RESEARCH

Tech Report #2019-05: Performance Interference of Big Data Frameworks in Resource Constrained Clouds

Stratos Dimopoulos*, Chandra Krintz and Rich Wolski

*Correspondence:
stratos@cs.ucsb.edu
Department of Computer Science, University of California, Santa Barbara, USA
Full list of author information is available at the end of the article

Abstract

In this paper, we investigate and characterize the behavior of "big" and "fast" data analysis frameworks, in multi-tenant, shared settings for which computing resources (CPU and memory) are limited. Such settings and frameworks are frequently employed in both public and private cloud deployments. Resource constraints stem from both physical limitations (private clouds) and what the user is willing to pay (public clouds). Because of these constraints, users increasingly attempt to maximize resource utilization and sharing in these settings.

To understand how popular analytics frameworks behave and interfere with each other under such constraints, we investigate the use of Mesos to provide fair resource sharing for resource constrained private cloud systems. We empirically evaluate such systems using Hadoop, Spark, and Storm multi-tenant workloads. Our results show that in constrained environments, there is significant performance interference that manifests in multiple ways. First, Mesos is unable to achieve fair resource sharing for many configurations. Moreover, application performance over competing frameworks depends on Mesos offer order and is highly variable. Finally, we find that resource allocation among tenants that employ coarse-grained and fine-grained framework scheduling, can lead to a form of deadlock for fine-grained frameworks and underutilization of system resources.

Keywords: Big data; multi-tenancy; performance interference; Hadoop; Spark

Introduction

Data-driven actuation, decision support, and adaptive control is experiencing explosive growth as a result of recent technological advances in environmental and personal monitoring, sensing, and data analytics (e.g. Internet of Things (IoT)) coupled with the wide availability of low cost compute, storage, and networking (e.g. cloud computing). As a result, there is significant demand by software engineers, data scientists, and analysts with a variety of backgrounds and expertise, for extracting actionable insights from this data. Such data has the potential for facilitating beneficial decision support for nearly every aspect of our society and economy, including social networking, health care, business operations, the automotive industry, agriculture, Information Technology, education, and many others.

To address this need, a number of open source technologies have emerged that make effective, large-scale data analytics accessible to the masses. These include "big data" and "fast data" analysis systems such as Hadoop [1], Spark [2], and Storm [3] from the Apache foundation, which are used by analysts to implement a variety of applications for query support, data mining, machine learning, real-time stream analysis, statistical analysis, and image processing [4–7]. As complex software systems, with many installation, configuration, and tuning parameters, these frameworks are often deployed under the control of a distributed resource management system [8, 9] to decouple resource management from job scheduling and monitoring, and to facilitate resource sharing between multiple frameworks.

Each of these analytics frameworks tends to work best (e.g. most scalable, with the lowest turn-around time, etc.) for different classes of applications, data sets, and data types. For this reason, users are increasingly tempted to use multiple frameworks, each implementing a different aspect of their analysis needs. This new form of multi-tenancy (i.e. multi-analytics) gives users the most choice in terms of extracting potential insights, enables them to fully utilize their compute resources and, when using public clouds, manage their fee-for-use monetary costs.

Multi-analytics frameworks have also become part of the software infrastructure available in many private data centers and, as such, must function when deployed on a private cloud [10–12]. With private clouds, resources are restricted by physical limitations. As a result, these technologies are commonly employed in shared settings in which more resources (CPU, memory, local disk) cannot simply be added on-demand in exchange for an additional charge (as they can in a public cloud setting).

Because of this trend, in this paper, we investigate and characterize the performance and behavior of big/fast data systems in shared (multi-tenant), moderately resource constrained, private cloud settings. While these technologies are typically designed for very large scale deployments such as those maintained by Google, Facebook, and Twitter they are also common and useful at smaller scales [13–15].

We empirically evaluate the use of Hadoop, Spark, and Storm frameworks in combination, with Mesos [9] to mediate resource demands and to manage sharing across these big data tenants. Our goal is to understand

- How these frameworks interfere with each other in terms of performance when they are deployed under resource pressure,
- How Mesos behaves when demand for resources exceeds resource availability, and
- The degree to which Mesos is able to achieve fair sharing using Dominant Resource Fairness (DRF) [16] in resource restricted cloud settings.

From our experiments and analyses, we find that even though Spark outperforms Hadoop when executed in isolation for a set of popular benchmarks, in a multi-tenant system, their performance varies significantly depending on their respective scheduling policies and the timing of Mesos resource offers. Moreover, for some combinations of frameworks, Mesos is unable to provide fair sharing of resources and/or avoid deadlocks. In addition, we quantify the framework startup overhead and the degree to which it affects short-running jobs.

Background

In private cloud settings, where users must contend for a fixed set of data center resources, users commonly employ the same resources to execute multiple analytics systems to make the most of the limited set of resources to which they have been granted access. To understand how these frameworks interfere in such settings, we

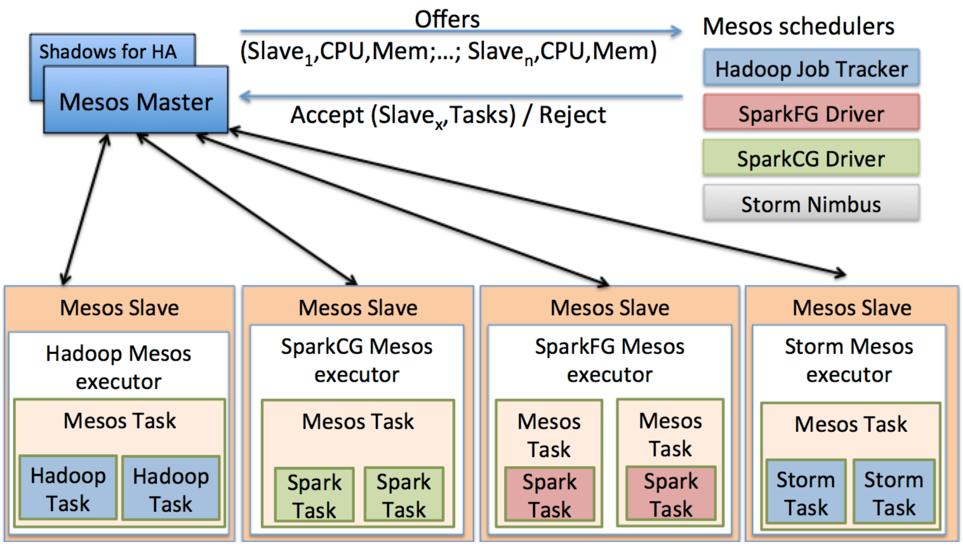


Figure 1: **Mesos Architecture.** Each distributed computing framework implements a Mesos scheduler which negotiates with the Mesos Master for Slave resource allocation, and a Mesos executor inside which framework tasks are spawned. Frameworks launch tasks in Coarse-Grained (CG) mode (1 Mesos task corresponds to 1+ framework tasks) or Fine-Grained (FG) mode (1 Mesos task corresponds to 1 framework task). Spark can use either CG or FG modes; Hadoop and Storm use CG.

investigate the use of Mesos to manage them and to facilitate fair sharing. Mesos is a cluster manager that can support a variety of distributed systems including Hadoop, Spark, Storm, Kafka, and others [9, 17]. The goal of our work is to investigate the performance implications associated with Mesos management of multi-tenancy for medium and small scale data analytics on private clouds.

We first overview the implementation of Mesos and its support for the analytics frameworks that we consider in this study: Hadoop, Spark, and Storm. Figure 1 provides a high level overview of the Mesos software architecture. Mesos provides two-level, offer-based, resource scheduling for frameworks. The Mesos Master is a daemon process that manages a distributed set of Mesos Slaves. The Master also makes offers containing available Slave resources (e.g. CPUs, memory) to registered frameworks. Frameworks accept or reject offers based on their own, local scheduling policies and control execution of their own tasks on Mesos Slaves that correspond to the offers they accept.

When a framework accepts an offer, it passes a description of its tasks and the resources it will consume to the Mesos Master. The Master (acting as a single contact point for all framework schedulers) passes task descriptions to the Mesos Slaves. Resources are allocated on the selected Slaves via a Linux container (the Mesos executor). Offers correspond to generic Mesos tasks, each of which consumes the CPU and memory allocation specified in the offer. Each framework uses a Mesos Task to launch one or more framework-specific tasks, which use the resources in the accepted offer to execute an analytics application.

Each framework can choose to employ a single Mesos task for each framework task, or use a single Mesos task to run multiple framework tasks. We will refer to the former as "fine-grained mode" (FG mode) and the later as "coarse-grained mode" (CG mode). CG mode amortizes the cost of starting a Mesos Task across multiple framework tasks. FG mode facilitates finer-grained sharing of physical resources.

The Mesos Master is configured so that it executes on its own physical node and with high availability via shadow Masters. The Master makes offers to frameworks using a pluggable resource allocation policy (e.g. fair sharing, priority, or other). The default policy is Dominant Resource Fairness (DRF) [16]. DRF attempts to fairly allocate combinations of resources by prioritizing the framework with the minimum *dominant share* of resources.

The dominant resource of a framework is the resource for which the framework holds the largest fraction of the total amount of that resource in the system. For example, if a framework has been allocated 2 CPUs out of 10 and 512MB out of 1GB of memory, its dominant resource is memory ($2/10\ CPUs < 512/1024\ memory$). The dominant share of a framework is the fraction of the dominant resource that it has been allocated (512/1024 or 1/2 in this example). The Mesos Master makes offers to the framework with the smallest dominant share of resources, which results in a fair share policy with a set of attractive properties (share guarantee, strategy-proofness, Pareto efficiency, and others) [16]. We employ the default DRF scheduler in Mesos for this study.

The framework implementations that we consider include the open source analytics systems Apache Hadoop [1], Apache Spark [2], and Apache Storm [3]. Hadoop implements the popular MapReduce programming model via a scalable, fault tolerant and, distributed batch system. Spark extends this model and system with an in-memory data-structure for Resilient Distributed Datasets (RDDs) [18]. Finally, Storm provides distributed, fault tolerant, real-time processing of streaming data. All frameworks leverage the Hadoop Distributed File System (HDFS) [19] for data persistence and durability. Each of these frameworks makes different design and implementation trade-offs (which result in different strengths and weaknesses), each is amenable to varying types of big data processing, analysis, and programming models, and each have had numerous applications written for them by developers [20–22]. For example, other studies show that Spark is much faster than Hadoop under normal conditions [23,24], but that Hadoop has better fault-tolerance characteristics [23]. Moreover, Spark is using Resilient Distributed Datasets that make it a better choice compared to Hadoop for iterative algorithms, as it avoids repeated and costly reads and writes to/from HDFS, but this same mechanism is what slows it down compared to Hadoop when there is no data re-use on the workflow [6] or when data shuffling efficiency determines the performance [23].

In a Mesos deployment, each framework implements the Mesos scheduler interface and the Mesos executor interface. For Hadoop, the scheduler corresponds to the Hadoop *JobTracker* and the executor is a Hadoop *TaskTracker*. For Spark, the scheduler corresponds to the Spark *Driver* and there is a Spark extension that implements the executor for task management. For Storm, the scheduler is called *Nimbus* and the Mesos executors correspond to the Storm *Supervisors*. Storm Supervisors spawn one or more Storm *workers*, each of which executes one or more application tasks as process threads.

Each framework creates one Mesos executor and one or more Mesos Tasks on each Slave in an accepted Mesos offer. In CG mode, frameworks release resources back to Mesos when all tasks complete or when the application is terminated. In FG mode, frameworks execute one application task per Mesos task. When a framework task completes, the framework scheduler releases the resources associated with the task back to Mesos. The framework then waits until it receives a new offer with sufficient resources from Mesos to execute its next application task. In our experiments we consider Spark, which provides both FG and CG modes as options, and Hadoop and Storm, which employ CG mode.

Throughout this paper, we use the term "tenant" to refer to a framework (e.g. Spark, Hadoop, Storm, etc., that uses Mesos for cluster management. Thus, in single tenant scenarios, one framework submits a job and makes use of all the available cloud resources. Multi-tenancy refers to more than one framework running jobs at the same time, while sharing the same cluster resources managed by Mesos. Mesos applies its DRF policy to share the cluster resources and does not differentiate based on the number of frameworks that use the cluster.

Mesos supports Roles [25] to statically separate resources between multiple frameworks. There is no dynamic sharing of resources between different roles, but instead the total cluster capacity has to be divided across the frameworks. A static assignment of resources limits the peak capacity a cluster can support and wastes the resources when a frameworks is idle. These disadvantages are important in resource constrained environments, where the peak capacity and the available resources are already limited. Therefore, we did not make use of Mesos roles and instead investigate dynamic resource allocation sharing among frameworks.

Experimental Methodology

We next describe the experimental setup that we use for this study. We detail our hardware and software stack, overview our applications and data sets, and present the framework configurations that we consider.

We employ two, resource-constrained, Eucalyptus [11] private clouds, each with nine virtual servers (nodes). We use three nodes for Mesos Masters that run in high availability mode (similar to typical fault-tolerant settings of most real systems) and six for Mesos Slaves in each cloud. The Slave nodes on the first cloud (Eucalyptus v3.4.1), to which we refer to as *development*, each have 2x2.5GHz CPUs, 4GB of RAM, and 60GB disk space. The Slave nodes on the second, *production* cloud (Eucalyptus v4.1), have 4x3.3GHz CPUs, 8GB of RAM, and 60GB of SSD disk. Both clouds use Gigabit Ethernet switches.

Our nodes run Ubuntu v12.04 Linux with Java 1.7, Mesos 0.21.1 [26] which uses Linux containers by default for isolation, the CDH 5.1.2 MRv1 [27] Hadoop stack (HDFS, Zookeeper, MapReduce, etc.), Spark v1.2.1 [28], and Storm v0.9.2 [29]. We configure Mesos Masters (3), HDFS Namenodes, and Hadoop JobTrackers to run with High Availability via three Zookeeper nodes co-located with the Mesos Masters. HDFS uses a replication factor of three and 128MB block size. In addition, we found and fixed a number of bugs that prevented the Hadoop stack from executing in our environment. Our modifications are available at [30].

Our batch processing workloads and data sets come from the BigDataBench and the Mahout projects [4, 31]. We have made minor modifications to update the algorithms to have similar implementations across frameworks (e.g. when they read-/write data, perform sorts, etc.). These modifications are available at [32]. In this study, we employ WordCount, Grep, and Naive Bayes applications for Hadoop and Spark and a WordCount streaming topology for Storm. WordCount computes the number of occurrences of each word in a given input data set, Grep produces a count of the number of times a specified string occurs in a given input data set, and Naive Bayes performs text classification using a trained model to classify sentences of an input data set into categories.

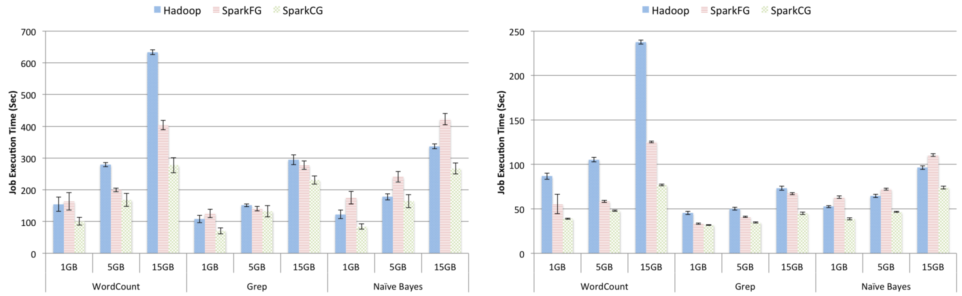
We execute each application 10 times after three warmup runs to eliminate variation due to dynamic compilation by the Java Virtual Machine and disk caching artifacts. We report the average and standard deviation of the 10 runs. We keep the data in place in HDFS across the system for all runs and frameworks to avoid variation due to changes in data locality. We measure performance and interrogate the behavior of the applications using a number of different tools including Ganglia [33], ifstat, iostat, and vmstat available in Linux, and log files available from the individual frameworks.

		Development Cloud		Production Cloud	
		CPU	MEM	CPU	MEM
Available	Slave	2	2931	4	6784
	Total	12	17586	24	40704
Min Required	Hadoop	1	980	1	980
	Spark	2	896	2	896
	Storm	2	2000	2	2000
Max Used Per Slave	Hadoop	2	2816	4	5888
	Spark	2	896	4	896
	Storm	2	2000	4	4000
Max Used Total	Hadoop	12	16896	24	35328
	Spark	12	5376	24	5376
	Storm	6	6000	6	6000

Table 1: CPU and Memory availability, minimum framework requirements to run 1 Mesos Task and maximum utilized resources per slave and in total.

Table 1 shows the available resources in our two private cloud deployments, the minimum required resources that should be available on a slave for a framework to run at least one task on Mesos and, the maximum resources that can be utilized when the framework is the only tenant on the cloud. We configure the Hadoop TaskTracker with 0.5 CPUs and 512MB of memory and each slot with 0.5 CPUs, 768MB of memory, and 1GB of disk space. We set the minimum and maximum map/reduce slots to 0 and 50, respectively. We configure Spark tasks to use 1 CPU and 512MB of memory, which also requires an additional 1 CPU and 384MB of memory for each Mesos executor container. We enable compression for event logs in Spark and use the default MEMORY ONLY caching policy. Finally, we configure Storm to use 1 CPU and 1GB memory for the Mesos executor (a Storm Supervisor) and 1 CPU and 1GB memory for each Storm worker.

This configuration allows Hadoop to run 3 and 7 tasks per Mesos executor for the development and production cloud, respectively. Hadoop spawns one Mesos executor per Mesos Slave and Hadoop tasks can be employed as either mapper



(a) Development Cloud (b) Production Cloud

Figure 2: **Single Tenant Performance:** Benchmark execution time in seconds for Hadoop and Spark on Mesos for different input sizes. In this experiment, we execute each job using Hadoop, Spark in coarse grained mode (SparkCG), and Spark in fine grained mode (SparkFG), using the development cloud (Figure 2a) and production cloud (Figure 2b).

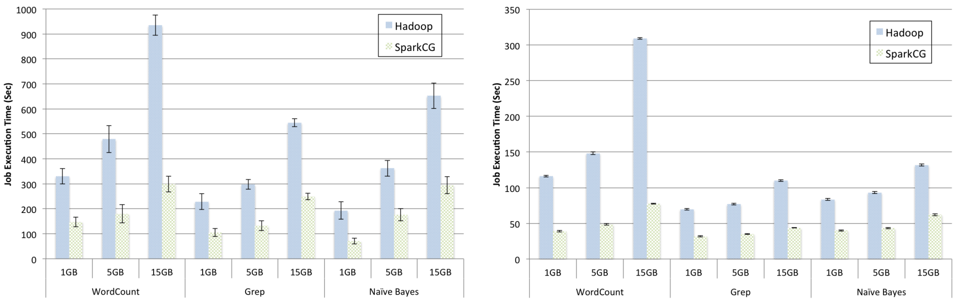
or reducer slots. Spark in FG mode runs 1 Mesos/Spark task per executor on the development cloud and 3 Mesos/Spark tasks per executor on the production cloud. In CG mode, Spark allocates its resources to a single Mesos task per executor that runs all Spark tasks within it. In both modes, Spark runs one executor per Mesos Slave. We configure the Storm topology to use 3 workers. On the development cloud 1 Supervisor (Mesos executor) that runs 1 worker fits per slave and therefore 3 Slaves are needed in total. On the production cloud up to 3 workers can fit in the same Supervisor and therefore the Storm topology can be deployed in 1 Slave or distributed in multiple Supervisors across Slaves. We consider three different input sizes for the applications to test for small, medium and long running jobs. As the number of tasks per job is determined by the HDFS block size (which is 128MB), the 1GB input size corresponds to 8 tasks, the 5GB input size to 40 tasks and, the 15GB input size to 120 tasks.

Results

For the first set of experiments, we use this experimental setup to measure the performance of Hadoop and Spark when they run in isolation (single tenancy) on our Mesos-managed private clouds. Throughout the remainder of this paper, we refer to Spark when configured to use FG mode as *SparkFG* and when configured to use CG mode as *SparkCG*.

Figure 2 presents the execution time for the three applications for different data set sizes (1GB, 5GB, and 15GB) for the development cloud (left graph) and production cloud (right graph). These results serve as a reference for the performance of the applications when there is no resource contention (no sharing) across frameworks in our configuration.

The performance differences across frameworks are similar to those reported in other studies, in which Spark outperforms Hadoop (by more than 2x in our case) [23, 24]. One interesting aspect of this data is the performance difference between SparkCG and SparkFG. SparkCG outperforms SparkFG in all cases and by



(a) Development Cloud (b) Production Cloud

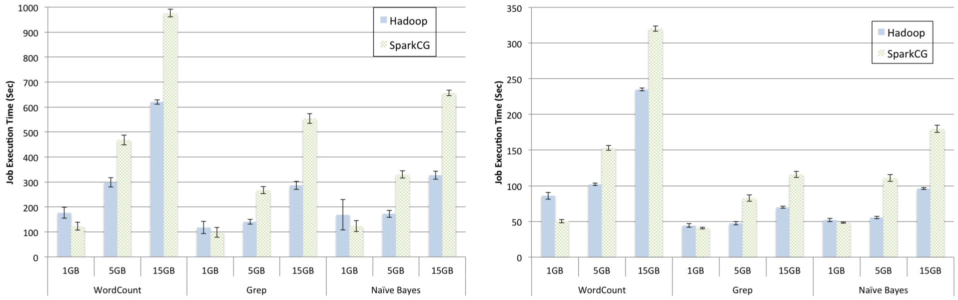
Figure 3: **Multi-tenant Performance:** Benchmark execution time in seconds for Hadoop and SparkCG using different input sizes deployed on the development cloud (Figure 3a) and the production cloud (Figure 3b). In this setting, SparkCG receives offers from Mesos first because it is able to setup the application faster than Hadoop is able to.

up to 2x in some cases. The reason for this is that SparkFG starts a Mesos Task for each new Spark task to facilitate sharing. Because SparkFG is unable to amortize the overhead of starting Mesos Tasks across Spark tasks as is done for coarse grained frameworks, overall performance is significantly degraded. SparkCG outperforms SparkFG in all cases and Hadoop outperforms SparkFG in multiple cases. In particular, for the small 1GB input size on the development cluster, for which the increased latencies of Spark fine-grained correspond to a significant overhead on the total job runtimes, Spark performance is worse than Hadoop. This is also true for the Naive Bayes benchmark. In this case Spark first collects the classifier's model from HDFS in a separate stage with one running task on a sigle executor and it proceeds with staging the executors for the other tasks of the job only after completion of this stage. This delayed staging of executors on Mesos leads to slower runtimes for Spark Fine-Grained.

In the next experiment, we investigate the performance impact of multi-tenancy in a resource constrained setting. For this study, we execute the same application in Hadoop and SparkCG and start them together on Mesos. In this configuration, Hadoop and SparkCG share the available Mesos Slaves and access the same data sets stored on HDFS. Figure 3 shows the application execution time in seconds (using different input sizes) over Hadoop and SparkCG in this multi-tenant scenario. As in the previous set of results, SparkCG outperforms Hadoop for all benchmarks and input sizes.

Multi-tenant Performance

We observe in the logs from these experiments that SparkCG is able to setup its application faster than Hadoop is able to. As a result, SparkCG wins the race to acquire resources from Mesos first. To evaluate the impact of such sequencing, we next investigate what happens when Hadoop receives its offers from Mesos ahead of SparkCG. To enable this timing of offers, we delay the Spark job submission by 10



(a) Development Cloud

(b) Production Cloud

Figure 4: **Multi-tenant Performance (Spark receives offers first):** Benchmark execution time in seconds for Hadoop and SparkCG using different input sizes deployed on the development cloud (Figure 4a) and the production cloud (Figure 4b), when we delay Spark to ensure that Hadoop receives Mesos offers first.

seconds. We present these results in Figure 4. In this case, SparkCG outperforms Hadoop for only the 1GB input size.

To understand this effect better, we summarize (i.e. we zoom in) the performance differences between Hadoop and SparkCG for different Mesos offer orders. Figure 5 shows execution time for WordCount and the 15GB input size using the production cloud. The first pair of bars shows the total time for the benchmark when each framework has sole access to the entire cluster (for reference from Figure 2b). The second pair of bars is the total time when Hadoop receives its resource offers from Mesos first. The third pair shows total time when SparkCG receives Mesos offers first (for reference from Figure 3b).

The data shows in this case that even though Spark is more than 160 seconds faster than Hadoop in single-tenant mode, it is more than 85 seconds *slower* than Hadoop when the Hadoop job starts ahead of the Spark job. Whichever framework starts first, executes with time similar to that of the single tenancy deployment.

This behavior results from the way that Mesos allocates resources. Mesos offers *all* of the available resources to the first framework that registers with it, since it is unable to know whether or not there will be a future framework to register. Mesos is incapable to change system-wide allocation when a new framework arrives, since it does not implement resource revocation. SparkCG and Hadoop will block all other frameworks until they complete execution of a job. In Hadoop, such starvation can extend beyond a single job, since Hadoop jobs are submitted on the same Hadoop JobTracker instance. That is, a Hadoop instance will retain Mesos resources until its job queue (potentially holding multiple jobs) empties.

These experiments show that when an application requires resources that exceed those available in the cloud (input sizes 5GB and above in our experiments), and when frameworks use CG mode, Mesos fails to share cloud resources fairly among multiple tenants. In such cases, Mesos serializes application execution limiting both parallelism and utilization significantly. Moreover, application performance in such cases becomes dependent upon framework registration order and as a result is highly variable and unpredictable.

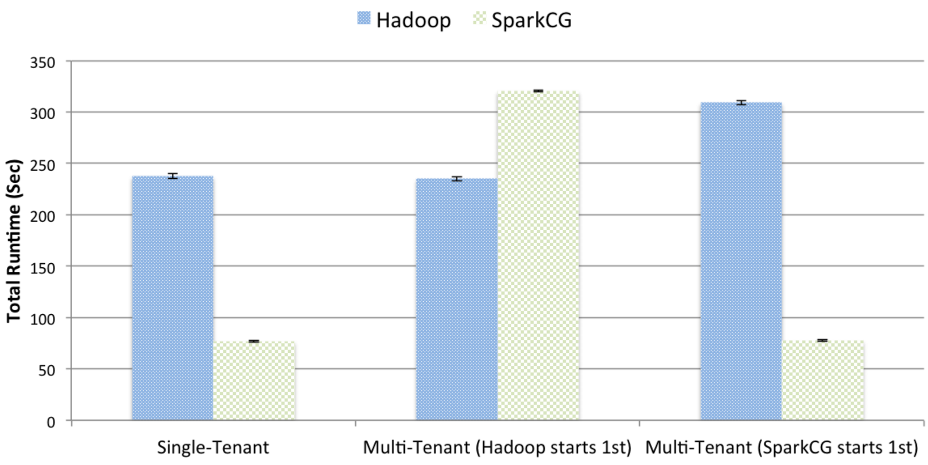


Figure 5: **Performance Implications of Multi-tenancy and Mesos Offer Order: Hadoop and SparkCG.** This graph shows WordCount execution time in seconds for input size 15GB using the production cloud (single-tenant, multi-tenant with Hadoop ahead of Spark, and multi-tenant with Spark ahead of Hadoop). The framework that receives Mesos offers first performs best.

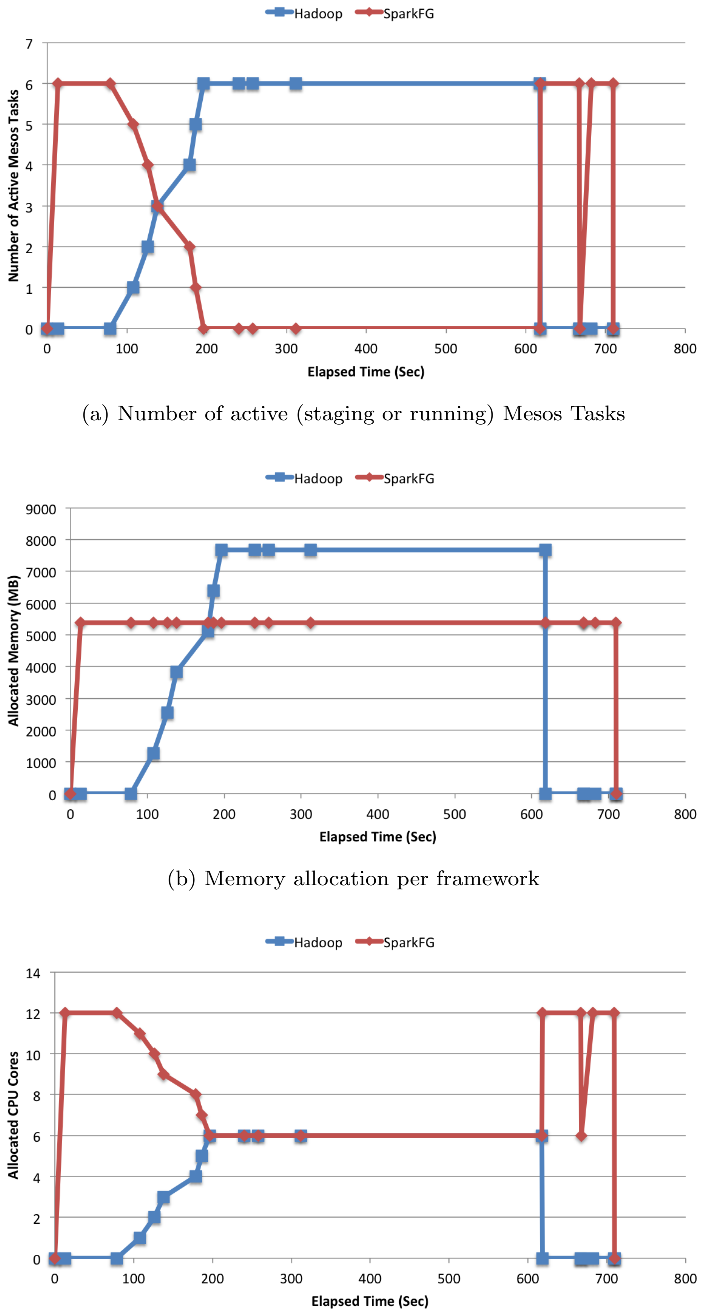
Fine-Grained Resource Sharing

In this section, we investigate the operation of the Mesos scheduler for frameworks that employ fine grained scheduling. For such frameworks (SparkFG in our study), the framework scheduler can release and acquire resources throughout the lifetime of an application.

For these experiments, we measure the impact of interference between Hadoop and SparkFG. As in the previous section, we consider the case when Hadoop starts first and when SparkFG starts first. We present a representative subset of the results for clarity and brevity. Figure 7 shows the total execution time in seconds for WordCount and its 15GB input on the production cloud when we run Hadoop and SparkFG together and alter the Mesos offer order. As for Figure 5, we present three pairs of bars. The first, for reference, is the single-tenant performance. The second is the performance when Hadoop receives offers from Mesos ahead of SparkFG. For the third, SparkFG receives Mesos offers ahead of Hadoop.

As we expect, when Hadoop receives offers from Mesos first, it acquires all of the available resources, blocks SparkFG from executing, and outperforms SparkFG. Similarly, when SparkFG receives its offers ahead of Hadoop, we expect it to block Hadoop. However, from the performance comparison, this starvation does not occur. That is, Hadoop outperforms SparkFG (the far right pair of bars) even when SparkFG starts first and can acquire all of the available resources.

We further investigate this behavior in Figure 6. In this set of graphs, we present a timeline of multi-tenant activities over the lifetime of two WordCount/5GB applications (one over Hadoop, the other over SparkFG). In the top graph, we present the number of Mesos Tasks allocated by each framework. Mesos Tasks encapsulate



(a) Number of active (staging or running) Mesos Tasks

(b) Memory allocation per framework

(c) CPU cores allocation per framework

Figure 6: **Multi-tenancy and Resource Utilization:** The timelines show active Mesos Tasks, memory, and CPU allocation in Mesos for the development cloud. Hadoop and Spark in FG mode compete for resources. Hadoop gradually takes over, running Tasks on its executors (Figure 6a), while memory (Figure 6b) and CPU cores (Figure 6c) previously assigned to Spark remain idle until Hadoop completes.

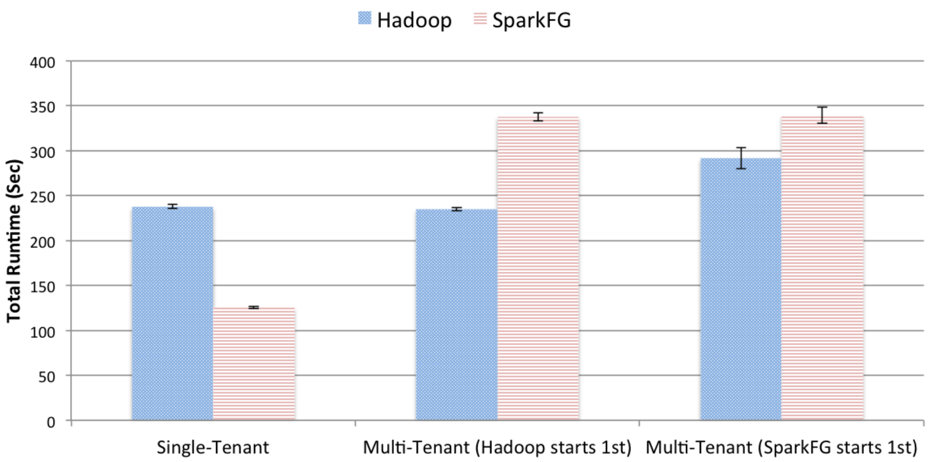


Figure 7: **Performance Implications of Multi-tenancy and Mesos Offer Order: Hadoop and SparkFG.** This graph shows WordCount execution time in seconds for input size 15GB using the production cloud (single-tenant, multi-tenant with Hadoop ahead of Spark, and multi-tenant with Spark ahead of Hadoop).

the execution of one (SparkFG) or many (Hadoop) framework tasks. The middle graph shows the memory consumption by each framework and the bottom graph shows the CPU resources consumed by each framework.

In this experiment, SparkFG receives first the offers from Mesos and acquires all the available resources of the cloud (all resources across the six Mesos Slaves are allocated to SparkFG). SparkFG uses these resources to execute the application and Hadoop is blocked waiting on SparkFG to finish. Because SparkFG employs a fine grained resource use policy, it releases the resources allocated to it for a framework task back to Mesos when each task completes. Doing so enables Mesos to employ its fair sharing resource allocation policy (DRF) and allocate these released resources to other frameworks (Hadoop in this case) – and the system achieves true multi-tenancy.

However, such sharing is short lived. As we can observe in the graphs, over time as SparkFG Mesos Tasks are released, they are allocated to Hadoop until only Hadoop is executing (SparkFG is eventually starved). The reason for this is that even though SparkFG releases its task resources back to Mesos, it does not release *all* of its resources back, in particular, it does not release the resources allocated to it for its Mesos executors (one per Mesos Slave).

In our configuration, SparkFG executors require 768MB of memory and 1CPU per Slave. Mesos DRF considers these resources part of the SparkFG dominant share and thus gives Hadoop preference until all resources in the system are once again consumed. This results in SparkFG holding onto memory and CPU (for its Mesos executors) that it is unable to use because there are insufficient resources for its tasks to execute but for which Mesos is charging under DRF. Thus, SparkFG induces a deadlock and all resources being held by SparkFG executors in the system are wasted (and system resources are underutilized until Hadoop completes and releases its resources).

In our experiments, we find that this scenario occurs for all but the shortest lived jobs (1GB input sizes). The 1GB jobs include only 8 tasks and so SparkFG will execute 6 out of its 8 task after getting all the resources on the first round of offers. Moreover, Hadoop doesn't require all the Slaves to run 8 tasks for this job as explained on Section leaving sufficient space to Spark to continue executing the remaining two tasks uninterrupted.

Deadlock in Mesos in resource constrained settings is not limited to the SparkFG scheduler. The fundamental reason behind this type of deadlock is a combination of (i) frameworks "hanging on" to resources and, (ii) the way Mesos accounts for resource use under its DRF policy. In particular, any framework scheduler that retains resources across tasks, e.g. to amortize the startup overhead of the support services (like Spark executors), will be charged for them by DRF, and thus may deadlock. Moreover, any Mesos system for which resource demand exceeds capacity can deadlock if there is at least one framework with a fine grained scheduler and at least one framework with a coarse grained scheduler.

Batch and Streaming Tenant Interference

We next evaluate the impact of performance interference in Mesos under resource constraints, for batch and streaming analytics frameworks. This combination of frameworks is increasingly common given the popularity of the *lambda architecture* [34] in which organizations combine batch processing to compute views from a constantly growing dataset and stream processing to compensate for the high latency between subsequent iterations of batch jobs and to complement the batch results with newly arrived unprocessed data [35–37].

We perform two types of experiments. In the first, we execute a streaming application using a Storm topology continuously, while we introduce batch applications. In the second, we submit batch and streaming jobs simultaneously to Mesos. Figure 8 illustrates the performance results for the former. We present execution time in seconds for the applications and input sizes using Hadoop, SparkFG, and SparkCG, when Storm executes in the background. The results show that the performance degradation introduced by the Storm tenant varies between 25% to 80% across frameworks and inputs, and is insignificant for the 1GB input.

The reason for this variation is that Storm accepts offers from Mesos for three Mesos Slaves to run its job on the development cloud. This leaves three Slaves for Hadoop, SparkFG, and SparkCG to share. The degradation is limited because fewer Slaves impose less startup overhead on the framework executors per Slave. The overhead of staging new Mesos Tasks and spawning executors is so significant that it is not amortized by the additional parallelization that results from additional Mesos Slaves. We omit results for the production cloud for brevity. The results are similar but show less degradation (insignificant for 1GB, and 5% to 35% across frameworks and other inputs) due to the additional resources available in the production cloud.

Figure 9 shows the impact of interference from batch systems on Storm throughput in tuples per second (Results for latency are similar and we omit them for brevity). We find that the interference is insignificant and Storm performance is the same

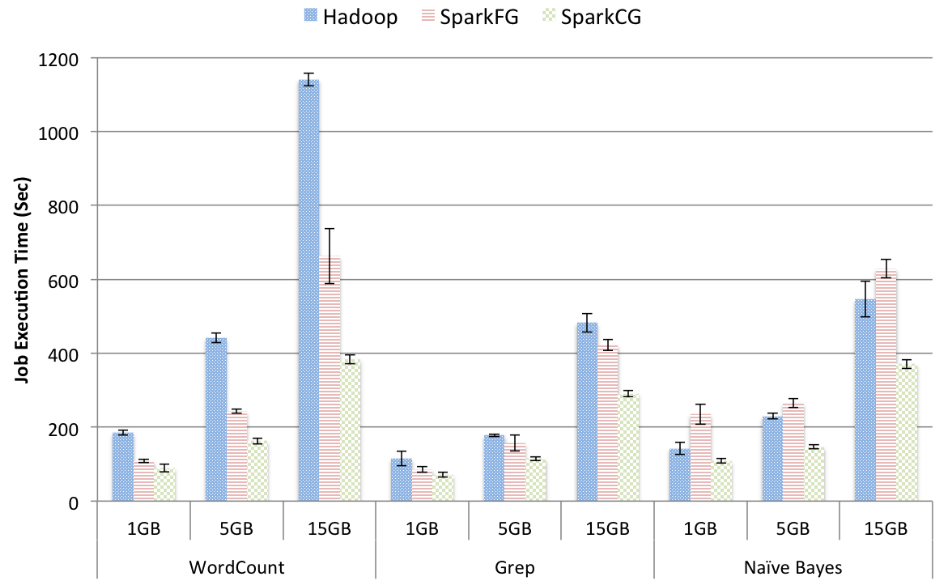


Figure 8: **Hadoop and Spark performance on Mesos-managed cloud shared with Storm:** Execution time in seconds for Hadoop and Spark applications in CG and FG mode, for different input sizes and benchmarks on the development cloud.

as that when it executes in single-tenant mode, since Storm receives its offers and allocates the resources it needs ahead of the batch frameworks. When a coarse-grained batch system receives its resource offers from Mesos ahead of Storm, Storm execution is blocked until the batch system finishes. Figure 10 shows the timing diagram and this effect on Storm when executed with a SparkCG tenant in Mesos for the development cloud. Each line type shows a different stage of execution for each framework for each Mesos Task. Results with Hadoop and SparkFG are not shown for brevity. Hadoop has the exact same effect on Storm as SparkCG, while with SparkFG this depends on the cloud size. On the development cloud the released resources from SparkFG are not sufficient for Storm to deploy its executors and therefore Storm is blocked, while in the production cloud Storm will acquire some of the resources released by SparkFG as described previously, without however deadlocking SparkFG because Storm does not consume all of the cloud resources to run its tasks.

Startup Overhead of Mesos Tasks

We next investigate Mesos Task startup overhead for the batch frameworks under study. We define the startup delay of a Mesos Task as the elapsed time between when a framework issues the command to start running an application and when the Mesos Task appears as running in the Mesos user interface. As part of startup, the frameworks interact with Mesos via messaging and access HDFS to retrieve their executor code. This time includes that for setting up a Hadoop or Spark job, for launching the Mesos executor (and respective framework implementation, e.g. TaskTracker, Executor), and launching the first framework task.

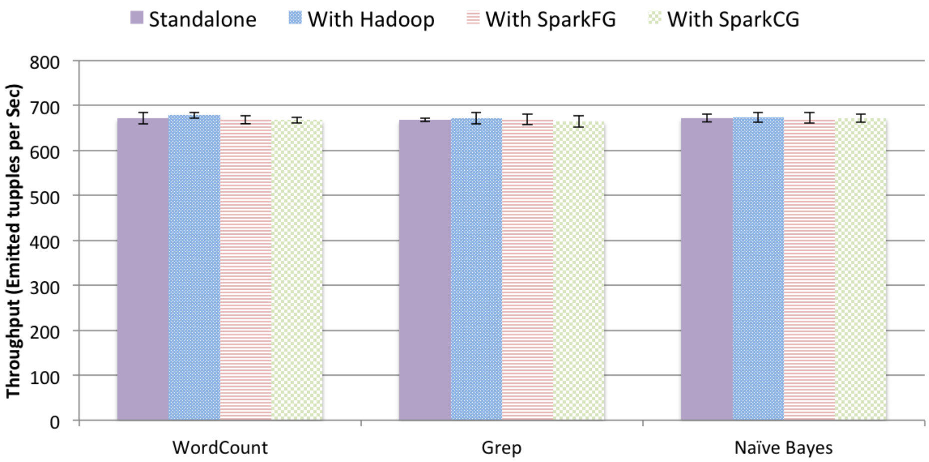


Figure 9: **Impact of Multi-tenancy on Storm in Mesos:** When Storm receives its offers from Mesos ahead of the batch frameworks, there is no interference impact on Storm, i.e. Storm throughput is the same in multi-tenant and single tenant deployments.

Figure 11a shows the average startup time in seconds for each Mesos Slave across applications for Hadoop, SparkFG, and SparkFG when running the WordCount application with input size 15GB. Our experiments indicate that other applications perform similarly. The data shows that as new tasks are launched (each on a new Mesos Slave), the startup delay increases and each successive Slave takes longer to complete the startup process. Our measurements show that this increase is due to network and disk contention. Slaves that start earlier complete the startup process earlier and initiate application execution (execution of tasks). Task execution consumes significant network and disk resources (for HDFS access) which slows down the startup process of *later* Slaves. This interference grows with the size of application input as shown in Figure 11b. The graph shows the Mesos Task startup overhead in seconds for each Slave for different input sizes for WordCount over Hadoop (other frameworks exhibit similar behavior).

Our results show that startup overhead impacts the overall performance of applications and can significantly degrade the performance of short running jobs: 30% for Hadoop and 55% for SparkFG for the 1GB experiments. Given that short running jobs account for an increasingly large portion of big data workloads today [13–15], such overheads can cause significant under-utilization and widely varying application performance in constrained settings.

Discussion

In this study, we use three different applications to expose the challenges to fair sharing in multi-tenant, resource constrained cluster settings. Regardless of application, the root cause of framework interference is how resources are allocated and shared. The applications we include are those used in previous performance

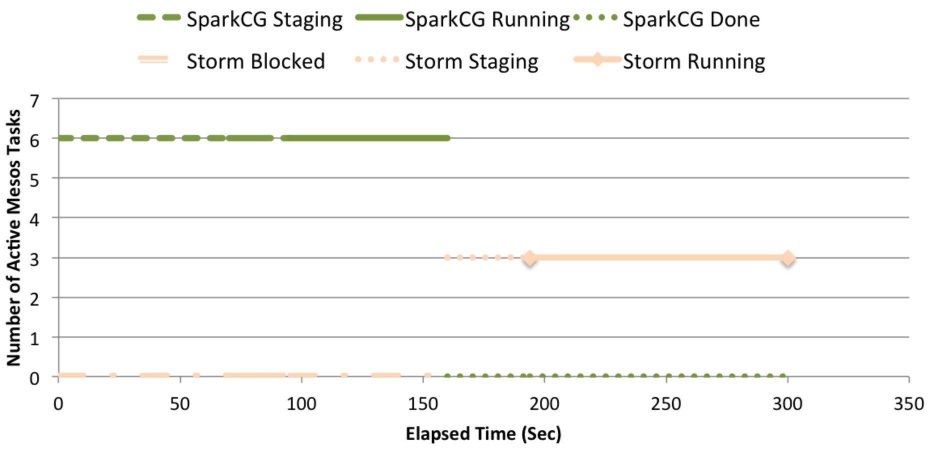


Figure 10: **Multi-tenant Interference between Storm and SparkCG in Mesos, broken down by stage of execution:** When SparkCG receives its Mesos offers first, Storm blocks until SparkCG finishes.

studies [6, 23, 24]. WordCount and Grep are core components in text processing applications, and Naive Bayes Classification is a popular algorithm used in social network and e-commerce analytics.

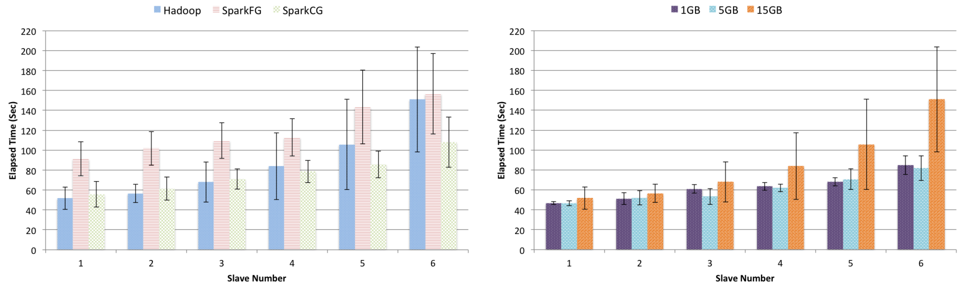
Number of tenants

To study multi-tenancy, we consider two and three tenant scenarios. Our experience has been that these scenarios capture much of the interference behavior representative of higher numbers of tenants. The reason for this is fundamental to Mesos resource allocation. The framework that submits a job first, receives Mesos offers and therefore acquires all available resources. If the framework is coarse-grained it will keep these resources and block the other frameworks until it completes its job. If the first job that arrives is from a fine-grained framework, then the fine-grained framework will release the resources related to its tasks while keeping the resources related to its executors regardless of the number of tenants in the system.

Key Insights

Performance interference occurs when the available cluster resources is less than the peak demand of the jobs running on the cluster which can occur on clusters of any size, but becomes the common case when resources are constrained. Static allocation of resources in constrained settings only exacerbates the problem. To reduce the impact of framework interference, Mesos must perform dynamic resource allocation and be extended to provide either intelligent admission control, a resource revocation capability, or both, to avoid the performance degradations revealed in this paper.

In particular, we can extend the resource manager to identify problematic behavior and revoke resources when fair sharing is violated. This requires the implementation of a pre-emption mechanism in each framework in order to avoid costly



(a) Task startup delays across frameworks

(b) Task startup delays across input sizes

Figure 11: **Task Startup Delays Across Frameworks (Figure 11a) and Input Sizes (Figure 11b) for the Development Cloud:** The data shows the time in seconds required for each Mesos Task to finish its startup process.

re-computations. Such a solution, however, will degrade performance of the framework from which the resources are revoked, but would enable a more predictable behavior of all the frameworks and guarantee continue progress of all the jobs.

Another way to address these issues is through job admission control. The resource manager can be extended to record historic information about job behavior or to support deadlines [38–40]. Mesos could then offer frameworks only the necessary resources required to meet a deadline to give the system more flexibility in achieving fair sharing. Deadline-driven admission control does not preclude all deadlock, but it can reduce its frequency.

We can also overcome framework interference in multi-tenant scenarios by forcing all frameworks to use fine-grained allocation in which they release all the resources related to their tasks and executors every time a task completes its execution. This way, even if the resource manager offers all the available resources to the framework that submits its job first, the framework will release resources upon task completion and these resources will be offered to the frameworks waiting. Such a requirement comes with a performance penalty as the overheads of creating executors and new Mesos tasks are significant but has the potential to increase performance stability and fair sharing in multi-tenant scenarios.

Related Work

This paper is an extended version of a conference publication [41]. Our extensions include results from experiments using a second private cloud system (called production) and a study of the impact of multi-tenancy when different resource-offer orders are considered. We also provide additional analysis on fine-grained resource sharing and on the impact of multi-tenancy consisting of combinations of big data and fast data (streaming) frameworks.

Cluster managers like Mesos [9] and YARN [42] enable the sharing of cloud and cluster resources by multiple, data processing frameworks. YARN uses a classic resource request model in which each framework asks for the resources it needs to run its jobs. Mesos as described herein, implements an offer-based model in which frameworks can accept or reject offers for resources based on whether the offers satisfy the resource requirements of the application. Our work focuses on fair-sharing and deadlock issues that occur on Mesos due to lack of admission control and resource revocation. However, Mesos is not the only cluster manager that suffers from such problems. Other work [43] shows that, when the amount of required resources exceeds that which is available, deadlocks also occur on YARN.

Recently, new big data workflow managers that support multiple execution engines have emerged. Musketeer [6] dynamically maps a workflow description to a variety of execution engines, including Hadoop and Spark to select the best performing engine for the particular workflow. Similarly, [7] optimizes end-to-end data flows, by specializing and partitioning the original flow graph into sub flows that are executed over different engines. The advent of these higher-level managers calls for an increase in the combined use of data processing systems in the near future. Our work focuses on understanding system design limitations that will emerge under these new conditions.

The performance differences of MapReduce and Spark on very large clusters, with an emphasis on the architectural components that contribute to these differences is studied in [23]. [44] evaluates the memory and time performance of Spark and MapReduce on Mesos, for the PageRank algorithm. The authors in [24] extend MPI to support Big Data jobs and compare performance and resource utilization of Hadoop and Spark. Ousterhout et al [45] suggest using blocked-time analysis to quantify performance bottlenecks in big data frameworks and apply it to analyze Spark's performance. In a recent work, Li et al [46] extend incremental Hadoop [47] to support low latency stream queries and compare the performance of their system to Spark streaming [48] and Storm. The key aspect that differentiates our work is our investigation and characterization of the performance of Hadoop, Spark and Storm applications, when run over Mesos cluster manager, in resource constrained and multi-tenant settings.

There are numerous studies that characterize the performance of MapReduce workloads. Many show that these workloads consist of many jobs (if not the majority) that have small input sizes and that have short execution times. Chen et al [13] observe that most jobs have input, shuffle and, output sizes in the MB to GB range and that 90% of the jobs have input datasets less than few GBs. Similarly, authors of [14] find that over 40% of jobs have less than 10 tasks, while about 50% of jobs have between 10 to 2000 tasks and, 80% of the jobs have duration less than 2 minutes. Lastly, the authors in [15] observe that small jobs dominate their workloads and that more than 90% of jobs touch less than 20GB of data, and their duration is less than 8 minutes.

Other researchers (e.g. [49,50]) have shown the significant impact of startup overhead on MapReduce jobs that run on large cluster systems (hundreds to thousands of nodes). Our work differs from this past work in that we investigate the performance impact of multi-tenant interference on short running applications and analyze the overhead of job startup under moderate resource constraints. Such scenarios are increasingly common yet are not those for which large scale analytics systems were originally designed, warranting further study.

Conclusions

The goal of our work is to characterize the behavior of "big data" analytics frameworks in shared settings for which computing resources (CPU and memory) are limited. Such settings are increasingly common in both public and private cloud systems in which cost and physical limitations constrain the number and size of resources that are made available to applications. In this paper, we investigate the performance and behavior of distributed batch and stream processing systems that share resource constrained, private clouds managed by Mesos. We examine how these systems interfere with each other and Mesos, to evaluate the effect on systems performance, overhead, and fair resource sharing.

We find that in such settings, the absence of an effective resource revocation mechanism supported by Mesos and the corresponding data processing systems running on top of it, leads to violation of fair sharing. In addition, the naive allocation mechanism of Mesos benefits significantly the framework that submits its application first. As a result coarse-grained framework schedulers cause resource starvation for later tenants. Moreover, when systems (either batch or streaming) with different scheduling granularities (fine-grained or coarse-grained) co-exist on the same Mesos-managed cloud, resource underutilization and resource deadlocks can occur. Finally, the overhead introduced during application startup on Mesos affects all frameworks and significantly degrades the performance of short running applications.

Competing interests

The authors declare that there is no conflict of interest regarding the publication of this paper.

Acknowledgements

This work is funded in part by NSF (CCF-1539586, CNS-1218808, CNS-0905237, ACI-0751315), NIH (1R01EB014877-01), ONR NEEC (N00174-16-C-0020), Huawei Technologies, and the California Energy Commission (PON-14-304).

References

1. Apache Hadoop. https://hadoop.apache.org/. [Online; accessed 21-January-2016]
2. Apache Spark. http://spark.apache.org/. [Online; accessed 21-January-2016]
3. Apache Storm. http://storm.apache.org/. [Online; accessed 21-January-2016]
4. Apache Mahout. http://mahout.apache.org/. [Online; accessed 25-November-2015]
5. Spark MLlib. http://spark.apache.org/mllib/. [Online; accessed 29-January-2016]
6. Gog, I., Schwarzkopf, M., Crooks, N., Grosvenor, M.P., Clement, A., Hand, S.: Musketeer: all for one, one for all in data processing systems. In: Proceedings of the Tenth European Conference on Computer Systems, p. 2 (2015). ACM
7. Simitsis, A., Wilkinson, K., Castellanos, M., Dayal, U.: Optimizing analytic data flows for multiple execution engines. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 829–840 (2012). ACM
8. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., *et al.*: Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing, p. 5 (2013). ACM
9. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R.H., Shenker, S., Stoica, I.: Mesos: A platform for fine-grained resource sharing in the data center. In: NSDI, vol. 11, pp. 22–22 (2011)
10. Sefraoui, O., Aissaoui, M., Eleuldj, M.: Openstack: toward an open-source solution for cloud computing. International Journal of Computer Applications **55**(3) (2012)
11. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The eucalyptus open-source cloud-computing system. In: Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium On, pp. 124–131 (2009). IEEE
12. Kumar, R., Jain, K., Maharwal, H., Jain, N., Dadhich, A.: Apache cloudstack: Open source infrastructure as a service cloud computing platform. Proceedings of the International Journal of advancement in Engineering technology, Management and Applied Science, 111–116 (2014)
13. Chen, Y., Alspaugh, S., Katz, R.: Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. Proceedings of the VLDB Endowment **5**(12), 1802–1813 (2012)
14. Ren, Z., Xu, X., Wan, J., Shi, W., Zhou, M.: Workload characterization on a production hadoop cluster: A case study on taobao. In: Workload Characterization (IISWC), 2012 IEEE International Symposium On, pp. 3–13 (2012). IEEE

15. Ren, K., Gibson, G., Kwon, Y., Balazinska, M., Howe, B.: Hadoop's adolescence; a comparative workloads analysis from three research clusters. In: SC Companion, p. 1452 (2012)
16. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: Fair allocation of multiple resource types. In: NSDI, vol. 11, pp. 24–24 (2011)
17. Apache Mesos. http://mesos.apache.org/. [Online; accessed 21-January-2016]
18. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, pp. 2–2 (2012). USENIX Association
19. HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. [Online; accessed 21-January-2016]
20. Hadoop Users. http://wiki.apache.org/hadoop/PoweredBy. [Online; accessed 27-January-2016]
21. Spark Users. https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark. [Online; accessed 27-January-2016]
22. Storm Users. http://storm.apache.org/documentation/Powered-By.html. [Online; accessed 27-January-2016]
23. Shi, J., Qiu, Y., Minhas, U.F., Jiao, L., Wang, C., Reinwald, B., Özcan, F.: Clash of the titans: Mapreduce vs. spark for large scale data analytics. Proceedings of the VLDB Endowment **8**(13), 2110–2121 (2015)
24. Liang, F., Feng, C., Lu, X., Xu, Z.: Performance benefits of datampi: a case study with bigdatabench. In: Big Data Benchmarks, Performance Optimization, and Emerging Hardware, pp. 111–123. Springer, ??? (2014)
25. Mesos Roles. http://mesos.apache.org/documentation/latest/roles/. [Online; accessed 28-December-2016]
26. Apache Mesos 0.21.1 Release. http://mesos.apache.org/blog/mesos-0-21-1-released/. [Online; accessed 1-December-2015]
27. CDH 5.1.2 Release. http://www.cloudera.com/content/www/en-us/downloads/cdh/5-1-2.html. [Online; accessed 1-December-2015]
28. Apache Spark 1.2.1 Release. https://spark.apache.org/releases/spark-release-1-2-1.html. [Online; accessed 1-December-2015]
29. Apache Storm 0.9.2 Release. http://storm.apache.org/2014/06/25/storm092-released.html. [Online; accessed 1-December-2015]
30. Bug Fixes for Hadoop On Mesos. https://github.com/strat0sphere/hadoop. [Online; accessed 1-December-2015]
31. Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S., *et al.*: Bigdatabench: A big data benchmark suite from internet services. In: High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium On, pp. 488–499 (2014). IEEE
32. Modifications on Benchmarking Code. https://github.com/MAYHEM-Lab/benchmarking-code. [Online; accessed 22-January-2015]
33. Massie, M.L., Chun, B.N., Culler, D.E.: The ganglia distributed monitoring system: design, implementation, and experience. Parallel Computing **30**(7), 817–840 (2004)
34. Lambda Architecture. http://lambda-architecture.net/. [Online; accessed 1-December-2015]
35. Twitter Summingbird. https://github.com/twitter/summingbird. [Online; accessed 27-January-2016]
36. Lambdoop. https://novelti.io/lambdoop/. [Online; accessed 27-January-2016]
37. Lambda on Metamarkets. https://metamarkets.com/2014/building-a-data-pipeline-that-handles-billions-of/-events-in-real-time/. [Online; accessed 27-January-2016]
38. Verma, A., Cherkasova, L., Kumar, V.S., Campbell, R.H.: Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle. In: 2012 IEEE Network Operations and Management Symposium, pp. 900–905 (2012). IEEE
39. Delimitrou, C., Kozyrakis, C.: Quasar: resource-efficient and qos-aware cluster management. ACM SIGPLAN Notices **49**(4), 127–144 (2014)
40. Ferguson, A.D., Bodik, P., Kandula, S., Boutin, E., Fonseca, R.: Jockey: guaranteed job latency in data parallel clusters. In: Proceedings of the 7th ACM European Conference on Computer Systems, pp. 99–112 (2012). ACM
41. Dimopoulos, S., Krintz, C., Wolski, R.: Big data framework interference in restricted private cloud settings. In: IEEE International Conference on Big Data (2016). IEEE
42. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., *et al.*: Storm@ twitter. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 147–156 (2014). ACM
43. Yao, Y., Lin, J., Wang, J., Mi, N., Sheng, B.: Admission control in yarn clusters based on dynamic resource reservation. In: 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), pp. 838–841 (2015). IEEE
44. Gu, L., Li, H.: Memory or time: Performance evaluation for iterative operation on hadoop and spark. In: High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference On, pp. 721–727 (2013). IEEE
45. Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., Chun, B.-G., ICSI, V.: Making sense of performance in data analytics frameworks. In: Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)(Oakland, CA, pp. 293–307 (2015)
46. Li, B., Diao, Y., Shenoy, P.: Supporting scalable analytics with latency constraints. Proceedings of the VLDB Endowment **8**(11), 1166–1177 (2015)
47. Li, B., Mazur, E., Diao, Y., McGregor, A., Shenoy, P.: A platform for scalable one-pass analytics using mapreduce. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp. 985–996 (2011). ACM

48. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 423–438 (2013). ACM
49. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)
50. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 165–178 (2009). ACM