Big Data Framework Interference In Restricted Private Cloud Settings

Stratos Dimopoulos, Chandra Krintz, Rich Wolski
Dept. of Computer Science, Univ. of California, Santa Barbara
{stratos, ckrintz, rich}@cs.ucsb.edu - Contact: stratos@cs.ucsb.edu

***Abstract*—In this paper, we characterize the behavior of "big" and "fast" data analysis frameworks, in multi-tenant, shared settings for which computing resources (CPU and memory) are limited, an increasingly common scenario used to increase utilization and lower cost. We study how popular analytics frameworks behave and interfere with each other under such constraints. We empirically evaluate Hadoop, Spark, and Storm multi-tenant workloads managed by Mesos. Our results show that in constrained environments, there is significant performance interference that manifests in failed fair sharing, performance variability, and deadlock of resources.**

Keywords*—*Big Data Infrastructures and Frameworks, Private Cloud Multi-tenancy, Performance Interference

I. Introduction

Recent technological advances have spurred production and collection of vast amounts of data about individuals, systems, and the environment. 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 service 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, Spark, and Storm 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 [3, 13]. 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 [16, 4] 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. 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 [1, 11, 10].

We empirically evaluate the use of Hadoop, Spark, and Storm frameworks in combination, with Mesos [4] 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 under resource pressure, and how Mesos behaves and achieves fair sharing [2] when when demand for resources exceeds resource availability.

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.

II. 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 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 [4]. 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.

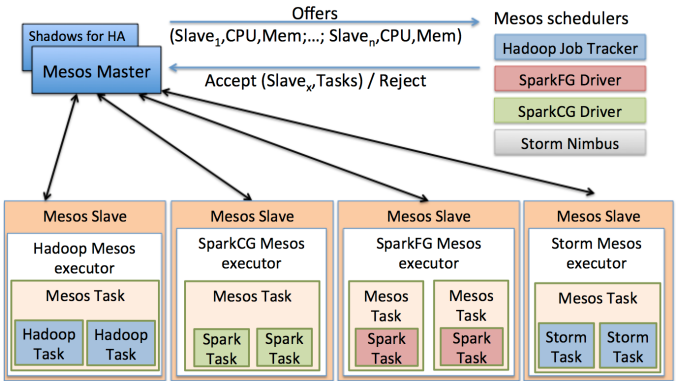


Fig. 1: Mesos Architecture.

Mesos provides two-level, offer-based, resource scheduling for frameworks as depicted in Figure 1. 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) [2]. 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) [2]. We employ the default DRF scheduler in Mesos for this study. We use Mesos to manage the Hadoop, Spark, and Storm data analytics frameworks from the Apache Foundation in a multitenent setting.

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. Spark supports both FG and CG modes; Hadoop and Storm implement the CG mode only.

III. 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.

Our private cloud is a resource-constrained, Eucalyptus [9] v3.4.1 private cluster with nine virtual servers (nodes) and a Gigabit Ethernet switch. 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 each have 2x2.5GHz CPUs, 4GB of RAM, and 60GB disk space.

Our nodes run Ubuntu v12.04 Linux with Java 1.7, Mesos 0.21.1 which uses Linux containers by default for isolation, the CDH 5.1.2 MRv1 Hadoop stack (HDFS, Zookeeper, MapReduce, etc.), Spark v1.2.1, and Storm v0.9.2. 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.

Our batch processing workloads and data sets come from the BigDataBench and the Mahout projects [17]. 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 [8]. 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

	Available		Min Required		
	Slave	Total	Hadoop	Spark	Storm
CPU	2	12	1	2	2
Mem (MB)	2931	17586	980	896	2000

	Max Used					
	per Slave			Total		
	Hadoop	Spark	Storm	Hadoop	Spark	Storm
CPU	2	2	2	12	12	6
Mem (MB)	2816	896	2000	16896	5376	6000

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

Ganglia [7], ifstat, iostat, and vmstat available in Linux, and log files available from the individual frameworks.

Table I shows the available resources in our private cloud deployment, 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. 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 tasks per Mesos executor. Hadoop spawns one Mesos executor per Mesos Slave and Hadoop tasks can be employed as either mapper or reducer slots. Spark in FG mode runs 1 Mesos/Spark task per executor. 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. We run one worker per Supervisor (Mesos executor) so three Slaves are needed in total. 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.

IV. Results

We use this experimental setup to first measure the performance of Hadoop and Spark when they run in isolation (single tenancy) on our Mesos-managed private cloud. 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). 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.

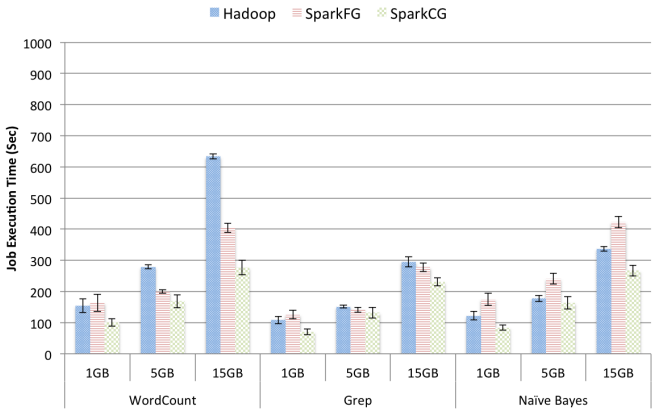


Fig. 2: Single Tenant Performance: Benchmark execution time in seconds for Hadoop and Spark on Mesos for different input sizes.

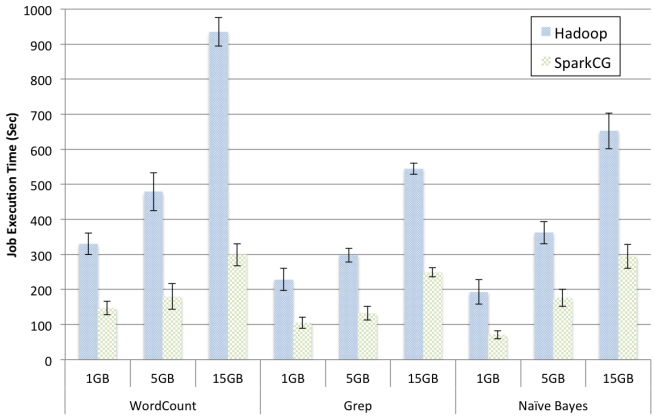


Fig. 3: Multi-tenant Performance: Benchmark execution time in seconds for Hadoop and SparkCG using different input sizes, with SparkCG receiving its offers first.

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) [12, 6]. One interesting aspect of this data is the performance difference between SparkCG and SparkFG. SparkCG outperforms SparkFG in all cases and more than 1.5x 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.

A. Multi-tenant Performance

We next evaluate 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.

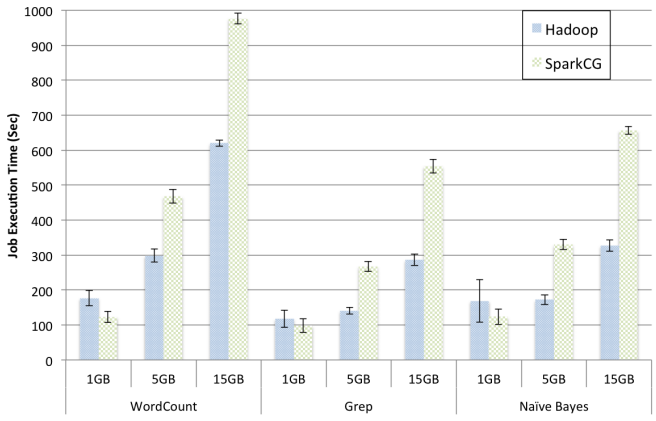


Fig. 4: Multi-tenant Performance: Benchmark execution time in seconds for Hadoop and SparkCG using different input sizes, with Hadoop receiving its offers first.

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 control the timing of offers in this way, we delay the Spark job submission by 10 seconds. We present these results in Figure 4. In this case, SparkCG outperforms Hadoop for only the 1GB input size.

The data shows in this case that even though Spark is more than 355 seconds faster than Hadoop in single-tenant mode, it is more than 600 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 unable to change its system-wide allocation decisions 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.

B. Fine-Grained Resource Sharing

We next 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 find (as expected) that 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 even when SparkFG starts first and can acquire all of the available resources. (We omit the figure with execution times for each framework due to space limitations from these text and we provide it on [14])

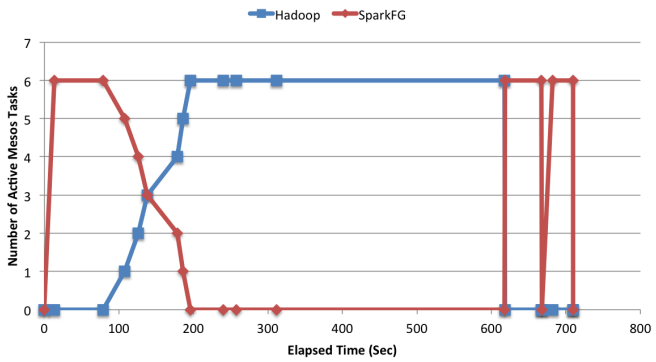
We further investigate this behavior in Figure 5. In this set of graphs, we present a timeline of multi-tenant activities over the lifetime of two WordCount/5GB applications (one over Hadoop, the other over SparkFG). In the top graph, we present the number of Mesos Tasks allocated by each framework.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

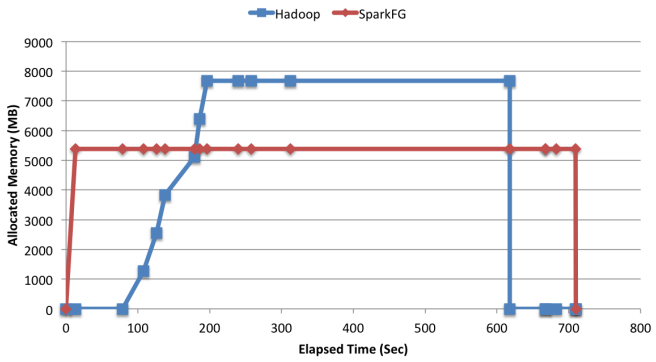
However, such sharing is short lived. As we can observe in the graphs, over time as SparkFG Mesos Tasks are released, they are allocated to Hadoop until only Hadoop is executing (SparkFG is eventually starved). The reason for this is that even though SparkFG releases its task resources back to Mesos, 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 (system resources are underutilized until Hadoop completes and releases its resources).

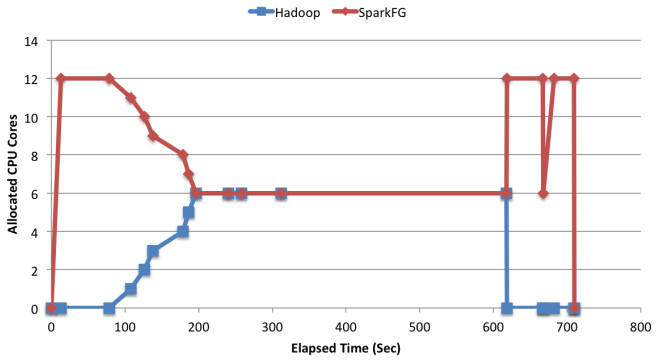
In our experiments, we find that this scenario occurs for all but the shortest lived jobs (1GB input sizes). The 1GB jobs include only 8 tasks and so SparkFG will execute 6 out of its 8 task after getting all the resources on the first round of offers. Moreover, Hadoop does not require all the Slaves to



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



(b) Memory allocation per framework



(c) CPU cores allocation per framework

Fig. 5: Multi-tenancy and Resource Utilization: The timelines show active Mesos Tasks, memory, and CPU allocation in Mesos.

run 8 tasks for this job as explained on Section III 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.

C. 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* [5] 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.

For this experiment, we execute a streaming application using a Storm topology continuously, while we introduce batch applications. Our measurements show (Detailed performance graphs can be found on [14]) that across frameworks and input sizes, 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. 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.

When we submit batch and streaming jobs simultaneously to Mesos, we find no perceivable interference from batch systems on Storm throughput and latency when storm receives its offers ahead of those for the batch frameworks. When Storm receives its offers after a batch framework, it blocks in the same way that fine grain frameworks do (as we describe the previous section) and fair sharing is violated.

D. 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.

We find 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 as a result of 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. We also find that this interference grows with the size of application input. (We provide the performance graph on [14]) Our measurements indicate that this overhead contributes to a significant degradation in the

performance of short running jobs. We observe a slow down of 30% for Hadoop and 55% for SparkFG for 1GB inputs across benchmarks. Given that short running jobs account for an increasingly large portion of big data workloads today [1, 11, 10], such overheads can cause significant under-utilization and widely varying application performance in constrained settings.

V. Related Work

Cluster managers like Mesos [4] and YARN [15] 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 [18] shows that, when the amount of required resources exceeds that which is available, deadlocks also occur on YARN. Our work differs from the numerous papers that characterize the performance of Hadoop and Spark applications on large-scale resources. To our knowledge, ours is the first paper to expose the performance implications of multi-tenant interference in resource constrained settings and for a variety of job types and frameworks.

Recently, new big data workflow managers that support multiple execution engines have emerged. Musketeer [3] 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, [13] 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.

VI. Conclusions and Future Work

In this paper, we 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 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.

This work is supported in part by NSF (CCF-1539586, CNS-0905237), NIH (1R01EB014877-01), the Naval Engineering Education Consortium (NEEC-n00174-16-C-0020), the California Energy Commission (PON-14-304), and Sedgwick Reserve.

References

[1] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 1802–1813.

[2] A. Ghodsi et al. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In: *NSDI*. 2011.

[3] I. Gog et al. Musketeer: all for one, one for all in data processing systems. In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 2.

[4] B. Hindman et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In: *NSDI*. Vol. 11. 2011, pp. 22–22.

[5] *Lambda Architecture*. http://lambda-architecture.net/.

[6] F. Liang et al. "Performance benefits of DataMPI: a case study with BigDataBench". In: *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer, 2014.

[7] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. In: *Parallel Computing* 30.7 (2004), pp. 817–840.

[8] *Modifications of Benchmarking Code*. https://github.com/MAYHEM-Lab/benchmarking-code.

[9] D. Nurmi et al. The eucalyptus open-source cloud-computing system. In: *CCGRID*. IEEE. 2009.

[10] K. Ren et al. Hadoop's Adolescence; A Comparative Workloads Analysis from Three Research Clusters. In: *SC Companion*. 2012, p. 1452.

[11] Z. Ren et al. Workload characterization on a production Hadoop cluster: A case study on Taobao. In: *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE. 2012, pp. 3–13.

[12] J. Shi et al. Clash of the titans: MapReduce vs. Spark for large scale data analytics. In: *Proceedings of the VLDB Endowment* 8.13 (2015), pp. 2110–2121.

[13] A. Simitsis et al. Optimizing analytic data flows for multiple execution engines. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 829–840.

[14] *Performance Interference of Multi-tenant, Big Data Frameworks in Resource Constrained Private Clouds*. http://www.cs.ucsb.edu/research/tech-reports/2016-08.

[15] A. Toshniwal et al. Storm@ twitter. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 147–156.

[16] V. K. Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 5.

[17] L. Wang et al. Bigdatabench: A big data benchmark suite from internet services. In: *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE. 2014, pp. 488–499.

[18] Y. Yao et al. Admission control in YARN clusters based on dynamic resource reservation. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE. 2015, pp. 838–841.