) by a different value for each job.

YARN-ME improves by up to 45% in the case of positive mis-estimation of ideal memory (Fig. 7b). In this case, all tasks (in both YARN and YARN-ME) spill data to disk and become penalized tasks. However, penalties in YARN-ME are lower because YARN-ME can choose the under-sized allocation that minimizes penalty while YARN lacks this capability. Negative mis-estimation of ideal memory has negligible impact.

In the case of penalty mis-estimation (Fig. 7c), only the $(0, 0.5]$ runs exhibit gains reduced by at most 4%. This is due to the scheduler being more conservative since it perceives elastic tasks as taking longer.

7 Related Work

Current schedulers do not leverage memory elasticity. Next, we review the most related mechanisms employed by current schedulers.

Tetris [16] improves resource utilization (including memory) by better packing tasks on nodes. It adapts heuristics for the multi-dimensional bin packing problem to the context of cluster scheduling. However, it only schedules a task on a node that has enough memory available to cover its estimated peak memory requirements.

Heracles [22] aggressively but safely collocates best-effort tasks alongside a latency critical service. It does this by dynamically managing multiple hardware and software mechanisms including memory. However, Heracles only considers RAM bandwidth and not capacity.

Apollo [7] is a distributed scheduler that provides an opportunistic scheduling mode in which low priority tasks can be scheduled using left-over memory unused by normal priority tasks. Normal priority tasks are scheduled only if their resource demands are strictly met. Apollo has no principled way of reasoning about the performance implications of opportunistic allocations nor does it provide a decision mechanism about when such allocations are useful. Borg [26] provides similar capabilities with a centralized design.

Quasar [12] leverages machine-learning classification techniques to reason about application performance with respect to scale-up allocations. A greedy algorithm places tasks starting with nodes that give the best performance satisfying application SLOs and improving resource utilization. Quasar does not identify nor discusses memory elasticity.

ITask [13] is a new type of data-parallel task that can be interrupted upon memory pressure and have its memory reclaimed. The task can then be resumed when the pressure goes away. ITask is a system-level mechanism that uses preemption to mitigate unmanageable memory pressure before it can hurt system performance. Memory elasticity can work in tandem with ITask, since elastic tasks will need less time to spill, and thus can be preempted and resumed faster than regular tasks.

8 Conclusion

The main contribution of this paper is identifying, quantifying and demonstrating memory elasticity, an intrinsic property of data-parallel workloads. Memory elasticity allows tasks to run with significantly less memory than ideal while incurring only a moderate performance penalty. We show that memory elasticity is prevalent in the Hadoop, Spark, Tez and Flink frameworks. We also show its predictable nature by building simple models for Hadoop and extending them to Tez and Spark. Applied to cluster scheduling, memory elasticity helps reduce task completion time by decreasing task waiting time for memory. We show that this can be transformed into improvements in job completion time and cluster-wide memory utilization. We integrated memory elasticity into Apache YARN and showed up to 60% improvement in average job completion time on a 50-node cluster running Hadoop workloads.

Acknowledgements We thank our anonymous reviewers and our shepherd, Christina Delimitrou, for valuable feedback and advice. We thank Laurent Bindschaedler, Diego Didona, Christos Gkantsidis, Ashvin Goel, Sergey Legtchenko, Baptiste Lepers, T. S. Eugene Ng, Amitabha Roy, and Yiting Xia for feedback and suggestions on earlier drafts of the paper and insightful discussions. We also thank Rolf Möhring, Roel Leus, and Sigrid Knust for valuable explanations regarding RCPSP, and Adrian Popescu for providing the TPC-DS queries.

References

[1] Hadoop MapReduce Next Generation - Fair Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[2] Hardware Platform Specifications. http://www.nutanix.com/products/hardware-platforms/.

[3] Improving Sort Performance in Apache Spark: It's a Double. http://blog.cloudera.com/blog/2015/01/improving-sort-performance-in-apache-spark-its-a-double/.

[4] The Bigdata Micro Benchmark Suite. https://github.com/intel-hadoop/HiBench.

[5] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *Proc. NSDI 2012*.

[6] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *Proc. USENIX ATC 2014*.

[7] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proc. OSDI 2014*.

[8] P. Brucker, A. Drexi, R. Moerhing, K. Neumann, and E. Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. In *European Journal of Operational Research, Volume 112, Issue 1, 1 January 1999, Pages 3–41*.

[9] I. Cano, S. Aiyar, and A. Krishnamurthy. Characterizing private clouds: A large-scale empirical analysis of enterprise clusters. In *Proc. SoCC 2016*.

[10] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. In *Proc. VLDB 2012*.

[11] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proc. SoCC 2014*.

[12] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proc. ASPLOS 2014*.

[13] 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. SOSP 2015*.

[14] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proc. EuroSys 2012*.

[15] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proc. NSDI 2011*.

[16] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proc. SIGCOMM 2014*.

[17] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proc. OSDI 2016*.

[18] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. OSDI 2016*.

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

[20] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proc. SIGCOMM 2015*.

[21] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated SLOs for enterprise clusters. In *Proc. OSDI 2016*.

[22] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proc. ISCA 2015*.

[23] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics framework. In *Proc. NSDI 2015*.

[24] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. SoCC 2012*.

[25] 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 *Proc. SOCC 2013*.

[26] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proc. EuroSys 2015*.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI 2012*.

[28] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Proc. NSDI 2012*.