Tyrex

Size-Based Resource Allocation in MapReduce Frameworks

Ghit, Bogdan; Epema, Dick

DOI
10.1109/CCGrid.2016.82

Publication date
2016

Document Version
Accepted author manuscript

Published in
Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016

Tyrex: Size-based Resource Allocation in MapReduce Frameworks

Bogdan Ghiț[†] and Dick Epema[†§]
[†]Delft University of Technology, the Netherlands
[§]Eindhoven University of Technology, the Netherlands
{b.i.ghit,d.h.j.epema}@tudelft.nl

***Abstract*—Many large-scale data analytics infrastructures are employed for a wide variety of jobs, ranging from short interactive queries to large data analysis jobs that may take hours or even days to complete. As a consequence, data-processing frameworks like MapReduce may have workloads consisting of jobs with heavy-tailed processing requirements. With such workloads, short jobs may experience slowdowns that are an order of magnitude larger than large jobs do, while the users may expect slowdowns that are more in proportion with the job sizes. To address this problem of large job slowdown variability in MapReduce frameworks, we design a scheduling system called TYREX that is inspired by the well-known TAGS task assignment policy in distributed-server systems. In particular, TYREX partitions the resources of a MapReduce framework, allowing any job running in any partition to read data stored on any machine, imposes runtime limits in the partitions, and successively executes parts of jobs in a work-conserving way in these partitions until they can run to completion. We develop a statistical model for dynamically setting the runtime limits that achieves near-optimal job slowdown performance, and we empirically evaluate TYREX on a cluster system with workloads consisting of both synthetic and real-world benchmarks. We find that TYREX cuts in half the job slowdown variability while preserving the median job slowdown when compared to state-of-the-art MapReduce schedulers such as FIFO and FAIR. Furthermore, TYREX reduces the job slowdown at the 95^{th} percentile by more than 50% when compared to FIFO and by 20-40% when compared to FAIR.**

I. Introduction

Data-processing frameworks such as MapReduce that can be used for large-scale analytics and small interactive queries may face workloads of jobs with heavy-tailed processing requirement distributions. As has been abundantly clear both from theoretical analysis of queueing systems [14], [15] and from experience with actual deployments of MapReduce and other frameworks [27], [31], such workloads usually lead to job slowdowns of small jobs that are at least an order of magnitude larger than those of long jobs, which may be intolerable to users. In this paper, we present the design and analysis of TYREX, [1] a MapReduce scheduler that aims at *reducing the slowdown variability* in workloads with many short jobs.

Despite the plethora of performance related studies of data-intensive workloads, state-of-the-art MapReduce schedulers still lead to high slowdown variability. In our experience with processing monitoring data from the BitTorrent global network using a MapReduce-based logical workflow [18], we have found that 15% of the jobs account for 80% of the total load, and that 65% of the jobs complete in a minute. Similarly, several studies on the performance of modern production clusters in commercial companies like Google and Facebook [7], [20], [24] report *heavy-tailed* workloads with highly variable runtimes. Allowing such workloads to run in non-isolated environments and to greedily share the system resources severely impacts the performance of short jobs as they experience long delays due to large jobs ahead of them.

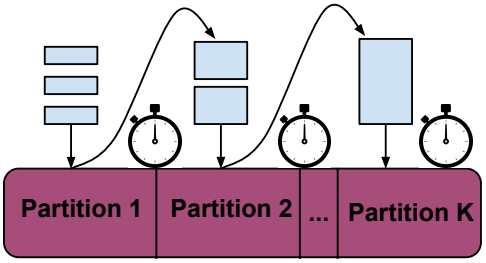


Fig. 1. Schematic overview of the queueing model employed by TYREX with resource partitioning and timers for migrating jobs.

In Figure 1 we illustrate the scheduling model employed by the TYREX scheduler. TYREX uses *resource partitioning* and *work-conserving job migration* across these partitions as its two main principles. A common way of partitioning the resources of a datacenter is to allocate disjoint sets of machines to multiple instances of the MapReduce framework [12]. However, this scheduling model is not attractive for jobs that are moved across partitions but still require access to the same data, as the cost of replicating the data across partitions may be prohibitive. Instead, TYREX operates within a single MapReduce framework so that jobs running in any partition may read data stored on any machine.

For isolating the sets of jobs with processing requirements (their *sizes*) in different ranges, TYREX imposes runtime limits (*timers*) with increasing values which limit the amounts of processing time jobs may receive in the partitions—jobs that exceed the timer of one partition are migrated to the next, retaining the work they have already completed. As fixed timers are difficult to configure because they require trial executions of every new workload to find their optimal values, we propose a method to *dynamically adapt the timers* based on statistical properties of the job size distribution.

TYREX is inspired by the TAGS policy [14], but there are a few important differences. First, as MapReduce frameworks are usually deployed in datacenters, TYREX has to divide the framework capacity across the partitions, rather than having predetermined single-server partitions as in TAGS. Secondly, MapReduce jobs are conveniently parallel jobs that only require

[1]Inspired by the dinosaur Tyrannosaurus rex, known for its long, heavy tail.

TABLE I. A POLICY FRAMEWORK FOR OPTIMIZING JOB SLOWDOWN IN DIFFERENT SYSTEM MODELS.

Policy	System Model	Job Size	Preemptive	Waste	Utilization	Description
PSJF [29]	single-server	known	yes	none	best	run the job with shortest original size
SRPT [17]	single-server	known	yes	none	best	run the job which can finish earlier
SITA [16]	distr. servers	known	no	high	worst	each server is assigned only jobs of a given size
TAGS [14]	distr. servers	unknown	no	high	worst	each server limits the amount of CPU per job
FAIR [4]	datacenter	unknown	yes	low	high	weighted fair-sharing
TYREX	datacenter	unknown	yes	none	high	see Section IV

synchronization between the map and reduce phases, which makes them malleable or elastic with the opportunity to run multiple jobs simultaneously as opposed to the rigid job model supported by the FIFO servers in TAGS. Finally, having a shared underlying distributed filesystem for all partitions of a MapReduce framework, TYREX enables a work-conserving mechanism for moving a job from one partition to the next without loosing the intermediate results, as happens with TAGS.

In this paper, we make the following contributions:

1) We design TYREX, a datacenter MapReduce job scheduler that places jobs across multiple partitions of a single MapReduce framework in order to isolate the sets of jobs of very different sizes. TYREX assumes no prior job size information and moves jobs across system partitions without loosing their completed work (Sections IV-A and IV-B).
2) We incorporate in TYREX two policies to migrate jobs across partitions that are differentiated by the type of timer they use in partitions, which may be *fixed* or *dynamic*. To adapt the timers dynamically, we propose a statistical technique to identify jobs that are likely to monopolize a given partition for a long time (Sections IV-C and IV-D).
3) With a set of experiments in a real multicluster system, we evaluate TYREX relative to standard MapReduce schedulers and assess the impact of different aspects of its operation on the job slowdown variability. We show that our scheduler delivers very low slowdown variability without a large impact on the median slowdown for several representative MapReduce workloads (Sections V and VI).

II. PROBLEM STATEMENT

In this section we explain the problem of job slowdown variability in MapReduce workloads and we formulate the goals of our TYREX. To capture the delay sensitivity of jobs of different sizes, we consider the *job slowdown* as a metric, defined for a job as the ratio of its response time and its wall-clock time in an empty system – when a fixed set of resources (in our case, the entire capacity of the system) are allocated to it. To formulate the goal of our scheduler, let F be the cumulative distribution function of the job slowdown when executing a certain workload. We define the *job slowdown variability at the q^{th} percentile*, denoted by $V_F(q)$, as the q^{th} percentile of F normalized by the median job slowdown, that is:

$$V_F(q) = \frac{F^{-1}(q)}{F^{-1}(50)}. \quad (1)$$

We call $V_F(95)$ the *overall job slowdown variability* of the workload. Our target is to *minimize* both the median job slowdown and the overall job slowdown variability of MapReduce workloads.

We consider job slowdown as a more fundamental metric than response time, and the problem of large job slowdowns as a more fundamental problem than large response times. The latter are continously being improved by better hardware, leading to higher speedups of applications and faster data transfers. In contrast, even though there may be shifts in the balance of the speeds and capacities of computer systems hardware, (large) job slowdowns will continue to exist in systems with contention for resources.

Many MapReduce clusters execute workloads that are often characterized by heavy-tailed processing requirement distributions [7], [20]. These workloads contain jobs with the amount of data to be processed or the sum of the execution times of all tasks of a job, that may vary by several orders of magnitude. In systems with such workloads, the small jobs suffer because they may be delayed for a relatively very long time due to long jobs ahead of them [31]. Nevertheless, the users of clusters or datacenters may expect their jobs to be delayed proportionally to their processing requirements.

In this paper we will generalize the TAGS policy to scheduling MapReduce workloads with heavy-tailed job-size distributions running in partitioned datacenters with each partition having a runtime limit. We will show that this way of scheduling MapReduce jobs outperforms FAIR, the most popular MapReduce scheduler, with respect to both the median slowdown and the slowdown variability for a broad range of job size distributions.

III. BACKGROUND

In this section we present an overview of the main scheduling disciplines that optimize the mean response time or the mean job slowdown in both single-server and distributed-server systems. In Table I, we show the main characteristics of several policies which have been investigated in the past. In single-server systems, policies that are biased towards short jobs, also known as SMART policies [30], are to be preferred as they prevent short jobs from experiencing long delays. The fundamental idea behind these policies is to prioritize short jobs over longer ones like Preemptive-Shortest-Job First (PSJF [29]) and Shortest-Remaining-Processing-Time (SRPT [17]). Although SRPT is optimal with respect to mean response time, the policy is rarely used in practice as it may lead to starvation of long jobs in order to help the short ones [15]. Another reason for the limited popularity of SRPT is that its effectiveness relies on preemption, which is difficult to implement for long jobs that can easily overflow the memory [16]. Instead, in supercomputers where jobs may be served by multiple hosts, size-based partitioning is often employed to isolate the performance of jobs with very unbalanced processing requirements [9].

When the job size distribution and the individual sizes of (rigid) jobs are known, the servers of a distributed-server system using the FCFS policy can be configured to serve each

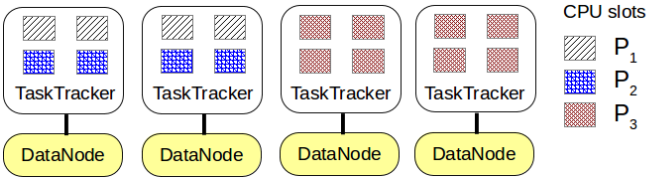


Fig. 2. The fine-grained resource partitioning employed by TYREX with CPU slots allocated across three partitions.

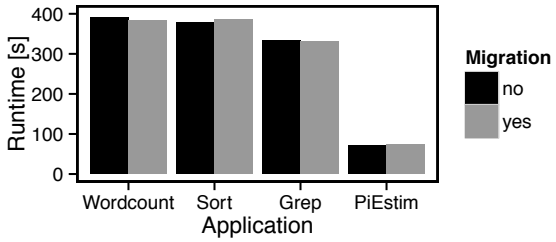


Fig. 3. The job runtime *without* migration versus the job runtime *with* migration across two partitions in a 20-worker Hadoop framework.

only the jobs whose sizes are in a specific range (the Size Interval Task Assignment (SITA) policy [15]). It can be shown that when the size ranges are chosen in such a way that all servers have the same load, the SITA policy is optimal in terms of the average job slowdown if the job-size distribution is heavy-tailed. The intuition behind this result is that in SITA, the job-size variability of each server is very much reduced. The SITA policy can be generalized for unknown job sizes through a simple yet very efficient technique that guesses the job sizes by killing them when they exceed the maximum runtime of a server and restarting them on the service with the next higher runtime range (the Task Assignment based on Guessing Sizes (TAGS) policy [14]). Traditionally, these have been successfully implemented in distributed-server systems, but they have not been used so far in clusters or datacenters for fear that they may lead to system fragmentation and underutilization of resources.

IV. The Tyrex Scheduler

In this section, we present a scheduling model to reduce the variability in job slowdown in MapReduce. Our scheduler, TYREX, assumes no prior knowledge about the jobs, divides the MapReduce framework computing capacity in disjoint partitions, and migrates jobs across those partitions. We propose two policies used by TYREX to confine jobs with similar processing requirements to separate partitions.

A. Design Considerations

TYREX resembles the structure of the TAGS policy, but there are three key elements in which it is different. First, TAGS was designed for a distributed-server model in which each host is a single multi-processor machine that can only serve one job at a time. In contrast, TYREX targets a datacenter environment in which the system capacity is divided across partitions with many resources. As a result, instead of only having the timers as parameters as in TAGS, in our model we also have the partition capacities as parameters.

Secondly, TAGS assumes simple, rigid (sequential or parallel) non-preemptible jobs that may only run on a single host until completion. In contrast, our (MapReduce) job model is more complex as there are intra-job data precedence constraints (map before reduce) and data locality preferences (of map tasks), and as jobs are elastic (or malleable) and can run simultaneously, taking any resources they can get when it is their turn.

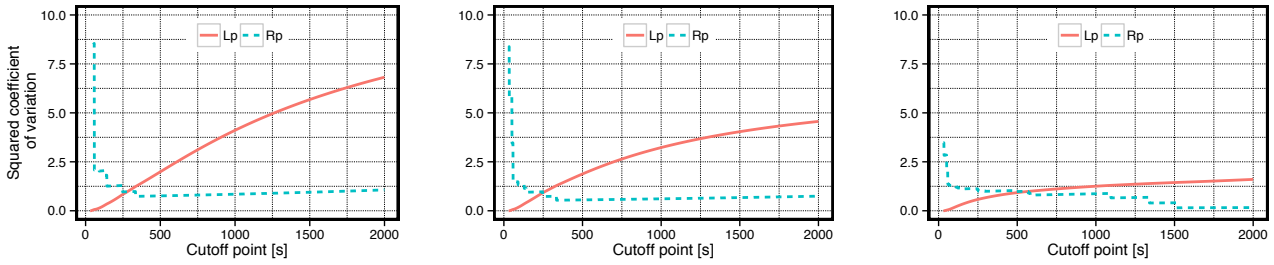
Thirdly, TAGS does not preserve the state of a job when it moves it from one server to the next. As a consequence, long jobs will get killed at every server except at the one where they run to completion, at every step losing all the work performed and thus wasting CPU time. Instead, TYREX takes a work-conserving approach by allowing jobs that are being moved from one partition to the next to retain their work and to gracefully resume their executions without redoing previously completed work.

B. System Model

The main design elements of our scheduler are *resource partitioning* and *migration of jobs* from one partition to another. Frameworks such as MapReduce employ a fine-grained resource sharing model, with processors divided into slots and with jobs divided into short tasks that run on slots. Therefore, we partition the compute slots of the MapReduce framework into disjoint partitions of fixed sizes, while keeping the storage accessible for all processors. This way of partitioning is attractive for jobs that may be moved across partitions but still require access to the same data, as there is no need of replicating the data across the partitions. TYREX operates within a single MapReduce framework and partitions the compute slots of this framework into some number K of partitions, each with its own queue of jobs with similar processing requirements, that all share access to the same underlying filesystem. TYREX allocates fixed capacities to its system partitions. The fraction of compute slots allocated to partition P_k represents its capacity and is denoted by $C_k, k = 1, 2, \ldots, K$. We assume jobs to be served in FIFO order in all partitions. In Figure 2 we give an example of a standard MapReduce framework which uses fine-grained resource partitioning to split its CPU slots among three partitions ($C_1 = 25\%$, $C_2 = 25\%$, and $C_3 = 50\%$). All three partitions have access to the data stored in HDFS.

Job migration in TYREX is facilitated by the distributed file system that is shared across the whole framework, with the intermediate results of tasks executed within any partition being persistent and visible after a job has been moved to another partition. To avoid wasted work, the scheduler allows a job to finish its running tasks in a given partition after the timer has expired. As TYREX migrates jobs across partitions of a single framework, the cost of migrating a job across partitions is zero. In Figure 3 we assess the overhead of moving jobs across different partitions in a 20-node Hadoop framework. We set two equally sized partitions and we measure the job runtime *without* and *with* migration of four MapReduce applications (Wordcount, Sort, Grep, and PiEstimator) executing 600 map tasks and 60 reduce tasks. To execute a job without migration, we set the timer of P_1 to ∞. To migrate a job across the two partitions, we set the timer of P_1 to 50% of the job size. As expected, TYREX has no overhead in migrating jobs across partitions.

TYREX decides which jobs should be migrated from one partition to the next whenever a task of a job in any partition



(a) HVW (b) MVW (c) LVW

Fig. 4. The squared coefficient of variation of L_p versus the squared coefficient of variation of R_p in our HVW, MVW, and LVW workloads for cutoff points p in the range between 30-2000 s.

is completed. To do so, TYREX uses one of two policies, STATICTAGS and DYNAMICTAGS, which we present in the remainder of this section. For both our policies, we define the *current partial size* of a job in a given partition at time t as the total runtime of its tasks completed in that partition at time t.

C. The STATICTAGS Policy

As we assume that the processing requirements of jobs cannot be anticipated, we cannot adapt the SITA [15] policy to our situation, but we need a variation of the TAGS [14] policy in order to differentiate job sizes. Towards this end, we define for every partition $k, k = 1, 2, \ldots, K$ a timer T_k which is set to the total amount of service that jobs may receive while they are in partition P_k. Any job submitted to TYREX is first dispatched to partition P_1. When the partial size of a job in any partition P_k exceeds the timer T_k, TYREX moves the job to the next partition P_{k+1}.

As we are especially interested in workloads with heavy-tailed size distributions and many short jobs, we can expect that the timer T_k of partition P_k should be set to a value that is (much) larger than the timer T_{k-1} of partition P_{k-1}, $k = 2, 3, \ldots, K$. We set T_K to ∞. However, setting the timers is difficult, and may in practice have to be repeated often. Unfortunately, the optimal timers of TAGS in distributed server systems already have complex forms even for well-behaved distributions like Pareto [14]. Hence, we take an experimental approach to determining the optimal timers with the STATICTAGS policy while keeping the partition capacities fixed.

D. The DYNAMICTAGS Policy

In this section, we present the DYNAMICTAGS policy in which timers are dynamic, so the problem of setting their values disappears. We take a statistical approach to assessing the job size variability, which we apply to three realistic MapReduce workloads. The basic idea behind DYNAMICTAGS is to reduce the variability of the current partial job sizes in any partition by migrating those jobs that are likely to require significantly more processing time than the rest of the jobs in the same partition.

In order to present DYNAMICTAGS, we need the following definitions. Let X be a positive, real stochastic variable (e.g., corresponding to a heavy-tailed job-size distribution), let CV be its coefficient of variation, and let p be a cutoff point in that distribution. Similarly to previous work [14], [15], we measure the variability of the job-size distribution using the squared CV, which is the ratio between the variance and the squared mean of X. We define $L_p = \min(X, p)$ and $R_p = X - p$ when $X > p$ as two random variables so that $X = L_p + R_p$ for any cutoff point p. To balance the variability across L_p and R_p, we seek a cutoff point p for which the values of the squared CV of L_p and R_p are equal, and we call that p the *optimal cutoff point* (we implicitly assume that there is a unique point with this property, which is always the case for our distributions).

To put the DYNAMICTAGS policy into perspective, we consider the distribution of job sizes in three workloads, i.e., HVW, MVW, and LVW, which we define in Section V. As has been abundantly shown in recent studies, MapReduce workloads may have very variable job size distributions [20], [8]. In particular, the squared CV values of the HVW, MVW, and LVW workloads are 20, 10, and 4, respectively. In Figure 4 we show the values of the squared CV of L_p and R_p for cutoff points p in each workload in the range of 30-2000 s. We see that increasing the value of the cutoff point has opposite effects on the variability in L_p and R_p, with the squared CV of L_p being maximum when the squared CV of R_p is minimum and vice versa. Of course, L_p (R_p) is close to the complete workload for very large (small) values of p, respectively.

More importantly, we find that the variabilities in L_p and R_p are unbalanced when the squared CV of L_p has a value that is higher than 2. As we are covering with our workloads a wide range of job size variabilities (see Table II) for which 2 turns out to be a good value (see the results in Section VI-D), we conclude that having a squared CV higher than 2 in L_p is a good indicator of unbalanced job sizes. Hence, we aim for a squared CV in L_p that is lower than 2. Figure 4 also shows that the squared CV of R_p is flat and relatively low (below 2), which means that further splitting R_p, and so having more than two partitions, is not very promising. Indeed, several experiments we did with three partitions confirmed this conclusion.

The DYNAMICTAGS policy now works in the following way. When it is invoked, it checks all partitions too see whether the value of the squared CV of the distribution of the current partial job sizes is higher than 2. If this does not hold for a partition, DYNAMICTAGS does not migrate any job from it. Otherwise, DYNAMICTAGS determines the optimal cutoff point in the distribution of the current partial job sizes, and uses that cutoff point as the value of the *dynamic timer* to migrate those jobs that exceed it to the next partition.

From a queueing-theory perspective, L_p captures the notion of young jobs, while R_p represents the residual lifetime of jobs.

In particular, if the distribution of job sizes is heavy-tailed then the residual lifetime of young jobs is stochastically smaller than the residual lifetime of old jobs. As a consequence, the DYNAMICTAGS policy seeks in any partition P_k a cutoff point p so that P_k only serves young jobs that are likely to leave the system soon. In contrast, old jobs with larger residual lifetimes are migrated to the next partition.

V. EXPERIMENTAL SETUP

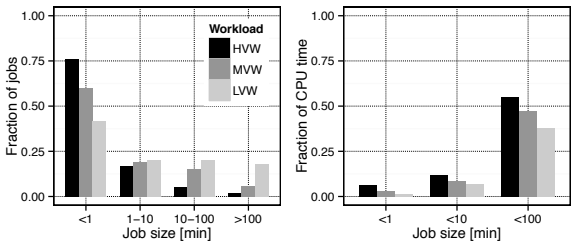
In this section we present the workloads and the configuration of the infrastructure we use for the experimental evaluation of TYREX. To design our workloads we use a comprehensive set of representative MapReduce applications, including both standard benchmarks and complex, real-world workflows. In this paper we take an experimental approach and we evaluate TYREX by means of experiments in a real-world multicluster system. The total time used for experimentation exceeded 20,000 hours system time.

Standard benchmarks. The Hadoop distribution provides a set of synthetic benchmarks abundantly used in performance evaluation studies of MapReduce frameworks. We use the following applications from Hadoop: Wordcount, PiEstimator, Sort, and Grep. Wordcount and PiEstimator are CPU-intensive applications used to retrieve the number of occurences of each unique word in a given set of input files and to estimate the real value of π using the quasi-Monte Carlo algorithm. In contrast, Grep and Sort are disk intensive applications used to search for a given pattern in the input files and to generate the content of the input files in non-decreasing order.

Complex workflows. BTWORLD is a complex and very challenging MapReduce-based logical workflow which we designed to observe the evolution of the global-scale peer-to-peer system BitTorrent [18]. The workflow consists of 26 MapReduce jobs with different resource bottlenecks (CPU, memory, or disk) and is used for processing monitoring data collected periodically from the BitTorrent system. We use 9 BTWORLD jobs which compute answers to a large number of questions related to the operation of BitTorrent (e.g., *How does a tracker evolve in time?*, *How many active hashes are in the system?*, *Which are the most popular hashes and swarms?*). BTWORLD jobs process a multi-column input dataset that contains timestamped, per-tracker statistics of the BitTorrent content: the number of users with fully and partially downloaded content, and the number of completed user downloads.

MapReduce workloads. In our experiments we will use three workloads that we create using the Wordcount, PiEstimator, Sort, Grep, and BTWORLD applications and that are differentiated by their variability of job sizes. Each of these workloads consists of a stream of 300 jobs with a Poisson arrival process. In our evaluation of TYREX we designed all three workloads to impose an average load of 70%.

The three workloads we create are called HVW, MVW, and LVW for High/Medium/Low Variability Workload, and they have, going from one to the next, decreasing squared CV values of their job size distributions. In Figure 5(a) we distinguish four ranges of job sizes to show the job size distributions in our workloads: "tiny" ($<$ 1 minute), "short" (1-10 minutes), "medium" (10-100 minutes), and "large" ($>$ 100 minutes). In all workloads there is a large fraction of tiny and short jobs combined (more than 60%). In Figure 5(b), we show the fractions of the total CPU time consumed by all jobs together with a cutoff point as defined in Section IV-D of 1, 10, and



(a) (b)

Fig. 5. Comparing the workloads: (a) the job size distributions, and (b) the sum of the CPU time of all jobs up to 1, 10, and 100 minutes as a fraction of the total CPU time.

100 minutes, respectively. For example, even for the HVW workload, if the timer of partition P_1 is set to 10 minutes, the fraction of the load processed by partition P_1 is only 10%. Our workloads are representative for a broad range of computer systems workloads, having squared CV values in the range of 4 to 20 – the standard TPC benchmarks for evaluating the performance of computer systems exhibit squared CV values in this range [26]. As heavy-tailed workloads fit recent measurement of MapReduce production clusters [20], the HVW and MVW workloads with high variability of job sizes are more interesting in our evaluation.

TABLE II. SUMMARY OF THE EXECUTION OF THE HVW, MVW, AND LVW WORKLOADS.

Statistics	HVW	MVW	LVW
Total jobs		300	
Squared CV	20	10	4
BTWORLD jobs	33	45	10
Total maps	6,139	11,866	30,576
Total reduces	788	1,368	3,089
Temporary data [GB]	573	693	1,062
Persistent data [GB]	100	92	303
Total CPU time [h]	63.6	124.6	306.9
Total runtime [h]	3.51	3.98	5.31

To understand in more detail the structure of our workloads, we show in Table II execution summaries when all jobs are executed sequentially in an empty 20-node Hadoop framework. HVW spans more than 7,000 (map and reduce) tasks in total and requires more than 60 h CPU time. MVW has more than 12,000 (map and reduce) tasks in total and requires twice as much CPU time. The former two workloads are comparable in the amount of persistent data generated and the total runtime in an empty system. LVW consists of more than 33,000 (map and reduce) tasks, requires five times more CPU time than HVW, and generates three times more persistent data than both HVW and MVW.

DAS-4 deployment. The DAS-4 [2] cluster we use in our experiments has 20 dual-quad-core compute nodes with 24 GiB memory per node and 50 TB total storage. The DAS-4 nodes are connected through 1 Gbit/s Ethernet (GbE) and 20 Gbit/s QDR InfiniBand (IB) networks. We use Hadoop-1.0.0 over InfiniBand and we configure at each node 6 map slots, 2 reduce slots, and 3 GiB memory per running task. The HDFS uses a virtual disk device with RAID-0 software and 2 physical devices with 2 TB storage in total per node.

Baseline policies. We contrast TYREX with two state-of-the-art scheduling algorithms in data-intensive frameworks – FIFO

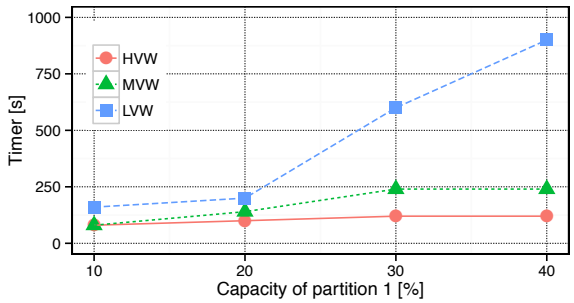


Fig. 6. The optimal timer of partition P_1 for each capacity between 10-40% and for each workload.

and FAIR. The former is the default scheduler in Hadoop and assigns both map and reduce tasks using the first-in-first-out scheduling discipline. The later is widely used in deployment and is a discrete version of the weighted processor-sharing discipline which allocates slots to jobs proportional to the number of their tasks.

VI. Experimental Evaluation

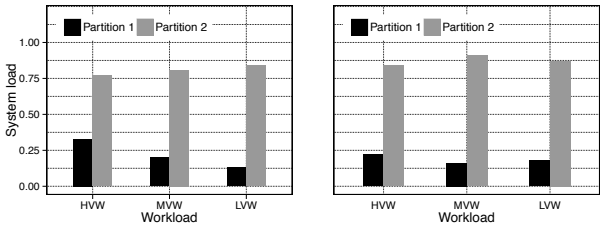
We evaluate TYREX using our prototype implementation in Hadoop on a 20-machine cluster. We investigate the setting of capacities and timers and the performance of the STATICTAGS policy in Sections VI-A and VI-B. Further, we analyze the way the dynamic timers adapt over time and the performance of the DYNAMICTAGS policy in Sections VI-C and VI-D. Finally, we compare TYREX with our two baselines, the FIFO and FAIR schedulers in Section VI-E.

A. Setting Capacities and Timers

One of the issues in analyzing and deploying TYREX with the STATICTAGS policy is setting the number of partitions and the values of the partition capacities and timers. As we argued in Section IV-D, having more than two partitions is not worthwhile, so we consider having only two partitions.

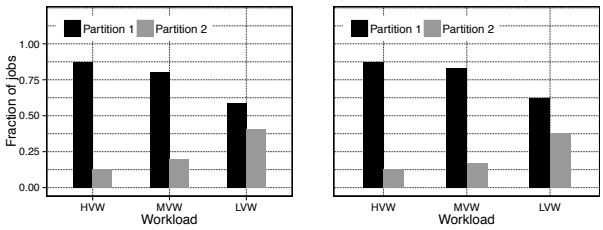
We will first investigate the relation between the partition capacity and the partition timer with the STATICTAGS policy for each workload. To this end, we seek to determine for each partition capacity its optimal timer, that is the timer that minimizes the overall job slowdown variability, as defined in Section II. Figure 6 shows the values of the optimal timer for a range of capacities of partition P_1. We see that the optimal timer of partition P_1 is very insensitive to the partition capacity when the job size variability is high. In particular, STATICTAGS has a low optimal timer (below 250 s) of partition P_1 for a relatively long range of partition capacities with both the HVW (squared CV of 20) and the MVW (squared CV of 10) workloads. However, the optimal timer of partition P_1 is considerably more sensitive to the partition capacity when the job size distribution is more balanced. Hence, the optimal timer of partition P_1 increases by a factor of 5 when the partition capacity increases from 10% to 40% for the LVW (squared CV of 4) workload.

Intuitively, TYREX aims to fit in partition P_1 the vast majority of tiny and short jobs, which implies that at most 10% of the total load in all workloads has to be processed in partition P_1 (see Figure 5(b)). As a consequence, more than 90% of the total system load, which at an imposed load of



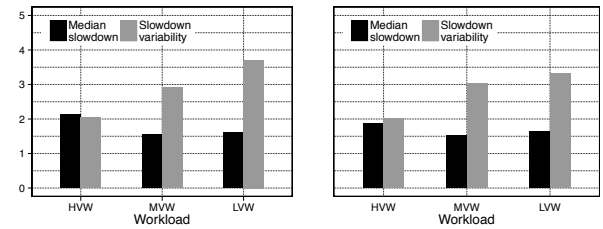
(a) $C_1 = 20\%$ (b) $C_1 = 30\%$

Fig. 7. The utilizations of partitions P_1 and P_2 versus the workload variability with STATICTAGS.



(a) $C_1 = 20\%$ (b) $C_1 = 30\%$

Fig. 8. The fractions of jobs completed in partition P_1 and P_2 versus the workload variability with STATICTAGS.



(a) $C_1 = 20\%$ (b) $C_1 = 30\%$

Fig. 9. The median job slowdown and the job slowdown variability versus the workload variability with STATICTAGS.

70% amounts to 63% of the system, has to be handled by partition P_2. Therefore, to keep the second partition stable we need to set the size of partition P_1 equal to at most 30% of the total system capacity. Indeed, it turns out that the STATICTAGS policy achieves the best job slowdown performance for all our workloads when the capacity of partition P_1 is set to 20% or 30%. Therefore, in the remainder of this section we report results only for these two partition sizes.

Next, we analyze how the STATICTAGS policy distributes the system load across its partitions. In Figure 7 we show the utilizations in partitions P_1 and P_2 for a capacity of partition P_1 of 20% and 30% with its timer set to the optimal value according to Figure 6. As we expected, STATICTAGS is rather aggressive in migrating jobs from partition P_1 to partition P_2 so that short jobs which occur in large fractions in our workloads, consistently receive service in partition P_1 under a relatively low load. In particular, we see that the utilization of partition P_1 is lower than 30% while the utilization of partition P_2 is higher than 75%. Interestingly, we see that the unbalance of the utilizations in partitions P_1 and P_2 is larger for MVW and LVW than for HVW. The reason for this result is that STATICTAGS shifts the job slowdown variability to partition P_2 so that the

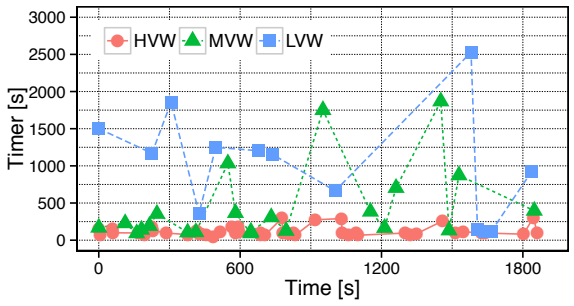


Fig. 10. The dynamic timer of partition P_1 with its capacity set to 30%.

vast majority of short jobs in our HVW workload are completed in partition P_1.

B. Performance of STATICTAGS

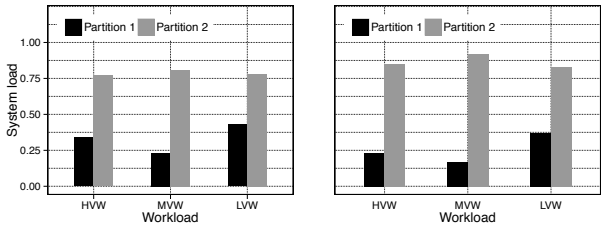
The main purpose of setting timers is to have a decent fraction of short jobs finished in the first partition and to avoid overload in any partition. In Figure 8 we show the fraction of jobs that are completed in each of the two system partitions when the capacity of partition P_1 is set to 20% and 30%. The timers are set to their optimal values depicted in Figure 6. As we expected, the fractions of jobs that are completely executed within partition P_1 is very high for both HVW and MVW (more than 80%). In contrast, the fractions of jobs that are finished in partitions P_1 and P_2 are more balanced for LVW (less than 60% are completed in partition P_1). For all workloads and more noticeable for LVW, the fraction of jobs finished in partition P_1 increases for larger partition sizes.

We show in Figure 9 the slowdown performance of the STATICTAGS policy having the capacities of partition P_1 of 20% and 30% with the timers set to their optimal values according to Figure 6. As a hint to reading this and later similar figures, the values at 20% capacity of partition P_1 should be interpreted as having a 95^{th} percentile of the job slowdown distribution of about 4.38 (2.05 x 2.14). We see that STATICTAGS has very good performance for the HVW workload, with both the job slowdown variability and the median job slowdown being less than 2. Moreover, we observe that the performance of STATICTAGS is relatively good for the MVW and LVW workloads, with the median job slowdown and the job slowdown variability being less than 2 and 3.3, respectively.

We find that STATICTAGS is not always effective in unbalancing the load across its partitions. Clearly, if STATICTAGS uses a low capacity for partition P_1 then short jobs may experience large job slowdowns because they always run under a relatively high load. We observe this phenomenon when the capacity of partition P_1 is set to 10% for the HVW workload. Similarly, we see that it is difficult to find a good timer when the capacity of partition P_1 is set to 40% because we are at risk at overloading partition P_2.

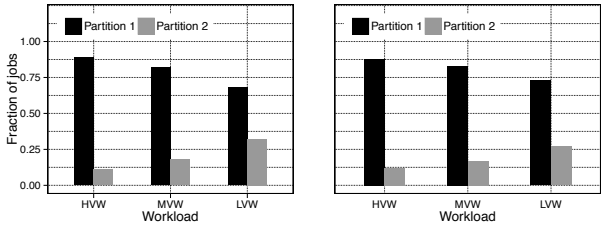
C. Evolution of Dynamic Timers

The distinguishing element of the DYNAMICTAGS policy is its operation without the burden of finding the optimal timers for given partition capacities. In this section we will investigate how the dynamic timers of DYNAMICTAGS adapt over time and their effect on the load across partitions.



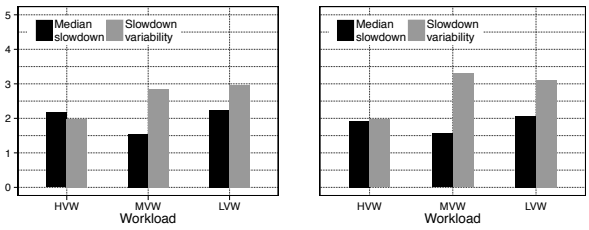
(a) $C_1 = 20\%$ (b) $C_1 = 30\%$

Fig. 11. The utilizations of partitions P_1 and P_2 versus the workload variability with DYNAMICTAGS.



(a) $C_1 = 20\%$ (b) $C_1 = 30\%$

Fig. 12. The fractions of jobs completed in partition P_1 and P_2 versus the workload variability with DYNAMICTAGS.

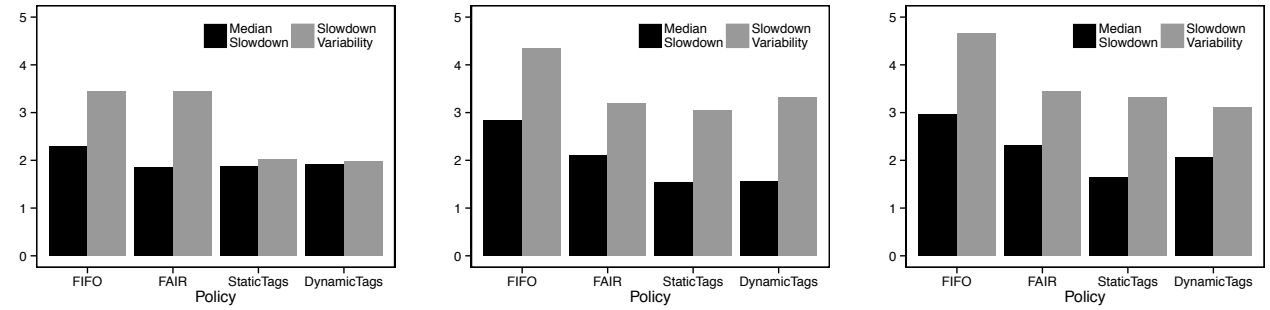


(a) $C_1 = 20\%$ (b) $C_1 = 30\%$

Fig. 13. The median job slowdown and the job slowdown variability versus the workload variability with DYNAMICTAGS.

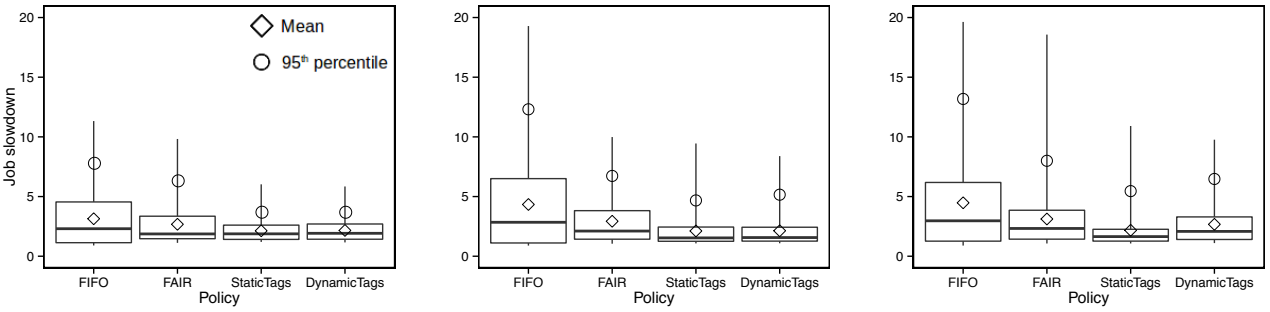
In Figure 10 we show for each workload how the value of the dynamic timer of partition P_1 changes during a 30-minute period when the capacity of that partition is set to 30%. Similarly to the STATICTAGS policy, we see that the timer converges to lower values when the job-size distribution is more variable. Indeed, for HVW the timer is always set to some value in the range between 50 and 300 s, for MVW most of the timer values are in the range between 100 and 500 s, and for LVW the timer varies between 500 and 2500 s. These increasingly wider and higher ranges nicely match the results in Figure 4.

In Figure 11 we show how DYNAMICTAGS distributes the imposed system load across its partitions for capacities of partition P_1 of 20% and 30%. We observe that DYNAMICTAGS assigns a significantly lower load to partition P_1 (less than 30% of the total load), which is exactly the same phenomenon as in the case of the STATICTAGS policy (see Figure 7). Apparently, DYNAMICTAGS is rather aggressive in migrating jobs from partition P_1 to partition P_2 so that short jobs, which occur in large fractions in our workloads, consistently receive service in partition P_1 under a relatively low load. We see that the partition utilizations with DYNAMICTAGS and STATICTAGS



(a) HVW (b) MVW (c) LVW

Fig. 14. The median job slowdown and the job slowdown variability using FIFO, FAIR, STATICTAGS, and DYNAMICTAGS policies for each workload.



(a) HVW (b) MVW (c) LVW

Fig. 15. The job slowdown distributions using FIFO, FAIR, STATICTAGS, and DYNAMICTAGS policies for each workload.

are very close for the HVW and MVW workloads. In contrast, DYNAMICTAGS has a higher load in partition P_1 for LVW because its job size variability is only 4. As this workload is more balanced, DYNAMICTAGS has to migrate fewer jobs from P_1 to P_2.

D. Performance of DYNAMICTAGS

DYNAMICTAGS adapts the timer of partition P_1 as the jobs in the queue make progress, rather than setting a fixed value which is used for all incoming jobs, as in STATICTAGS. Although DYNAMICTAGS operates fundamentally different than STATICTAGS, we show in this section that having timers of any kind (dynamic or fixed) has the same effect on the job slowdown variability.

In Figure 12 we show the fractions of jobs that are completed in each partition when the capacity of partition P_1 is set to 20% and 30%. We see that with the DYNAMICTAGS policy more than 80% of jobs in HVW and MVW are completed in partition P_1. Conversely, as the workload variability decreases, DYNAMICTAGS migrates more jobs to partition 2, so the fractions of jobs that are completed in different partitions are more balanced with LVW than with HVW and MVW. In particular, we observe that the fractions of jobs that are completed in partition P_1 are 10% higher with DYNAMICTAGS than with STATICTAGS for the LVW workload. This result shows that our DYNAMICTAGS policy is more conservative as it migrates jobs based on their relative progress (current partial job sizes), rather than using a fixed timer as in the STATICTAGS policy.

In Figure 13 we show the median job slowdown and the job slowdown variability when the capacity of partition P_1 is set to 20% and 30%. Two important things stand out. First, DYNAMICTAGS offers very low median job slowdown (below 2.2) and slowdown variability (below 3.2) for all our workloads, which are very close to the corresponding values with STATICTAGS shown in Figure 9. More importantly, Figure 13 also shows that the DYNAMICTAGS policy offers similar improvements for different capacities of partition P_1.

E. Improvements from TYREX

In this section we compare our TYREX scheduler with two baselines – FIFO, the default scheduler in Hadoop, and FAIR, the most popular MapReduce scheduler. For TYREX we set the capacity of partition P_1 to 30% and we use the optimal timer according to Figure 6 (only for STATICTAGS).

In Figure 14 we show the median job slowdown and the job slowdown variability for each policy with all our workloads. For the HVW workload, we find that TYREX with any of its two policies cuts in half the job slowdown variability while maintaining roughly the same median job slowdown when compared with both FIFO and FAIR. For the MVW and LVW workloads, when compared with FIFO, TYREX reduces the job slowdown variability and the median job slowdown by 30% and 50%, respectively. Somewhat surprisingly, in these cases the job slowdown variability is roughly equal with TYREX and FAIR, but TYREX with any of its two policies improves the median job slowdown by up to 30%.

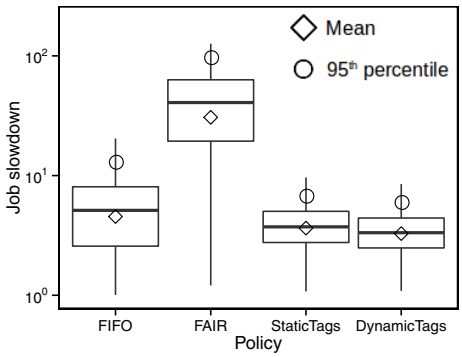


Fig. 16. The job slowdown distributions using FIFO, FAIR, STATICTAGS, and DYNAMICTAGS policies for the HVW workload under a system load of 0.9 (vertical axis in logscale).

In order to better understand the magnitude of the improvements we obtain with TYREX, we compare in Figure 15 the job slowdown distributions for each policy with all our workloads. Clearly, TYREX using any of the two policies is superior to both FIFO and FAIR as the distributions of the job slowdowns with TYREX have (much) smaller interquartile ranges and outliers than with the baselines, which was our main goal defined in Section II. In particular, for all workloads TYREX reduces the job slowdown at the 95^{th} percentile by 50-60% when compared with FIFO and by 20-40% when compared with FAIR.

To be realistic, so far we imposed a moderate to high system load of 70%. Finally, we design a stress test to evaluate the performance of TYREX under heavy loads. In Figure 16 we show the job slowdown distributions with all policies for the HVW workload under a system load of 90%. TYREX with any of the two policies is by far the best scheduler as the job slowdown distributions are very narrow and much lower than with the baselines. In particular, with TYREX, no job in the workload has a slowdown that is higher than 10. Surprisingly, we find that FAIR has very poor performance and cannot handle the workload when the system is heavy-loaded. Not only is the job slowdown distribution very wide and unbalanced, but also the makespan of the experiment is twice as long as with TYREX. The workload has a total submission schedule of 25 minutes and is completed by FAIR only 45 minutes after the arrival process ends. In contrast, with TYREX the workload is completed already 9 minutes after the arrival of the last job. Apparently, with FAIR large jobs may occupy underutilized reduce slots which results in very long waiting times for shorter jobs. These non-work-conserving effects in FAIR have a major impact on the stability of the system under high loads. In contrast, TYREX alleviates this issue by confining jobs to smaller partitions, so that no large job monopolizes the reduce slots in the system.

VII. RELATED WORK

The problem of job size variability in MapReduce clusters was first identified in [31], but since then there has been little work on the performance of size-based scheduling policies in data analytics frameworks. This work aims to advocate for the need of such policies for MapReduce workloads. To this end, we investigate the design and implementation of size-based scheduling policies in MapReduce-based systems.

During the past decade, performance of MapReduce became a rich exploration domain, leading to several papers focusing on diverse aspects of MapReduce scheduling: data locality [31], stragglers [5], [6], resource heterogeneity [33], or elastic scaling [12], [13], [21]. State-of-the art schedulers for MapReduce-based systems assume they have complete control over a fixed set of resources, thus they are typically deployed on dedicated clusters of machines. All three main schedulers incorporated in Hadoop (FIFO [1], FAIR [4], and CAPACITY [3]) fall into this category. Whereas FIFO executes jobs in order of their arrival with five priority levels using the full system capacity, both FAIR and CAPACITY divide the system capacity across a number of queues. FAIR uses a processor-sharing scheduling policy to divide the system processors across different (sets of) jobs and CAPACITY employs multiple statically configured queues in order to confine distinct users to single partitions. The main design motivation of the latter two was to prevent large jobs or heavy users from monopolizing the framework. However, these schedulers do not solve the problem of job slowdown variability. While FAIR hurts the overall cluster performance due to resource contention and thrashing, CAPACITY does not explicitly handle job sizes.

We have discussed throughout the paper several *size-based scheduling* policies which have a strong bias towards short jobs. Although these policies have been analyzed for distributed-server systems [14], [15], supercomputing workloads [25], and cloud compute-intensive workloads [10], a realistic investigation of such policies in datacenters for MapReduce workloads is currently missing. In particular, size-based scheduling has been employed in Hadoop [23] with adaptations of two policies: Shortest-Remaining-Processing-Time (SRPT) and Fair-Sojourn-Protocol (FSP). However, these approaches have rather limited applicability in large-scale datacenters as they require either accurate estimations of job sizes [22] or periodic simulations of queued jobs in a virtually fair system [11], [23]. The main idea behind FSP is to extend the SRPT policy with a job aging function which virtually decreases the sizes of the waiting jobs, thus avoiding starvation of the large jobs.

MapReduce workloads have a complex internal structure with intra-job data precedence constraints between the map and reduce phases which can run on any number of resources as long as the input data is accesible through a distributed filesystem. Therefore, in this paper, we adapt the structure of the TAGS policy to datacenter environments by having partition capacities in addition to the timers as parameters and we use the underlying distributed fileystem to migrate jobs in a work-conserving way instead of killing jobs when they are migrated across partitions.

Datacenter schedulers such as MESOS [19] or YARN [28] employ a two-level scheduling architecture by dynamically allocating resources to different specialized frameworks (e.g., Hadoop [1] or Spark [32]). MESOS employs fine-grained resource sharing across different frameworks and delegates the control of resources to the frameworks by initiating *resource offers*. Thus, the frameworks implement specific policies to decide which and how many resources to accept. In contrast, YARN takes a *request-based* approach and considers application-specific constraints (e.g., job size, hardware requirements, data locality) to allocate resources from a fixed set of machines to each application. As our work focuses on job scheduling in frameworks which have complete control over a fixed set of

resources, YARN can easily incorporate a scheduling system such as TYREX to schedule a more diverse array of applications beyond the traditional MapReduce.

VIII. Conclusions

Reducing job slowdown variability while keeping the median job slowdown relatively low is an attractive yet challenging target in MapReduce frameworks. In this paper we investigated a class of size-based scheduling policies that confine (sets of) jobs of similar sizes to single partitions of the MapReduce framework. We have first introduced a general model for exploring the job slowdown variability problem with heavy-tailed MapReduce workloads. Based on this model we have designed a scheduling system called TYREX that partitions the set of resources of the framework and isolates sets of jobs of very different sizes in those partitions. To do so, TYREX imposes runtime limits (partition timers) and successively executes parts of jobs in a work-conserving way in each partition (the STATICTAGS policy). On top of that, to remove the burden of finding the optimal timers, we develop a statistical model for migrating jobs in a dynamic way that achieves near-optimal job slowdown performance (the DYNAMICTAGS policy).

With a comprehensive set of experiments on a cluster system, we showed that TYREX achieves very balanced job slowdown distributions for a broad range of representative MapReduce workloads. In particular, TYREX cuts in half the job slowdown variability while preserving the median job slowdown for workloads with high variability of job sizes. Furthermore, unlike FIFO and FAIR, TYREX delivers good performance and balanced job slowdowns even in unfavorable conditions of heavy load. We conclude that TYREX outperforms state-of-the-art schedulers like FIFO and FAIR with respect to both job slowdown variability and median job slowdown for typical MapReduce workloads.

References

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

[2] "The Distributed ASCI Supercomputer 4," http://www.cs.vu.nl/das4.

[3] "Hadoop Capacity Scheduler," http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[4] "Hadoop Fair Scheduler," http://hadoop.apache.org/docs/r2.5.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[5] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "GRASS: Trimming Stragglers in Approximation Analytics," *NSDI*, 2014.

[6] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-Reduce Clusters using Mantri," *OSDI*, 2010.

[7] Y. Chen, S. Alspaugh, and R. Katz, "Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads," *VLDB*, 2012.

[8] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The Case for Evaluating MapReduce Performance using Workload Suites," *IEEE MASCOTS*, 2011.

[9] W. Cirne and F. Berman, "Adaptive Selection of Partition Size for Supercomputer Requests," *JSSPP*, 2000.

[10] L. Fei, B. Ghit, A. Iosup, and D. Epema, "KOALA-C: A Task Allocator for Integrated Multicluster and Multicloud Environments," *IEEE Cluster*, 2014.

[11] E. J. Friedman and S. G. Henderson, "Fairness and Efficiency in Web Server Protocols," *ACM SIGMETRICS*, vol. 31, no. 1, 2003.

[12] B. Ghit, N. Yigitbasi, A. Iosup, and D. Epema, "Balanced Resource Allocations Across Multiple Dynamic MapReduce Clusters," *ACM SIGMETRICS*, 2014.

[13] Í. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini, "GreenHadoop: Leveraging Green Energy in Data-Processing Frameworks," *EuroSys*, 2012.

[14] M. Harchol-Balter, "Task Assignment with Unknown Duration," *JACM*, vol. 49, no. 2, 2002.

[15] M. Harchol-Balter, M. E. Crovella, and C. D. Murta, "On Choosing a Task Assignment Policy for a Distributed Server System," *JPDC*, vol. 59, no. 2, 1999.

[16] M. Harchol-Balter, A. Scheller-Wolf, and A. R. Young, "Surprising Results on Task Assignment in Server Farms with High-Variability Workloads," *ACM SIGMETRICS*, 2009.

[17] M. Harchol-Balter, K. Sigman, and A. Wierman, "Asymptotic Convergence of Scheduling Policies with Respect to Slowdown," *Performance Evaluation*, vol. 49, no. 1, 2002.

[18] T. Hegeman, B. Ghit, M. Capota, J. Hidders, D. Epema, and A. Iosup, "The BTWorld Use Case for Big Data Analytics: Description, MapReduce Logical Workflow, and Empirical Evaluation," *IEEE Big Data*, 2013.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," *NSDI*, 2011.

[20] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An Analysis of Traces from a Production MapReduce Cluster," *CCGrid*, 2010.

[21] J. Leverich and C. Kozyrakis, "On the Energy (In) Efficiency of Hadoop Clusters," *SIGOPS*, vol. 44, no. 1, 2010.

[22] M. Lin, L. Zhang, A. Wierman, and J. Tan, "Joint Optimization of Overlapping Phases in MapReduce," *Performance Evaluation*, vol. 70, no. 10, 2013.

[23] M. Pastorelli, A. Barbuzzi, D. Carra, M. Dell'Amico, and P. Michiardi, "HFSP: Size-based Scheduling for Hadoop," *IEEE Big Data*, 2013.

[24] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," *SoCC*, 2012.

[25] B. Schroeder and M. Harchol-Balter, "Evaluation of Task Assignment Policies for Supercomputing Servers: The Case for Load Unbalancing and Fairness," *Cluster Computing*, vol. 7, no. 2, pp. 151–161, 2004.

[26] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman, "How to Determine a Good Multi-Programming Level for External Scheduling," *IEEE ICDE*, 2006.

[27] J. Tan, X. Meng, and L. Zhang, "Delay Tails in MapReduce Scheduling," *ACM SIGMETRICS*, 2012.

[28] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop Yarn: Yet Another Resource Negotiator," *SOCC*, 2013.

[29] A. Wierman and M. Harchol-Balter, "Classifying Scheduling Policies with Respect to Unfairness in an M/GI/1," *ACM SIGMETRICS*, 2003.

[30] A. Wierman, M. Harchol-Balter, and T. Osogami, "Nearly Insensitive Bounds on SMART Scheduling," *SIGMETRICS*, 2005.

[31] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," *EuroSys*, 2010.

[32] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," *NSDI*, 2012.

[33] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." *OSDI*, 2008.