POSUM: A Portfolio Scheduler for MapReduce Workloads

Maria A. Voinea
Technische Universiteit Delft
m.voinea@student.tudelft.nl

Alexandru Iosup
Vrije Universiteit Amsterdam
a.iosup@vu.nl

***Abstract*—MapReduce systems are widely used in today's data-driven society and there is a large body of work dedicated to scheduling such jobs in data centers. However, when considering compound performance objectives (such as reducing runtime and costs) it is difficult to design only one scheduler to achieve specific results. In contrast, we propose POSUM, a system capable of adapting to the current workload characteristics and target objectives by periodically evaluating a set of potential policies, and switching to "the best" one. The work is ongoing, but we have found that designing a system with such flexibility requires total decoupling from the MapReduce framework itself. A consistent repository of processing statistics should be accessible by all components involved in simulation and decision making, especially for application agnostic frameworks such as Hadoop YARN. Also, prediction should depend on the exact mapper and reducer class references as a more stable indicator of similarity, preferable to application or user name.**

Keywords*—*MapReduce, portfolio scheduling, data center, provisioning and allocation; scheduling policies, Hadoop YARN

I. Introduction

Although not a universal panaceum for big data problems, MapReduce systems are widely used in industry, governance, and academia [1]. Scheduling MapReduce workloads is thus important, and, as related work indicates [2] [3] [4], challenging. Various schedulers have already been proposed to address MapReduce stragglers [5], resource utilization [6], and job deadlines [7]. However, designing only one scheduler to optimize for multiple performance goals can be very difficult and error-prone, and ephemeral in that a change of goals can render the scheduler ineffective. In contrast, we propose a system that dynamically switches between a set of scheduling policies, to achieve the desired (and possibly changing) goals. This concept, of portfolio scheduling [8], [9], has never been used in big data settings, and in particular has never been used for MapReduce workloads.

Companies are relying increasingly on big data and business analytics to make their products and services customer-centric, improve operational performance, and even identify new business opportunities. Worldwide revenues could grow from nearly 122 billion dollars in 2015 to more than 187 billion in 2019, an increase of more than 50% [10]. Concurrently, governments and municipalities are also investing in big data research for improving public health and safety, and sustainability.

MapReduce is a prominent programming model devised for achieving abstraction and scalability in data processing. It involves running parallel tasks to process the input data in chunks. To allocate data center resources for these tasks and thus achieve the desired optimization goal, current MapReduce frameworks, such as Hadoop [1], select and then rely on a single scheduling policy. Corresponding to the diverse needs of current users, in practice, there exist many different scheduling policies, focusing on different operational aspects and/or optimizing for different goals. For example, many policies try to reduce the *runtime*, that is, the time elapsed between the submission of a job and its completion, across the jobs of all users running on a fixed-sized cluster. Others, when considering the financial cost of compute resources and their energy consumption, optimize for *resource utilization* and scale the cluster according to load. More complex usage patterns require their own scheduling policies, for example when a MapReduce application is part of a more elaborate business pipeline and, thus, must produce results by a certain deadline; this performance goal is commonly expressed as a *Service Level Objective (SLO)* and is common for latency-sensitive applications (e.g., personalized advertising and live business intelligence, span or fraud detection, and real-time event log analysis [4]). Similarly, scheduling policies may be conceived to work at different levels and have various effects depending on workload composition [11].

Designing just one scheduling policy to optimize for an objective that spans multiple desired effects can prove to be very difficult. Only a slight miscalculation or misinterpretation can have a great impact on the outcome. To address this problem, we design POrtfiolio SchedUler for Mapreduce (POSUM, pronounced "possum") as an alternative for tackling compound objectives for dynamic MapReduce workloads. At its core, POSUM relies on online simulation to evaluate, given the current state of the system, which out of a set of policies will perform the best, before switching to it for a given time period. This allows each policy to remain manageable in terms of complexity and hold true to its target use case, while still leaving room for the system to adapt at runtime and achieve the necessary performance objectives. In this work, our contribution is four-fold:

1) We design of a portfolio scheduler architecture for MapReduce systems (Section III).
2) We design of a simulator for online prediction of task behavior (Section IV).
3) We design a portfolio of scheduling policies to be used by the POSUM scheduler (Section V).
4) We design a set of experiments for evaluating the performance of the proposed scheduler VI.

[1] https://hadoop.apache.org/

II. Background

The MapReduce model involves breaking up an application's input into chunks and processing it in several phases. During the initial phase, called *mapping*, the *map* routine processes each chunk into a meaningful key-value mapping. The pairs are then sorted by key, during the *shuffle* phase. The final step is *reducing*: the *reduce* routine is called to aggregate the values of each key and output the final key-value results.

The developer is required to provide only the input location, along with the map and reduce routines. Once the application is sent for execution on a computer cluster, it becomes a *job* to be automatically managed by the chosen MapReduce framework. As shown in Figure 1, the job is added to a queue where it waits to be *scheduled*, i.e, be allocated the resources (RAM memory and CPU) required to run. If the pool of resource requests (computation resources required by each task to execute) exceeds the cluster's capacity, the scheduling policy determines the order in which each of the requests are to be executed for achieving the desired system objectives.

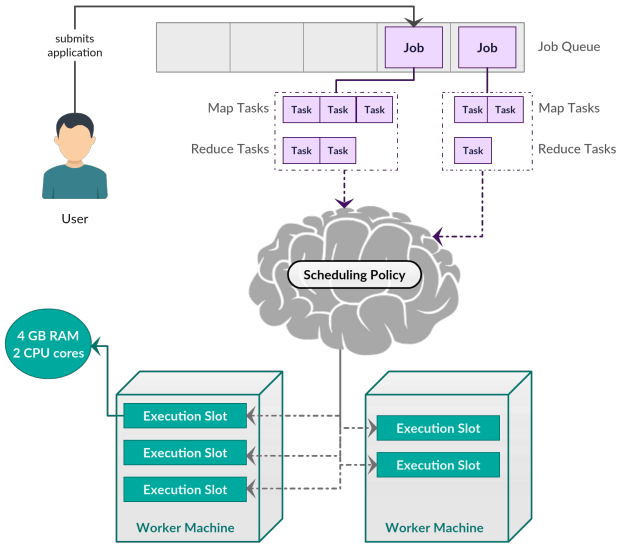

Fig. 1: MapReduce scheduling.

A core characteristic of MapReduce processing is the focus on data: its location, how long it takes to transfer it to computation nodes, the overhead of writing and reading it from disk, etc. The local storage of nodes in a MapReduce cluster forms a *distributed file system*. This structure allows for large amounts of data to be stored fault tolerantly and to be readily available for processing, even under scarce network bandwidth. This is the concept of *data locality*: scheduling a task on or close (on the same rack) to the node that holds its input data so as to lose less time on data transfer.

Job scheduling is a well-developed subject in the data center context. For MapReduce specifically, multiple scheduling policies have been devised to prioritize and optimize job execution, considering the particular characteristics of these workloads. The first MapReduce schedulers operated on *First-In-First-Out (FIFO)* order, meaning that the jobs were scheduled in the order of their arrival. Later, *Fair Scheduling* [12] was introduced for using max-min fairness to share resources between *pools* of jobs/ users.

Within the same job, tasks have always had higher priority on nodes holding their data. However, *delay scheduling* approaches [13] [2], relax inter-job priority even more to achieve better data locality. Cheriere et al. [14] argue that considering data locality when choosing which job should run on a given free slot leads to long wait times for small short jobs. Their *Shortest Remaining Time First (SRTF)* scheduling policies give higher priority to such jobs. Nguyen et al. [3] use a compound metric with variable coefficients to control the extent to which these short jobs are favored.

Meeting specific Service Level Objectives (SLOs) for jobs is another direction of optimization for MapReduce schedulers. Kc and Anyanwu [15], Polo et al. [16], Dong et al. [17], and Verma et al. [7] use historical information to predict future resource needs for achieving the required deadlines. Lim et al. [4] attempt to address the issue using offline constraint programming (CP). Other directions of research include mitigating stragglers (hanging tasks that need to be restarted), dynamic voltage scaling, virtualization, etc.

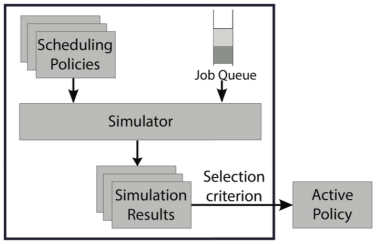

Fig. 2: The periodic portfolio scheduler in Deng et al. [8]

Our approach for achieving compound objectives under variable workloads relies on portfolio scheduling. The technique was borrowed from economics and first introduced to the field by Deng et al. [8]. As seen in Figure 2, the scheduler contains a portfolio of policies which it evaluates periodically using a simulator. The simulator predicts the behavior of each policy under the current system load, given the queued jobs. It returns a score as a combination of performance metrics. The scheduler chooses the policy with the highest score to switch to for the duration of the next period. The same concept has been successfully applied in previous studies to schedule scientific workloads on a compute cluster [18] and long-running virtual machines in data centers [9].

III. Design of POSUM

The design process of the portfolio scheduler started out with several goals in mind: include necessary conceptual elements of portfolio scheduling found in previous work (1); take into account the characteristics of MapReduce processing (2); adapt the resulting scheduler architecture to an actual, widely used MapReduce framework (3); keep the design flexible so as to explore and compare different approaches(4); follow or compare to state-of-the-art techniques wherever possible.

Addressing these goals, we design POSUM, an online meta-scheduler that can switch scheduling policies at runtime, based on real-time policy performance evaluations. POSUM is meant to be self-contained and exist alongside a MapReduce cluster, only interacting with it when gathering runtime

information and for switching scheduling policies. This makes it possible to easily integrate it with any MapReduce runtime framework. To test compliance with the *third design goal*, our implementation was integrated with the Hadoop (YARN) stack.

The system follows the high-level architecture illustrated in Figure 3. The system is comprised of three main processes which interact. The Data Master is responsible for monitoring the system and providing a coherent view of the stored statistics to the other processes. Two monitors operate on a configurable heartbeat. The Cluster Monitor gets real-time information about applications and tasks that are running from the MapReduce framework. The POSUM Monitor, on the other hand, gathers and interprets data on the operation of POSUM itself: simulation durations, the discrepancy between simulation scores and actual policy performance, etc.

The information gathered by the Data Master is used most intensively by The Simulator Master, a loosely coupled component that can simulate the outcome of a scheduling policy, given a certain queue composition, cluster state, and previous runtime statistics (see Section IV).

All decisions regarding POSUM's operations are made by the Orchestrator. It triggers or stops policy scoring simulations, it handles the application of policies, and makes decisions based on the feedback gathered by the system monitor. It is also capable of applying provisioning decisions, by reconfiguring the cluster to make use of or free recently added nodes.

However, actual policy application is done via the Portfolio Meta Scheduler, which is not an independent process, but a placeholder that extends the standard resource scheduler interface of the target framework. It delegates all its public and protected methods to the current policy that is being applied. It keeps evidence of the available scheduling policies and uses the logic of the currently plugged-in policy to reach each scheduling decision. This abstraction ensures that the transition between policies is seamless and does not disrupt the framework's operation. We have found that this component also needs access to the statistics on application progress, when policies need more information about the running jobs to make their decisions.

Keeping the architecture modular achieves separation of concerns and enables both flexibility in approach exploration (demanded by the *fourth goal*) and runtime performance tweaking. Each process can be deployed to a separate machine, can have different JVM characteristics, and can be restarted independently on failure.

IV. Design of the Core Simulator

POSUM uses a discrete-event simulator which mimics the same message-based event handling mechanism that enables MapReduce frameworks, like Hadoop, to operate. Resources are not modeled explicitly so as to reduce simulation time as much as possible (since the decisions need to be real-time). For further simplification, failure events are not considered in this version of POSUM. It hypothesizing about the behavior of jobs as they come into the system, by looking at their configuration, their current behavior (if they are already running), and historical data gathered from jobs that have already run on the system, that may or may not have had similar characteristics.

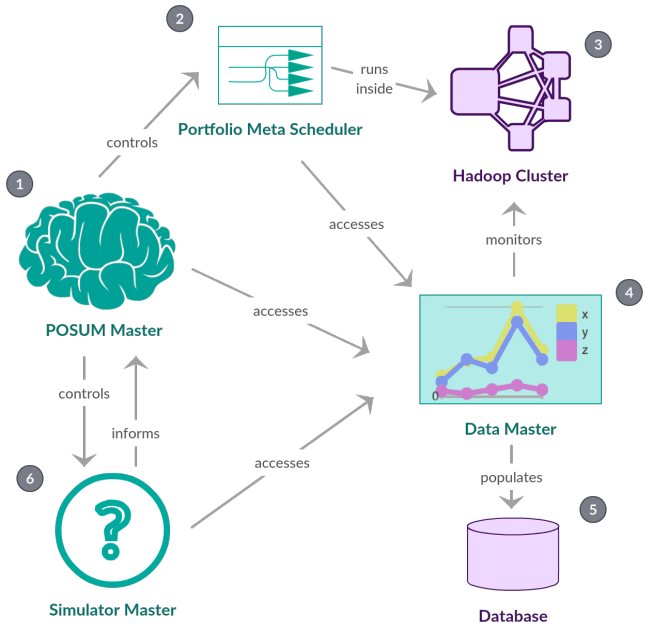

Fig. 3: POSUM overview.

The heart of the simulator is a component responsible for predicting how long each task will take. Several techniques from existing work have been combined into a set of three predictors of increasing complexity.The **Basic Predictor** calculates task runtimes as an average of the runtimes of tasks of the same type that ran on the cluster. They can be from the current job, if the job already has completed tasks, or calculated from historical data by looking at *similar* jobs that were run on the cluster. Jobs are considered "similar" if they have been submitted by the same user. The maximum number of past jobs that are used as reference is a configurable parameter in the system. The result is comparable what Polo et al. [16] use.

The **Standard Predictor** augments the prediction by considering task durations dependent on the data they have to process. Thus, the *average processing rate (APR* is calculated from past tasks, and, in the case of map tasks, also their *selectivity*. The result is close to what Kc et al. [15] describe, but without differentiating between the shuffle and reduce phases of reduce tasks. The following equation applies (see Table I for the notations):

$$r_i^M = d_i^M \cdot \rho^M \quad , \quad r^R = f \cdot d^M \cdot \rho^R \tag{1}$$

Moreover, the system considers the map and reduce task history separately: since mapper and reducer classes are listed in the job configuration, the predictor looks at past jobs that have used the map class when computing the map average, and past jobs with the same reducer for the reduce average. This is much more reliable than the application name, between subsequent runs. If there is no exact history, the latest jobs of the user are used for both map and reduce calculation. If no history is available at all, but some map tasks have already completed, the processing rate of reduces is considered equal to the map processing rate.

Symbol	Description
d^M	size of the data that each map task has to process
f	map selectivity
$\rho^M = \sum_{i \in C_M} \frac{r_i^M}{d_i}$	APR of completed maps
$\rho^R = \sum_{i \in C_R} \frac{r_i^R}{d_i}$	APR of completed reduces
r_i^M	runtime of task i of type map
r_i^R	runtime of task i of type reduce
d_i	input data of task i
C_M	set of similar completed map tasks
C_R	set of similar completed reduce tasks
ρ^{M_L}	APR of similar local maps
ρ^{M_R}	APR of similar remote maps
$\rho^{R_s,T}$	APR of similar typical reduces
ρ^{R_m}	APR of the merge step of a similar reduce task
ρ^{R_r}	APR of the reduce step of a similar reduces
$t^{R_s,1}$	average shuffle time for similar first wave reduces
$V_D = \frac{\sum_{i \in D} (max(0, r_i - d_i))^2}{\lvert D \rvert}$	cumulated deadline violations of DC jobs
$S_B = \sum_{i \in B} \frac{r_i}{max(e_i, MinE)}$	cumulated bounded slowdown of batch jobs
$C = \sum_{n \in N} u_n$	
D	set of DC jobs in workload
B	set of batch jobs in workload
r_i	runtime of job i
e_i	total execution time of the tasks of job i
$MinE$	lower bound on the execution time of jobs
d_i	desired runtime of job i
u_n	uptime (in hours) of node n
N	set of nodes in the cluster (maximum size)

TABLE I: Notations used in Equations 1 - 5

.

The **Detailed Predictor** goes even deeper into the task internals and constructs a profile for each job that passes through the system. This predictor takes into account the fact that map tasks running on nodes that do not have their data locally may run for longer than tasks that do. As such, when computing the average map process rate, either from historical or current data, two values are kept, one for local and one for remote tasks. Also, the reduce task duration is split up into its three constituent phases: the shuffle phase (when input records are copied from the mappers), the merge phase (when these are merged into the final reduce input), and the reduce phase (when the reduce algorithm is applied to the constructed input). Processing rates for all task phases are kept separately. Moreover, the profile also includes first wave shuffle times, which are fixed time values that correspond to the part of the first shuffle phase that does not overlap with the map phase. The resulting reduce estimation is similar to the one from Verma et al.[7], with the modification that merge time is considered and no regression is used (see Table I for the notations):

$$r_i^M = \begin{cases} d^M \cdot \rho^{M_L}, \text{ for local map i} \\ d^M \cdot \rho^{M_R}, \text{ for remote map i} \end{cases}, \quad (2)$$

$$r_i^R = \begin{cases} f \cdot d^M \cdot (\rho^{R_s,T} + \rho^{R_m} + \rho^{R_r}), \text{ for typical reduce i} \\ t^{R_s,1} + f \cdot d^M \cdot (\rho^{R_m} + \rho^{R_r}), \text{ for first reduce i} \end{cases}. \quad (3)$$

V. Design of the Portfolio of Scheduling Policies

The policies equipped in the portfolio should be representative and capable of performing on different workload patterns and application types. However, their number should be relatively restricted, so as to minimize the exploration step during the selection and application phases. Seeing as this is the first exploratory study of portfolio scheduling on MapReduce, and the operational model is not confined to a particular domain, the set of policies has been composed manually, keeping the three performance dimensions in mind: respecting deadlines, lowering batch job runtime and minimizing resource (node) uptime (see Section VI for the system model details).

Thus, each scheduling policy is composed of two sub-policies: the first controls task prioritization and slot allocation, while the latter handles node provisioning (scaling of the cluster). Table II contains a short description of each of them.

The allocation policies are inspired by a short survey of the field. For deadline constrained (DC) jobs, the baseline policy is usually Earliest-Deadline-First (EDF) in literature [15] [16] [17]. Following the same reasoning for the second performance dimension, jobs with larger slowdown should have higher priority. Thus, for batch jobs (BC), the Largest-Slowdown-First (LSF) heuristic is a good candidate. However, a form of prioritisation needs to be established between the two categories as well.

Two solutions were adopted for this, resulting in the pair of policies *EDLS-Sh* and *EDLS-Pr*.The δ parameter is configurable in the system (between 0 and 1) and represents the importance of DC jobs for the cluster owner. The two policies optimize by tailoring to the types of jobs in the workload. It is to be noted that although they derive conceptually from general cluster scheduling techniques, it is not entire jobs that are scheduled using the heuristics, but their constituent tasks. This results in constant reshuffling of the jobs in the priority queues even while they are running, a characteristic of MapReduce processing.

Another pair of policies was added to optimize for the size of the jobs in the workload: *hSRTF* and *LOCF*. The former should give preference to jobs that have smaller input and shorter execution times, while the latter favors the ones with large input sizes. This happens because a larger input would be distributed on a larger portion of the cluster, increasing the probability of at least one task being local on the target node (at least in the beginning of the job's execution).

The resulting set of four policies thus achieves a balance between both types of priority enforcement, while tailoring to

specific workload characteristics. Moreover, they enrich traditional job scheduling approaches with MapReduce-specific dimensions like task performance and data locality (in line with the *second design goal*).

For provisioning, the different policies choose at what rate to expand or shrink the cluster in order to scale up when the risk of deadline violations is high, and scale down when the cost of leasing nodes outweighs the benefits.

The two types of policies are always used in combination (a total of 16 posibilities) and the full spectrum is explored on each simulation. Once a decision is made, the sub-policy of the first type is plugged into the system as the main scheduler. The resize policy is used by the POSUM Master to reconfigure the cluster.

VI. Experimental Evaluation

We are still in the early stages of designing the experiments for evaluating the performance of POSUM. There are three directions that we are following:

1) evaluation of prediction accuracy: a set of experiments for comparing the capabilities of the three predictors with respect to results reported in the literature that inspired them;
2) evaluation of the performance of POSUM itself: a series of experiments for comparing POSUM's dynamic policy switching with the results obtained by running each constituent policy separately on the same workload;
3) a sensitivity analysis: runs with different configurations of parameters regarding prediction (i.e. algorithm and default runtimes), operation (i.e the δ factor), and the compound objective (Equation 5).

POSUM was implemented for and integrated into the Hadoop (YARN) 2.7.1 stack through a GitHub repository fork. All the components are started as separate processes that communicate via the same RPC mechanism that Hadoop uses internally. All experiments are run on the TU Delft site of the DAS-5 [2]. The cluster contains identical nodes running CentOS Linux. Each machine has a Dual 8-core processor at 2.4 GHz, 64 GB RAM, and two 4TB HDDs. The nodes are connected by both standard 1 Gbit/s Ethernet, and FDR InfiniBand.

As production trace data is often lacking in real-time system information (node topology, data placement, bandwidth, disk speed, failures), the main experiments are run with the micro-benchmarking tool called BigDataBench [3][19] (inspired by HiBench). The workloads will be synthetic, created using a fixed set of benchmarking algorithms. The input data for each job will be generated, with sizes drawn from an exponential distribution of predominantly small values, as per the findings by Chen et al. [11]). Both uniform and bursty arrival patterns will be explored for their effect on the performance. In the case of the real-time applications, deadlines are generated using techniques from literature [16] [4], i.e. the time it takes to run an application with the same data size alone on the cluster, multiplied by a relaxation factor.

Throughout the experiments, the performance objective comprises three aspects: the accumulated SLO violation penalties, total batch job slowdown, and the charged cost for leasing the cluster nodes by the hour. Since the three metrics operate in different value ranges and are difficult to normalize, the aggregated performance score is not a scalar value, but a three-dimensional vector:

$$P = (V_D, S_B, C) \tag{4}$$

Thus, any compound metric can be expressed with the same notation and plugged into the system. All that is needed is an appropriate function to compare two performance scores. The formula we use is:

$$F_c(P_1(V_{D_1}, S_{B_1}, C_1), P_2(V_{D_2}, S_{B_2}, C_2)) = \alpha\,|V_{D_1} - V_{D_2}| + \beta\,|S_{B_1} - S_{B_2}| + \gamma\,|C_1 - C_2| \tag{5}$$

, where α, β, and γ are normalization factors to compensate both the value range differences, and the relative importance of each metric to our hypothetical data center customer.

VII. Related Work

While the concept of portfolio scheduling has been used before, previous work is not directly compatibile with the described MapReduce cluster. The approach of van Beek et al. [9] took into consideration only the provisioning of long-running VMs, not the scheduling of individual jobs. Moreover, the resource usage and behavior of VMs in time is considered to be previously known, which is not realistic for real-time MapReduce clusters. Closer to the current model is the work of Deng et al. [18]. However, it differs in both workload type and operation. MapReduce workloads are generally data-intensive, as opposed to scientific computing, which are generally more concerned with CPU-RAM interaction. Furthermore, where in the model of Deng et al., jobs were considered independent and were assigned VMs from the pool, the current model divides each job into several tasks that have dependencies and communication needs between them.

This work also draws inspiration from previous research in the design of the simulator. However, none of the existing solutions fully match our simulation requirements. SLS [4], MRPerf [20] and MRSim [21] have very low-level resource models that result in heavy time costs, while MRSG [22] and YARNsim [23] are designed to run a specific job based on a manually-constructed behavior configuration, and not an entire workload. Mumak, SimMR and Starfish are not compatible with the YARN architecture and require upgrading, but Mumak is available for source modification. However, Mumak is not capable of running a new workload: it only replays previous traces with a given scheduler. In conclusion, we have designed and implemented a new MapReduce simulator for use with the portfolio scheduler, based on techniques and results of previous work.

[2] http://www.cs.vu.nl/das5/

[3] http://prof.ict.ac.cn/BigDataBench/

[4] https://hadoop.apache.org/docs/r2.4.1/hadoop-sls/SchedulerLoadSimulator.html

Name	Type	Description
EDLS-Sh	allocation	a share-based scheduler in which DC jobs are given a share of the cluster equal to δ and are ordered by EDF, while BC jobs get $(1-\delta)$ of the cluster and are ordered by LSF
EDLS-Pr	allocation	an order-based scheduler in which two queues are kept (one with DC jobs in EDF order and the other in with BC jobs in LSF order), and on each scheduling decision, the DC queue has δ chance of getting picked and the other has $(1-\delta)$
hSRTF	allocation	the share-based version of the Shortest-Remaining-Time-First scheduler described by Cheriere et al. [14]
LOCF	allocation	a FIFO scheduler that enforces locality along the lines of what Zaharia et al. [13] and He et al. [2] implemented
+X	provisioning	increase size of the cluster by a number of x nodes
-X	provisioning	increase size of the cluster by a number of x nodes
DLX	provisioning	resize to as many nodes as are needed to meet all the deadlines
MaxBSD	provisioning	resize to as many nodes as are needed to not exceed a configured maximum

TABLE II: A description of the policies used by POSUM.

With regard to the body of existing work for scheduling MapReduce applications (briefly described in Section II), the focus of this work is not devising a single new and effective scheduling policy, but rather adapting existing solutions to work in a complementary fashion so as to cater to different workload compositions and arrival patterns.

VIII. Conclusion and Ongoing Work

Designing one single scheduling policy to achieve compound performance goals is complex and risky. Our system uses the advantages of portfolio scheduling to cater to this use case. We have found that it is best to decouple such a system from the MapReduce framework itself and keep the architecture modular. Prediction and even some scheduling decisions rely heavily on past and current processing statistics, meaning that these must be available at all times and in a consistent fashion. Also, current simulators are not equipped for online prediction of task behavior on the newer Hadoop stack. We, thus, offer our own implementation of one.

We have yet to carry out the necessary experiments for evaluating the system and comparing it to existing solutions. After each series of experiments, additional work will be done for integrating feedback and adjusting configuration to better outline the benefits and limitations of using this approach.

References

[1] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas, "Nobody ever got fired for using hadoop on a cluster," in *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, ser. HotCDP '12. New York, NY, USA: ACM, 2012, pp. 2:1–2:5. [Online]. Available: http://doi.acm.org/10.1145/2169090.2169092

[2] C. He, Y. Lu, and D. Swanson, "Matchmaking: A new mapreduce scheduling technique," in *CloudCom*, 2011, pp. 40–47.

[3] P. Nguyen, T. A. Simon, M. Halem, D. Chapman, and Q. Le, "A hybrid scheduling algorithm for data intensive workloads in a mapreduce environment," in *UCC*. IEEE Computer Society, 2012, pp. 161–167.

[4] N. Lim, S. Majumdar, and P. Ashwood-Smith, "A constraint programming based hadoop scheduler for handling mapreduce jobs with deadlines on clouds," in *ICPE*, 2015, pp. 111–122.

[5] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *OSDI*, 2010, pp. 265–278.

[6] N. Yigitbasi, K. Datta, N. Jain, and T. Willke, "Energy efficient scheduling of mapreduce workloads on heterogeneous clusters," in *GCM*. ACM, 2011, p. 1.

[7] A. Verma, L. Cherkasova, and R. H. Campbell, "Resource provisioning framework for mapreduce jobs with performance goals," in *Middleware*, 2011, pp. 165–186.

[8] K. Deng, R. Verboon, K. Ren, and A. Iosup, "A periodic portfolio scheduler for scientific computing in the data center," in *JSSPP*, 2013, pp. 156–176.

[9] V. van Beek, J. Donkervliet, T. Hegeman, S. Hugtenburg, and A. Iosup, "Self-expressive management of business-critical workloads in virtualized datacenters," *IEEE Computer*, vol. 48, no. 7, pp. 46–54, 2015.

[10] L. Columbus, "Roundup of analytics, big data & bi forecasts and market estimates, 2016," http://www.forbes.com/sites/louiscolumbus/2016/08/20/roundup-of-analytics-big-data-bi-forecasts-and-market-estimates-2016.

[11] Y. Chen, A. Ganapathi, R. Griffith, and R. H. Katz, "The case for evaluating mapreduce performance using workload suites," in *MASCOTS*, 2011, pp. 390–399.

[12] Y. Tao, Q. Zhang, L. Shi, and P. Chen, "Job scheduling optimization for multi-user mapreduce clusters," in *PAAP*, 2011, pp. 213–217.

[13] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys*, 2010, pp. 265–278.

[14] N. Cheriere, P. Donat-Bouillud, S. Ibrahim, and M. Simonin, "On the usability of shortest remaining time first policy in shared hadoop clusters," in *SAC*, S. Ossowski, Ed. ACM, 2016, pp. 426–431.

[15] K. Kc and K. Anyanwu, "Scheduling hadoop jobs to meet deadlines," in *CloudCom*, 2010, pp. 388–392.

[16] J. Polo, D. Carrera, Y. Becerra, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for mapreduce environments," in *NOMS*, 2010, pp. 373–380.

[17] X. Dong, Y. Wang, and H. Liao, "Scheduling mixed real-time and non-real-time applications in mapreduce environment," in *ICPADS*, 2011, pp. 9–16.

[18] K. Deng, J. Song, K. Ren, and A. Iosup, "Exploring portfolio scheduling for long-term execution of scientific workloads in iaas clouds," in *SC*, 2013, pp. 55:1–55:12.

[19] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *HPCA*, 2014, pp. 488–499.

[20] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "A simulation approach to evaluating design decisions in mapreduce setups," in *MASCOTS*. IEEE Computer Society, 2009, pp. 1–11.

[21] S. Hammoud, M. Li, Y. Liu, N. K. Alham, and Z. Liu, "Mrsim: A discrete event based mapreduce simulator," in *FSKD*, 2010, pp. 2993–2997.

[22] W. Kolberg, P. de B. Marcos, J. C. S. dos Anjos, A. K. S. Miyazaki, C. F. R. Geyer, and L. Arantes, "MRSG - A mapreduce simulator over simgrid," *Parallel Computing*, vol. 39, no. 4-5, pp. 233–244, 2013.

[23] N. Liu, X. Yang, X. Sun, J. Jenkins, and R. B. Ross, "Yarnsim: Simulating hadoop YARN," in *CCGrid*, 2015, pp. 637–646.