PYTHIA: Admission Control for Multi-Framework, Deadline-Driven, Big Data Workloads

Stratos Dimopoulos, Chandra Krintz, Rich Wolski
Department of Computer Science
University of California, Santa Barbara
{stratos, ckrintz, rich}@cs.ucsb.edu

UCSB Technical Report #2017-02 (Jan. 12, 2017)

***Abstract*—In this paper, we present PYTHIA, deadline-aware admission control for systems that execute jobs from multiple big data (batch) frameworks using shared resources. PYTHIA adds support for deadline-driven workloads in resource-constrained cloud settings, for use by resource negotiators such as Apache Mesos or YARN. PYTHIA uses histories of job statistics to estimate the minimum number of CPUs to allocate to a job in order for it to meet its deadline. PYTHIA admits jobs when these resources are available. Any job not admitted "fails fast" and wastes no resources. We implement a PYTHIA prototype and empirically evaluate it using production YARN traces under different resource constraints and deadline assignments. Our results show that PYTHIA is able to meet significantly more deadlines than fair share approaches and wastes fewer cloud resources in resource-limited scenarios, for the workloads, cluster sizes, and deadline assignments that we consider.**

I. Introduction

Increasingly, developers employ multiple "big data" and "fast data" frameworks to drive diverse data analysis requirements including rich query support, data mining, machine learning, real-time stream analysis, statistical analysis, and image processing. These frameworks such as Hadoop [1], Spark [2], Storm [3], and others, range in terms of analytics capability, ease of configuration and management, available tooling and programming support, efficiency, and scale. Because no single framework works best for all applications and settings, users commonly deploy multiple frameworks using the same cloud resources. Given the popularity of such *multi-analytics* clouds, resource negotiators such as Mesos [19] and YARN [40] have emerged to simplify and automate resource sharing and management.

Given the popularity of multi-analytics deployments, they are frequently used in shared settings (e.g. private clouds) 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. Resource-limited deployments pose significant challenges for resource negotiators and can result in low utilization, poor performance, unfair sharing, deadlock, and hard-to-predict workload behavior because the big data frameworks which underly them have been designed for resource-rich scenarios, and because resource negotiators are framework-agnostic [10, 47].

In this paper, we investigate a simple, new admission control strategy for resource negotiators called PYTHIA – for deadline-driven workloads in resource-constrained settings. Such workloads represent an important class of big data applications [24, 26, 43], which are unfortunately not fully supported in multi-analytic settings. Recent advances optimize based on framework-specific characteristics [21, 33, 42], depend upon job repetition [6, 13, 36] (which is not always available), use additional cloud resources to build offline performance and scalability models [8, 13, 41] or require complicated system extensions such as job pre-emption [17, 31]. Approaches that are framework specific (e.g. for Hadoop [18, 20, 21, 23, 25, 42, 48, 49] or Spark [33, 44] in isolation), cannot be used directly by resource negotiators. A lack of deadline support can lead to deadline misses when resources are constrained. Deadline misses disrupt big data pipeline processing, lead to inefficient resource utilization, increase the cost of cloud use, and lead to poor decision making (e.g. when based on stale information or analysis).

To address these limitations, we design PYTHIA to be framework-agnostic and to not rely on job repetition or pre-emption support. Instead, PYTHIA uses histories of job statistics (resource negotiator log information) to control job admission. Instead of allocating all or a fair-share of CPUs to a job, PYTHIA allocates the fraction of this number that it estimates will enable the job to complete by its deadline. By "slowing down" the job while still meeting its deadline, PYTHIA is able to save resources and meet deadlines of other jobs. PYTHIA does not admit infeasible jobs (those that are likely to miss their deadlines) and thus, jobs submissions "fail fast" without wasting cloud resources. Extant systems admit all jobs and then "react" when a deadline passes by terminating or pre-empting a running job. PYTHIA uses jobs performance statistics from completed jobs to compute/update this allocation fraction online.

We implement PYTHIA for Mesos and empirically evaluate it using trace-based simulation of different cloud capacities. We use two real-world, 3-month, Hadoop traces from a production YARN system (contributed to us by an industry partner) for our experimentation. We evaluate PYTHIA and popular fair-share allocators, in terms of deadlines missed and useful work completed versus an oracle. We also consider both fixed and random deadline assignments. We find that PYTHIA achieves performance similar to that of the oracle in terms of deadlines met and useful work done. In addition, we find that PYTHIA is able to meet significantly more deadlines than the fair share allocators for the workloads, cluster sizes, and deadline assignments that we consider.

Algorithm 1 Job Completion Monitoring

1: **function** TRACK_JOB($compTime$, $requestedTasks$, $deadline$)
2: $deadlineCPUs = compTime/deadline$
3: $maxCPUs = min(requestedTasks, cloud\_capacity)$
4: $minReqRate = deadlineCPUs/maxCPUs$
5: **if** $minReqRate > CPUFrac$ **then**
6: $CPUFrac = minReqRate$
7: **end if**
8: **end function**

II. PYTHIA

PYTHIA is a deadline-aware resource allocator with admission control for resource negotiators that manage the execution of batch applications with deadlines, using resource-constrained, shared clouds. PYTHIA employes a black-box, framework-agnostic technique, to estimate the minimum number of CPUs (parallelism) that a job requires to meet its deadline. To enable this, PYTHIA employs a simple yet effective approach that does not require job clustering, modeling, sampling, or complex simulation.

A. Admission Control

To design PYTHIA, we make a number of simplifying assumptions. First, we assume that the number of tasks for a job (the division of its input size and the HDFS block size) is the maximum parallelization possible for the job. We refer to this number as the `requestedTasks` for the job. To determine how many CPUs to allocate to a new job, PYTHIA uses performance data from past jobs. PYTHIA analyzes each job when it completes and uses this information to estimate the number of CPUs that the job *would have needed* to have finished by its deadline (`deadlineCPUs`). PYTHIA also assumes that there is perfect parallelism (speedup per CPU).

PYTHIA tracks the maximal global fraction, which we call `CPUFrac`, which is `deadlineCPUs` over `requestedTasks` for jobs that complete. It multiplies this fraction by the number of tasks requested by frameworks for new jobs, to compute the number of CPUs to allocate to each job. Moreover, PYTHIA is *proactive* in that it prevents infeasible jobs (jobs likely to miss their deadlines) from ever entering the system and consuming resources wastefully. In this way, PYTHIA attempts to maximize the number of jobs that meet their deadline even under severe resource constraints (i.e. cloud capacity).

B. Algorithm

To bootstrap the system, PYTHIA sets `CPUFrac` to -1 and admits all jobs regardless of deadline; it allocates `requestedTasks` CPUs to the job. For any job for which there are insufficient resources for the allocation, PYTHIA allocates the number of CPUs available. When a job completes (either by meeting or exceeding its deadline), PYTHIA invokes the pseudocode function `TRACK_JOB` shown in Alg. 1 to potentially update `CPUFrac`.

`TRACK_JOB` calculates the minimum number of CPUs required (`deadlineCPUs`) if the job were to complete by its deadline, using its execution profile (available from the resource negotiator logs). Line 2 in the function is derived from the equality:

$$numCPUsAllocd * jobET = deadlineCPUs * deadline$$

On the left is the actual computation time by individual tasks, which we call `compTime` in the algorithm. `numCPUsAllocd` is the number of CPUs that the job used during execution and `jobET` is its execution time without queuing delay. The right side of the equation is the total computation time consumed across tasks if the job had been assigned `deadlineCPUs`, given this execution profile (`compTime`). `deadline` is the time (in seconds) specified in the job submission. By dividing `compTime` by `deadline`, we extract `deadlineCPUs` for this job.

Next, PYTHIA divides `deadlineCPUs` by the maximum number of CPUs allocated to the job. The resulting `minReqRate` is a fraction of the maximum that PYTHIA could have assigned to the job and still have it meet its deadline. PYTHIA compares `deadlineCPUs` against the global `CPUFrac` and updates `CPUFrac` if the former is larger. PYTHIA then uses `CPUFrac` as a prediction for this fraction for future jobs. `CPUFrac` is always less than or equal to 1. The tighter the deadline, the more conservative (nearer to 1) PYTHIA's resource provisioning will be.

Once `CPUFrac` has been initialized (from -1), PYTHIA employs it for admission control. To do so, PYTHIA multiplies `CPUFrac` by the number of tasks requested in the job submission (rounding to the next largest integer value). It uses this value (or the maximum cloud capacity, whichever is smaller) as the number of CPUs to assign to the job for execution. If this number of CPUs is not available, PYTHIA enqueues the job. PYTHIA performs this process each time a job is submitted or completes. It also updates the deadlines for jobs in the queue (reducing each by the time that has passed since submission), recomputes the CPU allocation of each enqueued job using the current `CPUFrac` value, and drops any enqueued jobs with infeasible deadlines.

PYTHIA admits jobs from the queue based on a pluggable priority policy. We have considered various policies for PYTHIA and use deadline maximization for the results in this study. In this policy, PYTHIA gives priority to jobs with a small number of tasks and greatest time-to-deadline. However, all of the policies that we considered (including shortest time-to-deadline) perform similarly. Once PYTHIA has selected a job for submission, it allocates the CPUs to the job and admits it to the system for execution. Once a job enters the system, its CPU allocation does not change.

III. Experimental Methodology

In this section, we describe the experimental methodology that we use to evaluate PYTHIA. We overview the trace-based simulation system, define our performance metrics, and present the deadline types that we consider.

We implement a discrete event simulator to empirically evaluate clouds under different resource constraints (number of instance cores available). Our system is based on Simpy [35] and replicates the execution behavior of production traces of big data workloads (cf Section IV). It supports multiple allocators that implement different policies for resource allocation and admission control. In this paper, we consider:

`NoDrop FS`: This allocator employs a fair sharing policy [4, 14, 15, 37, 45] without admission control. Its behavior is similar to that of the default allocator in Mesos and YARN. Its goal is to share the cluster resources fairly among frameworks and it is unaware of deadlines. Therefore, it never drops jobs even when there are insufficient resources to meet the deadlines or if the deadlines have passed.

`Reactive FS`: This allocator extends NoDrop FS and has no admission control. When a job exceeds its deadline, this allocator "reacts" and terminates the job, freeing resources in the cloud for use by other jobs.

`Oracle`: This allocator allocates the minimum number of resources that a job requires to meet its deadline. If sufficient resources are unavailable, the Oracle queues the job until the resources become available or until its deadline has passed (or is no longer achievable). This allocator is an oracle in the sense that it has future knowledge of actual execution time of jobs and thus will not admit jobs that will not meet their deadlines. However, it does not have a global view (i.e. ideal schedule). For the queued jobs, Oracle gives priority to jobs with fewer required resources and longer time until the deadline.

`PYTHIA`: As described in Section II, this allocator proactively drops, enqueues, or admits jobs submitted. It estimates the number of resources to allocate using the global *CPUFrac* which it computes from job performance histories. For the queued jobs, PYTHIA gives priority to jobs with fewer required resources and longer computation times. PYTHIA drops any jobs that are infeasible based on their deadlines.

Deadline Types. We evaluate the robustness of our approach by running experiments using different deadline types as is done in prior related work [13, 27, 42, 43, 50]. In particular, we assign deadlines that are multiples of the optimal execution time of a job (which we extract from our workload traces). We use two types of multiples: Fixed and variable. With fixed deadlines, we use a deadline of 2 times the optimal execution time as is done in [27, 50]. For variable deadlines, we compute deadline multiples by sampling distributions. We randomly pick a deadline between two possible values as is done in the Jockey study [13] from the sets with values $(1, 2)$ and $(2, 4)$, which we refer to as Jockey1x2x and Jockey2x4x. We also experiment with deadline multiples that are uniformly distributed in the intervals $[1, 3]$ and $[2, 4]$ as used in Aria [42, 43]; we refer to these experiments as Aria1x3x and Aria2x4x, respectively.

Evaluation Metrics. Our goal with PYTHIA is to maximize the number of satisfied deadlines and to avoid wasting resources on infeasible jobs. We assess the quality of PYTHIA using *Satisfied Deadlines Ratio (SDR)* and *Productive Time Ratio (PTR)*. SDR is the fraction of the jobs that completed before their deadline to the total number of submitted jobs. For the set of all the submitted jobs $J1, J2, ..., Jn$, let m be the subset of successful jobs $J1, J2, ...Jm$. Then SDR is: $\frac{\sum_{i=1}^{m} Ji}{\sum_{j=1}^{n} Jj}$. PTR is the fraction of the total computation time (across tasks) spent for jobs that satisfied their deadlines over the total computation time of all the jobs in the trace, regardless of whether they completed their execution. For the set of all the submitted jobs $J1, J2, ..., Jn$ with corresponding runtimes $T1, T2, ..., Tn$ we consider the subset of m successful jobs $J1, J2, ...Jm$. PTR then is: $\frac{\sum_{i=1}^{m} Ti}{\sum_{j=1}^{n} Tj}$.

Trace	CPUs	Jobs	Comp. Time (Hours)	1-Task Pct	1-Task Time Pct
TR1	9345	159194	8585673	58%	0.1%
TR2	24721	1140064	13301659	62%	0.3%

TABLE I: Summary of Traces. Columns are trace name, peak cloud capacity, total number of jobs, total computation time in hours, percentage of 1-task jobs, and percentage of 1-task job computation time.

IV. Workload Characterization

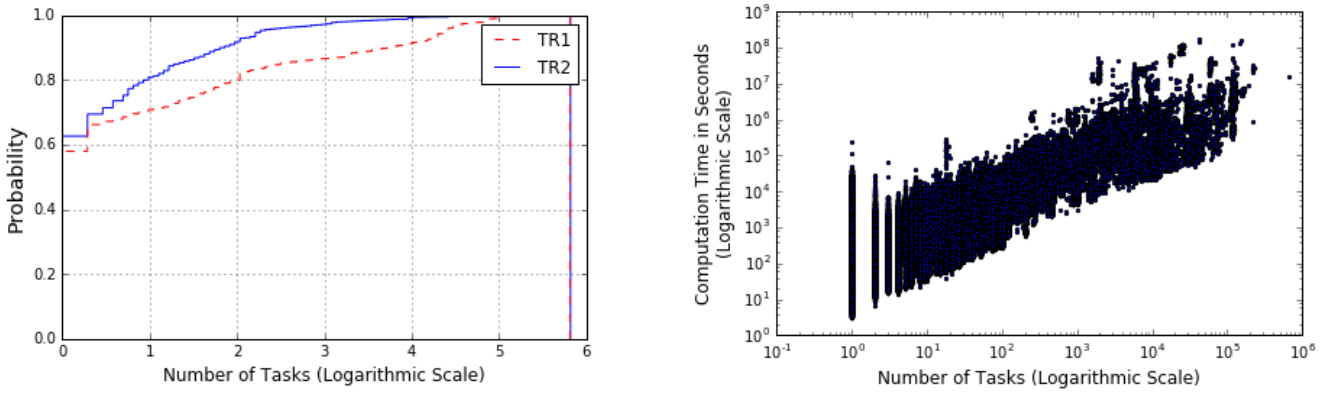
To evaluate PYTHIA, we use two 3-month traces from production Hadoop applications executing over different YARN system. The traces were recently donated to our research lab by an Industry partner. Each trace contains a job ID, job category, number of map and reduce tasks, map and reduce time (computation time across tasks), job runtime, among other data. We have no information about the scheduling policy or HDFS configuration used in each cluster. Thus we use and assume a minimum of CPU per task and use this minimum to derive cloud capacity; we are considering sub-portions of CPUs (vcores) as part of future work. PYTHIA uses the number of map tasks (as `requestedTasks`) and map time (as `compTime`) from the traces for simplicity; map task count and time dominate reduce task count and time for most jobs in both traces.

Table I summarizes the job characteristics of each trace. The table shows the peak cloud capacities (total number of CPUs), the total number of jobs, the total computation time across all tasks in the jobs, the percentage of jobs that have only one task, and the percentage of computation time that single-task jobs consume across jobs. We refer to the trace with 159194 jobs as TR2 and the trace with 1140064 jobs as TR2. The peak observed capacity (total number of CPUs) for TR1 is 9345 and for TR2 is 24721.

The table also shows that even though there are many single-task jobs, they consume a small percentage of the total computation time in each trace. To understand this characteristic better, we present the cumulative distribution of number of tasks per job in Figure 1 on a logarithmic scale. Approximately 60% of the jobs have a single task and 70-80% of the jobs have fewer than 10 tasks, across traces. Only 13% of the jobs in TR1 and 3% of the jobs in TR2 have more than 1000 tasks.

The right graph in the figure compares job execution time with the number of tasks per job (both axes are on a logarithmic scale) for the TR2 trace (TR1 exhibits a similar correlation). In both traces, 80% of the 1-task jobs and 60% of the 2-10 task jobs finish in fewer than 100 seconds. Their aggregate computation time is less than 1% of the total computation time of the trace. Jobs with more than 1000 tasks account for 98% and 94% of the total computation time for TR1 and TR2, respectively. Finally, job computation time varies significantly across jobs.

We have considered leveraging the job ID and number of map and reduce tasks to track repeated jobs, but find that for



(a) CDFs of the number of tasks per job in TR1 and TR2

(b) Job computation time vs number of tasks in TR2

Fig. 1: **Workload Characteristics:** Number of tasks per job and computation time relative to jobs size (in number of tasks). Small jobs are large in number but consume a very small proportion of trace computation time.

these real-world traces such jobs are small in number. In TR1, 18% of the jobs repeat more than once and 12% of the jobs repeat more than 30 times. In TR1, 25% of the jobs repeat more than once and 16% of the jobs repeat more than 30 times. Moreover, we observe high performance variation within each job class. Previous research has reported similar findings with production traces [13].

V. Empirical Evaluation

We evaluate PYTHIA using the production traces for different cloud capacities (number of CPUs) to evaluate its impact in both resource-constrained and resource-rich scenarios. Our cloud capacities range from 2250 and 15000 CPUs. We compare PYTHIA against different fair share schedulers, using two different deadline strategies, a random multiple (Jockey) and a uniform multiple (Aria) of the actual computation time, as described in Section III. We have also implemented a prototype of PYTHIA as an allocation module [29] on Apache Mesos 0.27.2 which will make available on Github following publication.

A. Fixed Deadlines

Figure 2 presents the deadline satisfaction ratio (SDR) for each allocator and multiple cloud capacities, when we employ deadlines that are 2 times the job runtime for TR1 (left graph) and TR2 (right graph). For both traces, PYTHIA meets significantly more deadlines than both of the fair share policies and performs similarly to the Oracle. Under tight resource constraints (the smallest cloud capacities), PYTHIA satisfies 295% more deadlines than NoDrop FS and and 143% more deadlines than Reactive FS.

As described previously, the Oracle does not have perfect information (i.e. it does not have a global optimal schedule) but it does know the actual job total computation time (`compTime`). Thus, it is able to assign the minimum number of CPUs to each job to satisfy its deadline. SDR for Oracle is not 100% because it must drop (refuse to admit) jobs for which there is insufficient capacity to meet their deadline.

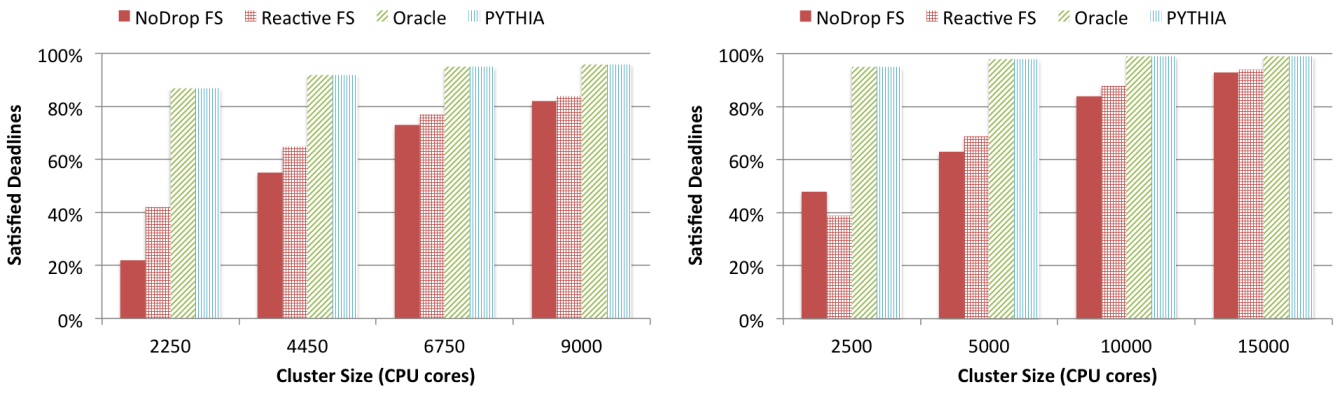
We next evaluate the useful work (PTR) that each allocator enables. PTR is the fraction of total computation time across all jobs, that is due to jobs that are admitted and that meet their deadlines. Figure 3 shows these results.

For PTR, PYTHIA results are similar to those of the *Oracle* and PYTHIA outperforms both fair share allocators across cloud capacities. In particular, PYTHIA facilitates up to 93% more productive time than to Reactive FS. With limited capacity, almost no useful work is achieved using NoDrop FS.

This pair of results shows that as resources become scarce, techniques without intelligent admission control for workloads with fixed deadlines lead increasingly to missed deadlines and wasted resources. This occurs because fair share allocators assign all or most CPUs (depending on their fair share fraction) required for the tasks requested. Because there is no deadline-awareness, these allocators do so regardless of whether or not the job is likely to meet its deadline. Jobs that miss their deadlines result in wasted (unproductive) work for the duration of the job for NoDrop FS and until the point the deadline is reached for Reactive FS.

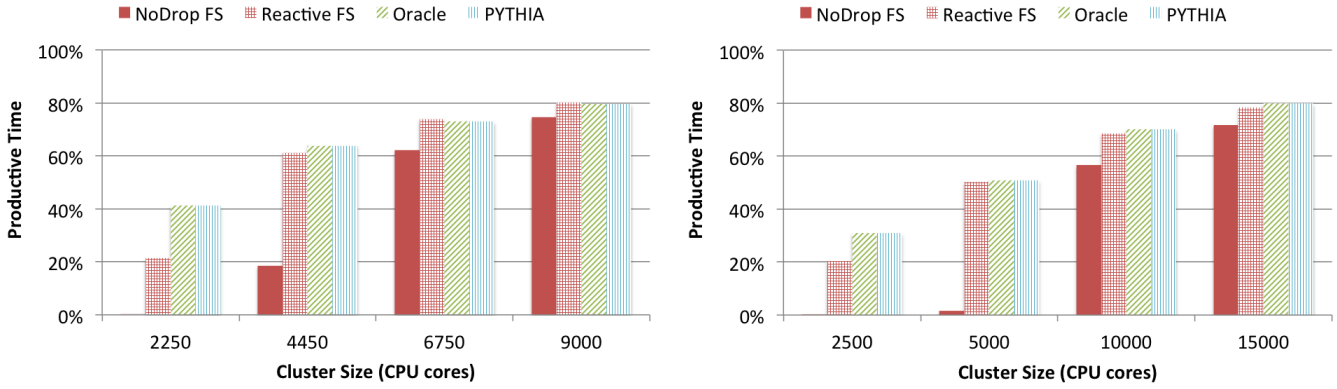
In constrained clouds, the fair share allocators satisfy deadlines without contributing much to PTR. For example, *NoDrop FS* successfully completes approximately 20% of the jobs in the TR1. Yet, its PTR is near zero. In constrained settings, fewer resources are shared between jobs, leading to a smaller fair share per job. For larger jobs, this fair share is insufficient to meet their deadlines. For jobs with a small number of tasks, this smaller fair share is still sufficient to meet deadlines. However, these jobs contribute minimally to PTR. As a result, the productive time achieved from the fair share allocators degrades rapidly as cloud capacity decreases even while meeting some deadlines.

The results from the PYTHIA allocator indicate that even a simple allocation and admission control strategy can restore the number of satisfied deadlines and useful work to near optimal. For unconstrained scenarios, all allocators perform similarly for both deadlines satisfied and productive computation time. Thus, PYTHIA can be used in either scenario to achieve the greatest deadline satisfaction.



(a) TR1 (b) TR2

Fig. 2: Satisfied Deadline Ratio (SDR) with Fixed Deadlines for TR1 (left graph) and TR2 (right graph). All jobs have deadline multiples of 2.



(a) TR1 (b) TR2

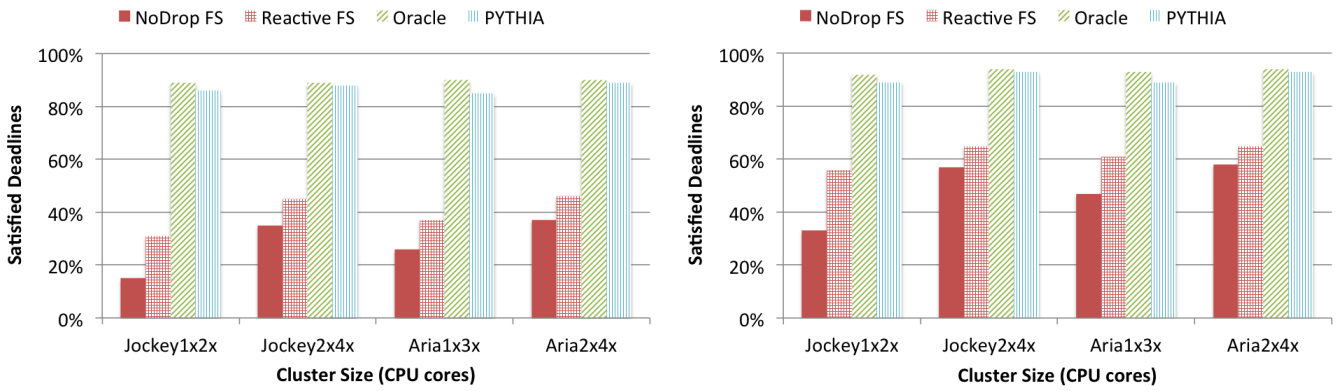
Fig. 3: Productive Time Ratio (PTR) with Fixed Deadlines for TR1 (left graph) and TR2 (right graph). All jobs have deadline multiples of 2.

B. Variable Deadlines

Next we consider the variable deadline assignments detailed under **Deadline Types** in Section III. Due to space constraints, we present results only for constrained cloud settings (2250 and 4500 cores) and for TR1. Our results for TR2 are similar to TR1 results and our results for unconstrained clouds show that all allocators behave similarly to each other.

Figure 4 shows the SDR for each allocator when we employ variable deadlines for two resource-constrained clouds. Using this deadline assignment, PYTHIA satisfies a similar number of deadlines as the Oracle allocator in both cases. In the worst case, PYTHIA is within 5% of the oracle. PYTHIA satisfies up to 177% more deadlines than Reactive FS on the 2250 CPU cloud and up to 58% more deadlines than Reactive FS on the 4500 CPU cloud.

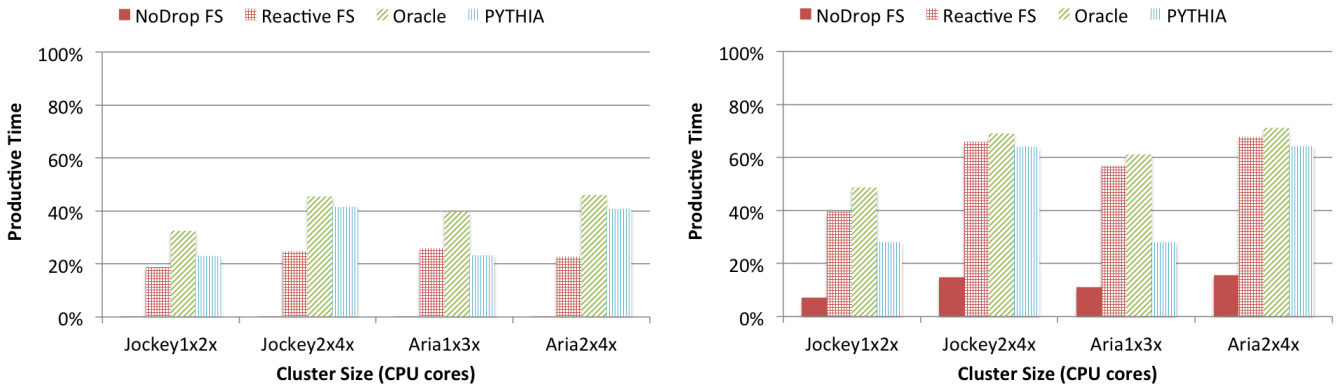
Figure 5 presents the PTR for these scenarios. Although PYTHIA meets significantly more deadlines than Reactive FS, it outperforms its PTR only for a subset of the deadline classes. This difference results because PYTHIA must be conservative (i.e. over-provisioning or dropping more jobs) to ensure that they meet their deadlines. That is, PYTHIA computes the `minReqRate` for jobs that now have deadlines that range from very strict (e.g. multiple of 1 which means the deadline is the same as its optimal runtime) to very loose (3 times this value). To ensure that as many jobs as possible meet their deadline and there is no wasted work, PYTHIA conservatively chooses a CPU allocation that will likely enable jobs with strict deadlines to succeed. For jobs with loose deadlines, this results in over-provisioning, preventing other jobs from sharing the cloud. In contrast, the *Oracle* knows the `minReqRate` that each job requires to satisfy its deadline and thus enables more free capacity on the cluster that can be used by other jobs. For cloud capacities of 2250 and 4500 CPUs, Reactive FS introduces from 24% to 30% and from 12% to 30% wasted work (computation time spent for jobs that miss their deadlines and are terminated midstream) respectively for the different deadlines of the evaluation. NoDrop FS, wastes 99% of the computation time on the cloud with 2250 CPUs and from 84% to 93% on the cloud with 4500 CPUs for the different deadline



(a) Cloud Capacity: 2250 CPU cores

(b) Cloud Capacity: 4500 CPU cores

Fig. 4: Satisfied Deadline Ratio (SDR) with Variable Deadlines for TR1 for cloud capacities of 2250 CPUs (left graph) and 4500 CPUs (right graph). Experiments denoted as 'Jockey' have deadline multiples picked randomly from a set with two values $(1, 2)$ and $(2, 4)$. Experiments denoted as 'Aria' have deadline multiples drawn from uniformly distributed intervals $[1, 3]$ and $[2, 4]$.



(a) Cloud Capacity: 2250 CPU cores

(b) Cloud Capacity: 4500 CPU cores

Fig. 5: Productivity Time Ratio (PTR) with Variable Deadlines for TR1 for cloud capacities of 2250 CPUs (left graph) and 4500 CPUs (right graph). Experiments denoted as 'Jockey' have deadline multiples picked randomly from a set with two values $(1, 2)$ and $(2, 4)$. Experiments denoted as 'Aria' have deadline multiples drawn from uniformly distributed intervals $[1, 3]$ and $[2, 4]$.

types because it doesn't drop jobs even after their deadline has passed. In contrast, PYTHIA keeps the wasted work near 0% for all the different deadlines and cluster capacities, because it adapts every time it encounters a stricter deadline. As part of future work, we are investigating ways to adapt PYTHIA's `CPUFrac` over time to improve its PTR performance.

VI. Related Work

Performance prediction: In order to allocate the required resources and meet job deadlines, much related work focuses on exploiting historic [8, 20, 25, 42, 43, 48, 50], and runtime [8, 18, 20, 21, 31, 42, 43, 49, 50] job information, while other research [8, 13, 23, 38, 41, 49] focuses on building job performance profiles and scalability models offline Although, effective in many situations, we show that approaches similar to these suffer when used under resource constrained settings.

Strategies that depend solely on repeated jobs, by definition, do not guarantee performance of ad-hoc queries. While approaches that use runtime models, sampling, simulations, and extensive monitoring, impose overheads and additional costs. Moreover, trace analysis in this paper and other research [13] shows that some production clouds have small ratio of repeated jobs and these jobs have often large execution times dispersion. Therefore, approaches based on past executions might not have the required mass of similar jobs over a short period of time in order to predict with high statistical confidence. Furthermore, the vast number of jobs have very short computation times [6, 13, 28, 30, 32]. Thus, approaches that adapt their initial allocation after a job has already started might be ineffective. Lastly, most of these approaches require task-level information, for the specific framework they target, either Hadoop [18, 20, 21, 23, 25, 31, 38, 42, 43, 48, 49, 50] or Spark [33, 44]. For this reason, they cannot be used on top of resource managers like Mesos.

PYTHIA in contrast, does not depend on job repetitions and can therefore target clouds with more diverse workloads;

it does not impose overheads to perform extensive runtime monitoring or use job sampling or offline simulations to predict performance. PYTHIA is also framework-independent because it does not require modeling of the different stages of any particular big data framework.

Sharing on Multi-tenant resource allocators: Cluster managers like Mesos [19] and YARN [40] enable the sharing of cloud and cluster resources by multiple data processing frameworks. Recent research [11, 16, 34] builds on this sharing, to allow users to run jobs without knowledge of the underlying data processing engine. In these multi-analytics settings, the goal of the resource allocator is to provide performance isolation to frameworks by sharing the resources between them [4, 5, 12, 15]. The sharing policies in these works are deadline-agnostic. To meet deadlines, administrators currently use dedicated clouds, statically split their clouds with the use of a capacity scheduler [5], or require users to reserve resources in advance [7, 39]. Such approaches incur costs for additional clouds or are inefficient and impractical, as they limit peak cloud capacity. Moreover, resources are underutilized when critical jobs are not running.

Another issue encountered in multi-analytics systems, is that frameworks like Hadoop and Spark that run on top of these resource allocators have their own intra-job schedulers that greedily occupy the resources allocated to them, even when they are not using them [10, 17, 47]. *CARBYNE* [17] attempts to address this issue by exploiting task-level resource requirements information and DAG dependencies. It also uses prior runs of recurring jobs to estimate task demands and durations. Then, it intervenes both at the higher level, on the resource allocator, and internally, on framework task schedulers. It withholds a fraction of job resources from jobs that do not use them while maintaining similar completion times. PYTHIA in contrast, introduces framework-independent admission control that these resource allocators can use to support dynamic sharing of the cloud and deadline driven workflows for either Hadoop or Spark jobs, without requiring task-level information or depending on recurring jobs.

Admission control: Admission control has been suggested as a cloud solution for SaaS providers to effectively utilize their clusters and meet Service Level Agreements (SLAs) [22, 46], to provide map-reduce-as-a-service [9], and to resolve blocking caused by greedy YARN allocations [47]. PYTHIA is similar in that but targets multi-analytics, resource-constrained clouds. We design PYTHIA for use by resource managers for deadline-driven big data workloads, to be framework and task independent.

VII. Conclusions and Future Work

We present PYTHIA, a novel admission control system for resource negotiators for big data multi-analytics systems, such as Apache Mesos and YARN. PYTHIA adds support for deadlines and facilitates more effective resource use when resources are constrained due to physical limitations (private clouds) and cost contraints (when using public clouds), two increasingly common big data deployment scenarios. PYTHIA is a black box, framework agnostic approach that estimates the CPU resources that a job requires to meet its deadline, based on historic deadline and runtime information of completed jobs. PYTHIA admits jobs when the estimated resources are available and drops them when their deadlines are no longer feasible. Thus, any job not admitted "fails fast" and wastes no resources, enabling more jobs to be admitted and meet their deadlines.

We empirically evaluate PYTHIA for different cloud capacities and deadline assignments, using trace-based simulation, driven by production YARN traces. Our results show that PYTHIA is able to meet significantly more deadlines compared to existing fair-share approaches. Unlike them wasting a big portion of the computation time for jobs that don't meet their deadlines, PYTHIA almosts eliminates resource waste while maintaining high levels of completed useful work, for the workloads, cloud capacities, and deadline assignments that we consider. Moreover, we have implemented a prototype of PYTHIA as an allocation module [29] for Apache Mesos 0.27.2 which we will make available following publication.

As part of future work, we are exploring how to extend PYTHIA to adapt to deadline variability and changing resource constraints. PYTHIA currently is conservative and only increases its resource provisioning rate (`CPUFrac`) as it encounters tighter deadlines. This attempts to minimize wasted computation, but also leaves room for improvement (e.g. in the PTR (useful work) metric). We are currently considering how to decrease this rate when we discover that it has been over-conservative using a sliding window and by differentiating between long and short running jobs. We are also considering an approach that incentivizes users to assign realistic deadlines to jobs, both to avoid over allocating resources and to protect from unnecessary increases on the resource provision rate. Finally, we are extending PYTHIA to consider overheads that lead to imperfect parallelization in our calculation of the minimum resource requirements of jobs, by tracking and incorporating divergence in observed and estimated job runtimes.

This work is funded in part by NSF (CCF-1539586, ACI-1541215, 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*. http://hadoop.apache.org/ [Online; accessed 2-January-2017].

[2] *Apache Spark*. http://spark.apache.org/ [Online; accessed 2-January-2017].

[3] *Apache Storm*. http://storm.apache.org/ [Online; accessed 21-January-2016].

[4] A. Bhattacharya et al. Hierarchical scheduling for diverse datacenter workloads. In: *ACM SoCC*. 2013.

[5] *YARN Capacity Scheduler*. https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

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

[7] C. Curino et al. Reservation-based Scheduling: If You're Late Don't Blame Us! In: *ACM Symposium on Cloud Computing*. 2014, pp. 1–14.

[8] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In: *ACM SIGPLAN Notices* 49.4 (2014), pp. 127–144.

[9] J. Dhok, N. Maheshwari, and V. Varma. Learning based opportunistic admission control algorithm for MapReduce as a service. In: *India software engineering conference*. ACM. 2010, pp. 153–160.

[10] S. Dimopoulos, C. Krintz, and R. Wolski. Big Data Framework Interference In Restricted Private Cloud Settings. In: *IEEE International Conference on Big Data. IEEE*. 2016.

[11] K. Doka et al. Mix'n'Match Multi-Engine Analytics. In: *IEEE International Conference on Big Data. IEEE*. 2016.

[12] *YARN Fair Scheduler*. https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[13] A. D. Ferguson et al. Jockey: guaranteed job latency in data parallel clusters. In: *ACM European Conference on Computer Systems*. ACM. 2012, pp. 99–112.

[14] E. Friedman, A. Ghodsi, and C.-A. Psomas. Strategyproof allocation of discrete jobs on multiple machines. In: *ACM EC*. 2014.

[15] A. Ghodsi et al. Dominant resource fairness: Fair allocation of multiple resource types. In: *NSDI*. 2011.

[16] I. Gog et al. Musketeer: all for one, one for all in data processing systems. In: *European Conference on Computer Systems*. 2015.

[17] R. Grandl et al. Altruistic scheduling in multi-resource clusters. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2016.

[18] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. In: *VLDB* 4.11 (2011), pp. 1111–1122.

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

[20] M. Hu et al. Deadline-Oriented Task Scheduling for MapReduce Environments. In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2015, pp. 359–372.

[21] Z. Huang et al. RUSH: A RobUst ScHeduler to Manage Uncertain Completion-Times in Shared Clouds. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 242–251.

[22] M. Islam et al. Towards provision of quality of service guarantees in job scheduling. In: *IEEE International Conference Cluster Computing*. IEEE. 2004.

[23] V. Jalaparti et al. Bridging the tenant-provider gap in cloud services. In: *ACM Symposium on Cloud Computing*. 2012.

[24] K. Kc and K. Anyanwu. Scheduling hadoop jobs to meet deadlines. In: *International Conference on Cloud Computing*. 2010, pp. 388–392.

[25] P. Lama and X. Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In: *ACM International Conference on Autonomic Computing*. 2012, pp. 63–72.

[26] S. Li et al. WOHA: deadline-aware map-reduce workflow scheduling framework over hadoop clusters. In: *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE. 2014, pp. 93–103.

[27] J. Liu, H. Shen, and H. S. Narman. CCRP: Customized Cooperative Resource Provisioning for High Resource Utilization in Clouds. In: *IEEE International Conference on Big Data. IEEE*. 2016.

[28] J. Liu, H. Shen, and H. S. Narman. CCRP: Customized Cooperative Resource Provisioning for High Resource Utilization in Clouds. In: *Resource* 40 (2016), p. 60.

[29] *Mesos Modules*. http://mesos.apache.org/documentation/latest/allocation-module/.

[30] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In: *ACM SIGMOD International Conference on Management of data*. 2009, pp. 165–178.

[31] J. Polo et al. Performance-driven Task Co-Scheduling for MapReduce Environments. In: *IEEE Network Operations and Management Symposium*. 2010, pp. 373–380.

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

[33] S. Sidhanta, W. Golab, and S. Mukhopadhyay. OptEx: A Deadline-Aware Cost Optimization Model for Spark. In: *arXiv preprint arXiv:1603.07936* (2016).

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

[35] *Simpy*. https://simpy.readthedocs.io/en/latest/.

[36] R. Sumbaly, J. Kreps, and S. Shah. The big data ecosystem at linkedin. In: *ACM SIGMOD International Conference on Management of Data*. 2013, pp. 1125–1134.

[37] J. Tan et al. Multi-resource fair sharing for multiclass workflows. In: *ACM SIGMETRICS Performance Evaluation Review* 42.4 (2015).

[38] F. Tian and K. Chen. Towards optimal resource provisioning for running mapreduce programs in public clouds. In: *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE. 2011, pp. 155–162.

[39] A. Tumanov et al. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In: *European Conference on Computer Systems*. 2016, p. 35.

[40] V. K. Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. In: *ACM Symposium on Cloud Computing*. 2013.

[41] S. Venkataraman et al. Ernest: efficient performance prediction for large-scale advanced analytics. In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 2016, pp. 363–378.

[42] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In: *ACM International Conference on Autonomic Computing*. 2011, pp. 235–244.

[43] A. Verma et al. Deadline-based workload management for MapReduce environments: Pieces of the performance puzzle. In: *2012 IEEE Network Operations and Management Symposium*. IEEE. 2012, pp. 900–905.

[44] K. Wang and M. M. H. Khan. Performance Prediction for Apache Spark Platform. In: *2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*. IEEE. 2015, pp. 166–173.

[45] W. Wang, B. Liang, and B. Li. Multi-resource fair allocation in heterogeneous cloud computing systems. In: *IEEE Trans. Parallel Distrib. Syst.* 26.10 (2015).

[46] L. Wu, S. K. Garg, and R. Buyya. SLA-based admission control for a Software-as-a-Service provider in Cloud computing environments. In: *Journal of Computer and System Sciences* 78.5 (2012), pp. 1280–1299.

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

[48] N. Zaheilas and V. Kalogeraki. Real-time scheduling of skewed mapreduce jobs in heterogeneous environments. In: *11th International Conference on Autonomic Computing (ICAC 14)*. 2014, pp. 189–200.

[49] W. Zhang et al. Mimp: Deadline and interference aware scheduling of hadoop virtual machines. In: *IEEE Cluster, Cloud and Grid Computing*. 2014, pp. 394–403.

[50] Z. Zhang et al. Automated profiling and resource management of pig programs for meeting service level objectives. In: *ACM International Conference on Autonomic computing*. 2012, pp. 53–62.