POSUM

A Generic Portfolio Scheduler for MapReduce Workloads

Maria A. Voinea

Technische Universiteit Delft

TU Delft

POSUM

A Generic Portfolio Scheduler for MapReduce Workloads

by

Maria A. Voinea

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday August 1, 2018 at 10:00 AM.

Student number:	4317602	
Project duration:	February 1, 2015 – August 31, 2018	
Thesis committee:	Prof. dr. ir. A. Iosup,	TU Delft/ VU Amsterdam, supervisor 1
	Prof. dr. ir. D.H.J. Epema,	TU Delft
	Dr. ir. A. Bozzon,	TU Delft
	Dr. ir. A. Uta,	VU Amsterdam, supervisor 2

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

The completion of this thesis has been one of the most challenging endeavours of my life. But also a uniquely educational and thoroughly gratifying experience that I will always cherish. Throughout the years of planing, implementation and assessment, several people have kept me focused and motivated and I would like to thank them specifically.

First of all, I would like to thank my supervisor, Prof. Alexandru Iosup, for giving me the opportunity to work with him and his exceptional team of students and researchers. You are an extraordinary educator and a constant source of inspiration to those around you. Thank you for your patience and guidance on this journey.

Secondly, a big thank you goes to the members of the @Large Research team, Otto, Ernst, Bogdan, Jesse, Tim, Laurens, and all the rest of you, for your valuable input and infectious enthusiasm. Over the years, you have always been there for meaningful discussions... and the occasional after-hours *borrel*. I am especially grateful for your help, Alex U., on the research paper submission. Debugging SVGs will not be the same without you.

Finally, I would like to give a resonating thank you to my adoring family and wonderful friends for supporting and encouraging me through this process. A special mention goes to my boyfriend, Cristi, for giving precedence to my work above all else and cheering me on, from start to finish.

Maria A. Voinea
Delft, August 2018

Contents

1 Introduction 1
1.1 Problem Statement . . . 2
1.1.1 Scheduling MapReduce Applications . . . 2
1.1.2 Compound Objectives . . . 2
1.1.3 Portfolio Scheduling . . . 3
1.1.4 Applicability of Previous Portfolio Schedulers . . . 4
1.2 Main Research Questions and Approach . . . 4
1.3 Main Contributions . . . 5
1.4 Thesis Structure . . . 5

2 Context 7
2.1 System Model for MapReduce Processing . . . 7
2.2 Primer on MapReduce and YARN Internals . . . 8
2.3 Survey of the State of the Art . . . 9
2.3.1 MapReduce Scheduling . . . 9
2.3.2 Simulator Types and Capabilities . . . 12
2.3.3 MapReduce Task Runtime Prediction . . . 16

3 Design of a Generic Simulator for Data Center-based MapReduce 19
3.1 Requirements Analysis . . . 19
3.2 Design of a Family of Task Behavior Predictors for Data Center-based MapReduce . . . 20
3.3 Simulator Design . . . 22
3.4 Adapting the MapReduce Simulator to the Hadoop YARN Stack . . . 22

4 Design of a Generic Portfolio Scheduler for Data Center-based MapReduce 25
4.1 Requirements Analysis . . . 25
4.2 Architectural Overview of POSUM . . . 25
4.3 Portfolio Scheduler Adapted to MapReduce Workloads . . . 26
4.3.1 Portfolio Creation . . . 27
4.3.2 Policy Selection . . . 27
4.3.3 Policy Application . . . 28
4.3.4 Reflection . . . 28
4.4 Key Processes and Components in POSUM . . . 29
4.5 The Administrator Dashboard . . . 33

5 Experimental Evaluation of the Portfolio Scheduler 35
5.1 Requirements Analysis . . . 35
5.2 Experimental Setup . . . 36
5.2.1 Hardware and Software Environment . . . 36
5.2.2 Workload Generation . . . 36
5.2.3 Performance Objectives . . . 37
5.3 Implementation of the POSUM Prototype . . . 38
5.3.1 Processes and Communication . . . 38
5.3.2 Policies. . . . 38
5.3.3 Simulator . . . 40
5.3.4 Monitoring. . . . 40
5.4 Experimental Results . . . 41
5.4.1 Prediction Accuracy . . . 41
5.4.2 Simulator Validation . . . 44
5.4.3 System Performance . . . 44
5.4.4 System Stability . . . 48

6 Conclusion and Future Work **51**
6.1 Conclusion on Main Research Questions 51
6.2 Lessons Learned 52
6.3 Recommendations for Future Research 52

A Glossary **53**

B POSUM Admin **55**

Bibliography **61**

1

Introduction

In recent years, more companies are turning to cloud solutions for hosting their business applications, as data centers allow for scalability, easier administration and better cost efficiency. Several frameworks have arisen as a result of the demand for abstraction of physical resources when running computation workloads or even full virtual environments in a data center.

One of the most prominent programming models devised for achieving abstraction and scalability in data processing is called MapReduce. It involves breaking up an application's input into chunks and processing it in several phases. Figure 1.1 illustrates the model. 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, while the model's implementation (the framework) does the rest automatically.

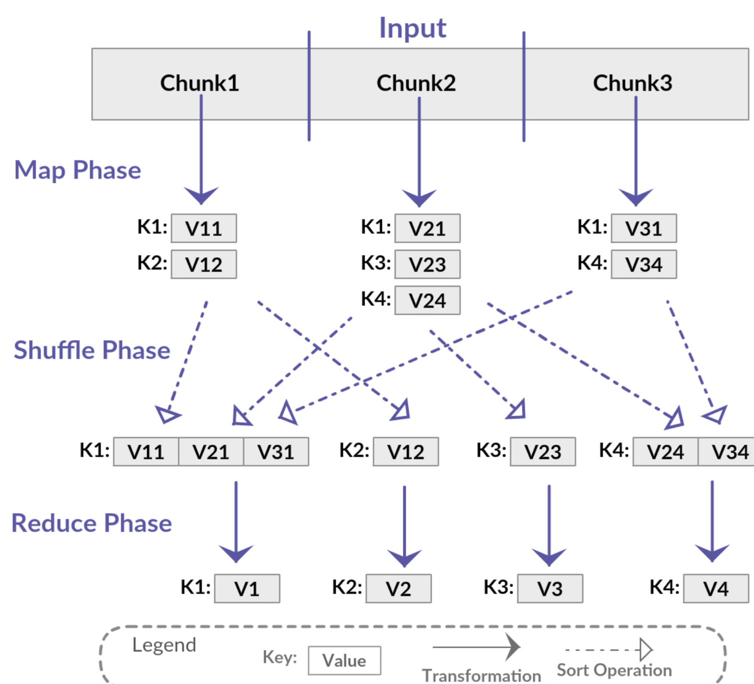


Figure 1.1: MapReduce programming model.

It was with the emergence of functional languages such as Lisp that map and reduce transformations were introduced in computer programming. Google researchers designed the MapReduce model using these principles, in order to distribute one data processing operation over a group of machines [17]. Since then, several

implementations of the model have appeared, including open source frameworks like Apache Hadoop [1].

The fact that, in MapReduce, data is processed at a granular level is what makes it especially useful in big data analytics, a crucial field of study in today's data-driven world. In 2013 alone 4.4 zettabytes (trillion gigabytes) of data were produced. The number is doubling in size every two years, meaning that by 2020, the data we create and copy annually will reach 44 zettabytes [2]. Businesses around the globe have been aligning with this trend for several years, turning the data excess into an opportunity to enhance business intelligence through descriptive and predictive analytics [11]. Big Data hardware, software and services will grow to reach 187 billion dollars in 2019 [3] and MapReduce plays a big role in this growth [11]. As Wikibon points out [4], this is especially evident since the addition of YARN (Yet-Another-Resource-Negotiator) to the Hadoop stack in 2013, which enabled it to "operate as a true multi-application framework".

1.1. Problem Statement

This section will introduce the concept of *scheduling* in the data center, which this thesis approaches, as well as the limitations that current solutions present on a specific workload type.

1.1.1. Scheduling MapReduce Applications

Once a MapReduce application, that is, an application following the MapReduce model, is sent for execution on the cluster, it becomes a *job* to be automatically managed by the system. As shown in Figure 1.2, 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. Job scheduling on computation nodes is a well-developed subject in the data center context, as it is crucial for performance. For MapReduce specifically, multiple scheduling policies have been devised to prioritize and optimize job execution, considering the particular characteristics of these workloads. However, different policies focus on different operational aspects and/or optimize with respect to different performance metrics. See Section 2.3.1 for a detailed overview.

1.1.2. Compound Objectives

In general, what most policies try to achieve is reducing the runtime of the users' jobs as much as possible on a fixed-sized cluster. Here job runtime is considered to be the time elapsed between the submission of a job and its completion. However, when considering the financial cost of compute resources, as well as their energy consumption, it is desirable to optimize resource utilization and scale the cluster according to load. Moreover, it might also be the case that a MapReduce application is part of a more elaborate business pipeline and, thus, must produce results by a certain deadline. This type of performance goal is called a Service Level Objective (SLO) and it is common for latency-sensitive applications (e.g., personalized advertising and live business intelligence, span or fraud detection, and real-time event log analysis [36]).

Let us consider a particular workload, containing a mix of both unconstrained jobs (*batch jobs*), and jobs that need to meet SLOs (*real-time jobs*). Also, the performance objective is a compound objective, looking to minimize SLO violations (the difference between the deadline and the completion time of the real-time jobs), and reduce batch job response time, while still keeping costs and consumption in check. One can already imagine that designing only one scheduling policy to achieve the objective would be difficult and error-prone.

Moreover, as shown by the short survey in Section 2.3.1, different scheduling policies work on different levels that may or may not overlap, and are specific to different kinds of workloads. Chen et al. [13] study the applicability of schedulers on their real-world MapReduce workloads. They conclude that their effects and advantages vary according to workload composition. The FIFO (First-In-First-Out) scheduler allows large jobs to run without delay, to the detriment of the smaller jobs, while the Fair Scheduler (which focuses on fair resource sharing between jobs in the system) achieves the opposite. This means that the FIFO scheduler would be more efficient for a workload containing large jobs, while the second would be more suited if large jobs are followed by many short jobs. This same effect is stated by He et al. [24], who also arrive to the

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

[2] http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

[3] https://www.forbes.com/sites/louiscolumbus/2016/08/20/roundup%2Dof%2Danalytics%2Dbig%2Ddata%2Dbi%2Dforecasts%2Dand%2Dmarket%2Destimates%2D2016

[4] http://wikibon.org/wiki/v/Big_Data_Vendor_Revenue_and_Market_Forecast_2013-2017

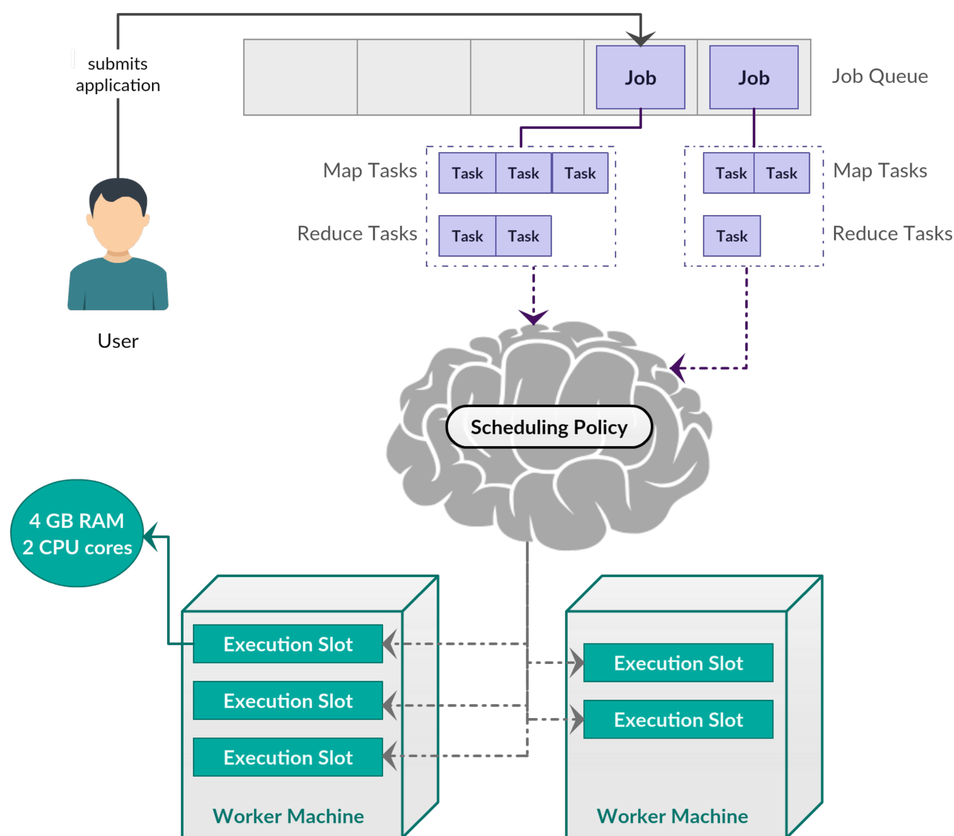


Figure 1.2: MapReduce scheduling.

conclusion that the approach for delay scheduling from Zaharia et al. [56] may behave worse than FIFO at times, allocating available slots to jobs whose data is not local if they have been skipped more times than the allowed maximum. Moreover, the authors state that finding an optimal value for this maximum is not trivial.

Thus, there is no silver bullet for scheduling any workload. *The scheduler should be optimized for the specific workload structure that the cluster will handle.* However, as Chen et al. [13] show, real-world MapReduce workloads have considerable variability. It is apparent in the types of jobs, their runtimes, their inputs and outputs, their arrival patterns, etc. And it is this variability that prevents rigorous scheduler specialization.

In conclusion, it would be desirable for the scheduler to not only keep track of all the performance metrics, but also adapt to the given workload at runtime, which adds even more complexity to its design and possibly considerable latency for runtime decisions.

1.1.3. Portfolio Scheduling

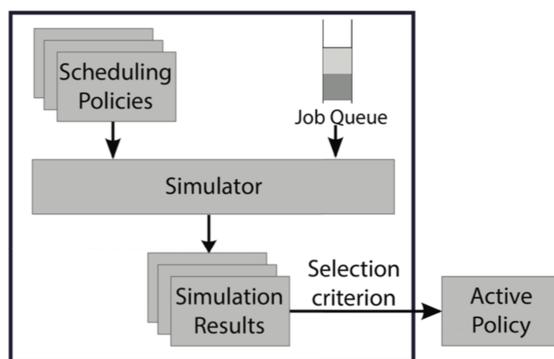


Figure 1.3: The periodic portfolio scheduler in Deng et al. [19]

A reasonable solution for achieving flexibility in scheduling would be to change the cluster's scheduling policy dynamically. This would mean each policy could remain manageable in terms of complexity and hold true to its target use case, while still leaving room for the system to adapt.

This principle is the basis of the *periodic portfolio scheduler* introduced by Deng et al. [19]. As seen in Figure 1.3, 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 concept of portfolio scheduling was borrowed from economics [30]. The first practical exploration of a portfolio scheduler can be found in for scientific workloads [18]. It was later adapted [47] to fit data center provisioning [5] needs for business-critical workloads.

1.1.4. Applicability of Previous Portfolio Schedulers

As mentioned above, portfolio scheduling has been applied to the data center context. However, previous work is not directly compatibile with the described MapReduce cluster. The approach of van Beek et al. [47] 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.

One could try to adapt the scientific portfolio scheduler to the MapReduce framework. This could be done in two ways:

1. Each job is assigned one VM. This would make the MapReduce environment run in pseudo-distributed mode, meaning parallelism is limited to concurrent thread execution and the VM must be "large" enough to support all data and computation needs. The problem with this is a higher hardware cost and the loss of scalability, which are the main benefits of distributed computing at its core.

2. Each worker is assigned one VM. Since dependencies are not supported, and reduce tasks need map output, map tasks would need to both get their data from and write their data to external permanent storage. Reduce tasks should only be added to the queue after their data is ready and must download it from the same external source. This already entails unwanted overhead, as I/O and network operations are time consuming on commodity hardware, and intermediate data sizes are often greater than input sizes (which for MapReduce applications can be up to TB order). Moreover, the portfolio scheduler defined for scientific computing predicts job runtime as a means of the last two submissions of the same user. Since this does not take into consideration any other characteristics of the job, the predictor loses accuracy when these data sizes vary. Inaccurate runtimes mean that simulated behavior can differ greatly from the real behavior of the system, resulting in the erroneous selection of scheduling policies.

1.2. Main Research Questions and Approach

Considering the above incompatibilities, it is the purpose of this thesis to explore the possibilities and limitations of applying portfolio scheduling to the MapReduce context by designing a new system specifically for MapReduce workloads. Three questions drive the research:

RQ1 *How can predictions be made on how a sequence of MapReduce jobs will behave (with respect to a series of performance metrics) when executed on a homogeneous cluster under a specific scheduling policy?*

Portfolio schedulers need to score their policies, according to a set of performance metrics. Scheduler decisions and the performance outcome relies heavily on which computation slots are freed and when. This means that a simulation needs to be conducted, using models for predicting task execution, data transfer and other system mechanisms. Chapter 3 explores the compatibility of existing solutions for this use case, and the process of combining their approaches, along with other MapReduce workload research, into a flexible simulator for a MapReduce platform.

[5] here meaning optimal placement of customers' virtual machines on compute resources

RQ2 *How to design and develop a generic portfolio scheduler that supports MapReduce workloads?*

Chapter 4 follows the design process of the proposed portfolio scheduler, POSUM (POrtfiolio SchedUler for Mapreduce, pronounced "possum"). It explores different parameter variations in achieving the target performance goal, based on the design method introduced by Deng et al.

RQ3 *How to demonstrate the capabilities and evaluate the performance of the proposed portfolio scheduler in realistic situations? In particular, does the portfolio scheduler perform better than its constituent policies?*

The resulting design propositions are evaluated in Chapter 5. We use a set of synthetic workloads to study the accuracy of predictions, the validity of the simulator, and finally, the performance and stability of the proposed scheduler, compared to the single-policy approach.

1.3. Main Contributions

The main contributions of this thesis are:

1. (Conceptual) We design POSUM, a generic portfolio scheduler architecture for MapReduce systems.
2. (Technical) We implement POSUM as a modular open-source addition to Hadoop YARN, which serves as a platform for practitioners and researchers to further investigate and extend portfolio scheduling for MapReduce ecosystems.
3. (Conceptual & Technical) We propose and implement an easily extensible set of scheduling policies, the portfolio of POSUM. Our policies of choice represent state-of-the-art scheduling techniques for MapReduce systems.
4. (Conceptual) We design POSim, a generic simulator for the online prediction of MapReduce task behavior, equipped with a set of three predictors, *basic, standard*, and *detailed*, each making use of increasingly complex system-level metrics.
5. (Technical) We implement and open-source POSim as part of the same Hadoop module.
6. (Technical) We design and carry out a set of real-world experiments to evaluate POSUM. Our workload is a mixture of real-time and batch jobs, with the purpose of minimizing deadline violations, while keeping batch job slowdown in check.

This work forms the basis for an article currently under submission: M. Voinea, A. Uta, and A. Iosup, POSUM: A Portfolio Scheduler for MapReduce Workloads, IEEE Big Data, submitted on the 19th of August, 2018 (expected acceptance ratio 15-20%).

1.4. Thesis Structure

The rest of this document is structured as follows: Chapter 2 contains sections with background information required for understanding the specifics behind some concepts or decisions contained in other sections of the thesis. Their applicability will be indicated by appropriate references throughout the document. Chapters 3, 4, 5 focus on answering research questions 1, 2, and 3, respectively. Finally, Chapter 6 reiterates the method and findings of the thesis. See Figure 1.4 for the possible paths for reading the document. If certain terms, abbreviations or phrases are unclear, or need specification, please refer to the glossary in Appendix A for their definition in the context of this thesis.

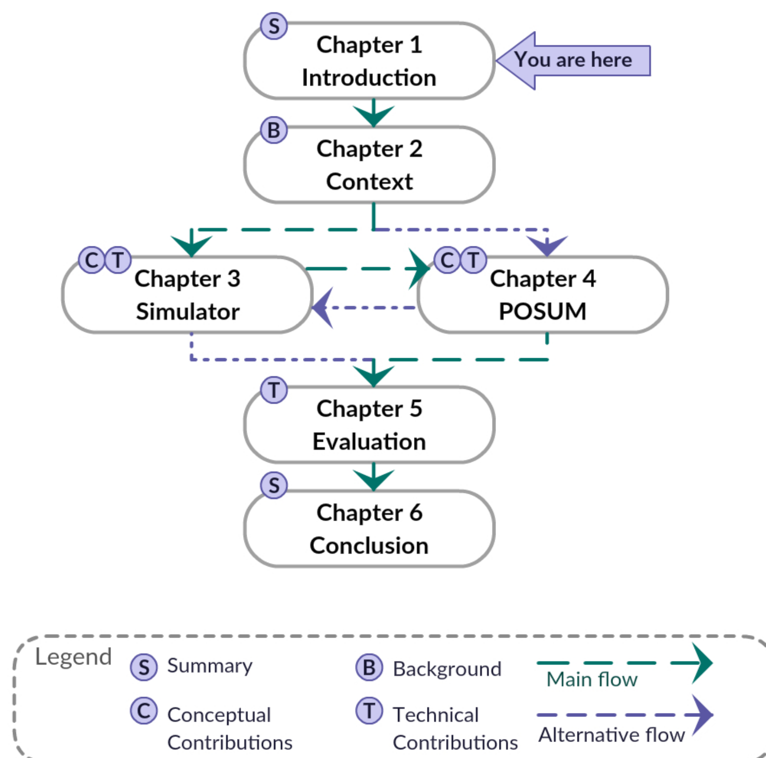


Figure 1.4: Thesis composition and sequence.

2
Context

This chapter conveys detailed information about the concepts, systems and techniques used in this work. Section 2.1 defines the system model that we are targeting. Section 2.2 dives into the internals of the Hadoop YARN architecture and operation. Section 2.3 contains short surveys on the state of the art of scheduling, simulation, and prediction within the MapReduce paradigm. Although this chapter does not answer any research question by itself, it lays the foundation for understanding the content of the rest of the thesis, and can be returned to whenever needed.

2.1. System Model for MapReduce Processing

Although implementations of MapReduce can be optimized for small shared-memory machines or large multiprocessors [43], the original framework targeted large clusters of commodity machines. This is the same setup on which MapReduce is widely used today at Google [44], Facebook, Yahoo!, Cloudera [13] [14] and many users of the Hadoop implementation [1]. For example, Ebay uses its Hadoop cluster for search optimization and research, LastFM and Spotify analyze audio features and generate music recommendations on theirs, while advertising agencies run analytics and process petabytes of customer interest data each day. Considering its popularity, researchers also use Hadoop deployments on commodity clusters to analyze and optimize workload performance (see Section 2.3.1). Consequently, this thesis will assume the same configuration.

This work considers only systems comprised of a network (cluster) of machines (nodes) located in a single data center, as shown in Figure 2.1. One of the machines is designated as master node and contains all the coordination logic, while the rest are worker nodes for processing and storing the data. The machines contain commodity hardware, meaning that it is affordable and easy to obtain even by regular consumers. Generally, this means 2 to 8 CPU cores, 4 to 24 GB of RAM, and 1 to 12 TB of spinning HDD storage [2]. To simplify analysis and simulation, the cluster will be considered homogeneous: the amount and quality of computation resources (RAM memory, CPU cores, disk storage space and speed) are the same on each node and they do not have any background load. However, the network speed between each pair of nodes is variable, depending on the arrangement of nodes in racks (stacks of machines that are served by the same network switch). For our experiments, we consider that the processes that comprise POSUM will run on a separate machine, to avoid any interference with the framework's operation during evaluation.

The computation nodes also share their data storage resources, forming a distributed and redundant storage system (DFS), the same way Google File System (GFS) and Hadoop Distributed File System (HDFS) work. This also allows for input data to sometimes reside on the same node that performs its computation, saving valuable data transfer time on clusters with commodity hardware. The file system displays the same master-slave architecture, with one node containing file metadata and replication information, and the rest storing file content blocks.

As mentioned in Section 1.1.2, the target workload of this study contains a mix of both batch (BC) jobs, and deadline-constrained (DC) jobs. The performance objective of the portfolio scheduler is to minimize

[1] `http://wiki.apache.org/hadoop/PoweredBy`
[2] `http://wiki.apache.org/hadoop/PoweredBy`

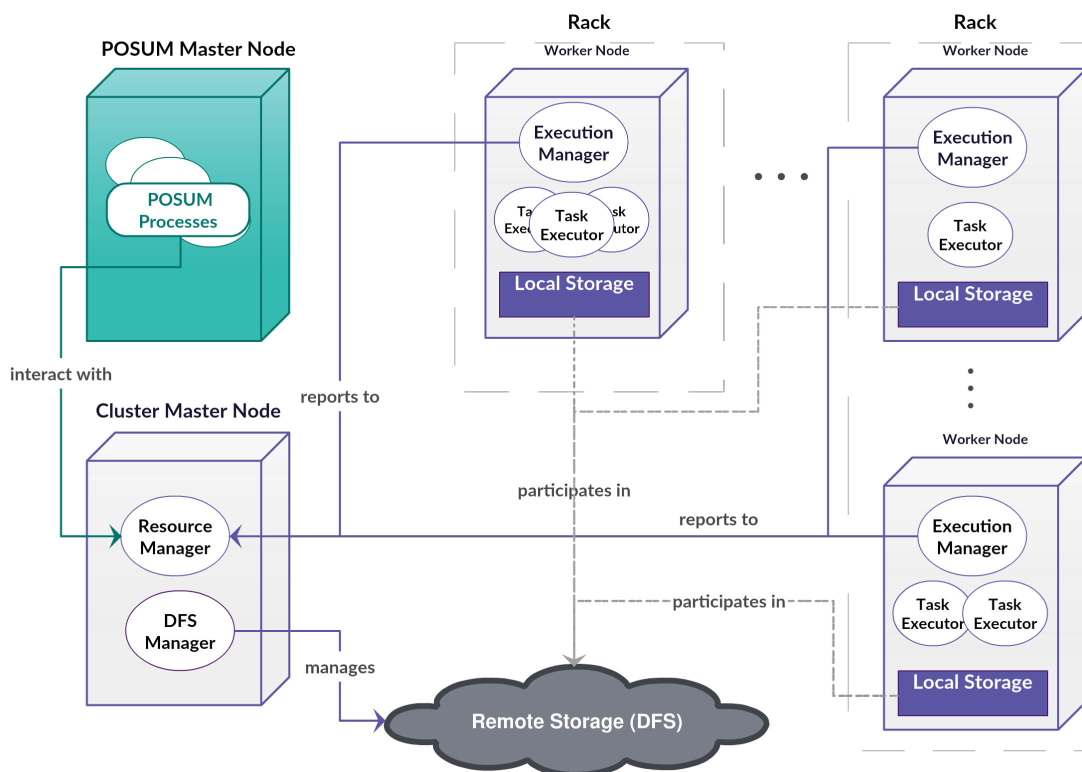


Figure 2.1: System model (DFS stands for Distributed File System).

deadline violations (the difference between the deadline and the completion time of the DC jobs), and reduce BC job slowdown (the ratio between the runtime of the job and the total execution time of its tasks, disregarding wait time in the system). Since short jobs can have a considerable impact on the overall performance, bounded slowdown (BSD) is used, as in the model by Deng et al. [18]: execution time has a fixed configurable minimum value.

In most of the user environments listed on the Hadoop website, and most of the research conducted on MapReduce optimization, the cluster size is considered fixed. It fluctuates only in studies referring specifically to online dynamic cluster resizing [22] [38], which proves difficult, because adding or removing nodes from the distributed file system requires heavy data transfer. However, if the cluster is hosted by a Infrastructure-as-a-Service (IaaS) provider, such as Amazon EC2 and Microsoft Azure, or it is part of a shared data center, it might be valuable to take advantage of the ease of scalability and, thus, better achieve the required SLOs. The drawback, of course, is the increase in costs, as IaaS customers are generally charged using a pay-as-you-go model that rounds up to the next whole unit of lease time (usually an hour). Consequently, *this thesis will also explore different methods of growing and shrinking the cluster dynamically, while minding the trade-off between cost efficiency and performance.*

2.2. Primer on MapReduce and YARN Internals

As mentioned before, a MapReduce application becomes a job once it is sent to be executed on the cluster. Since the job's input can be very large, before execution starts, the input is divided into a number of *splits* of a fixed maximum size, and each of these is assigned to one instance of the mapper code, called a *map task*. After being processed by the mapper, the preliminary output (key-value pairs) is sorted by key and grouped to be read over the network by the associated reducer in the process called the *sort and shuffle phase* which is done automatically by the framework. After arriving at the reducer, the *reducer tasks* (instances of reducer code) aggregate the pair values and output their results to the DFS. Since the target reducer is determined by the key of the pairing, partitioning data over a low entropy key space leads to imbalanced reducer load [1].

Thus, jobs are comprised of a set of independent map tasks and a set of independent reduce tasks, and each task is scheduled separately. Since reduce tasks need their input from map tasks, reduce tasks cannot finish until all map tasks have finished, but they can be started in parallel if they have their input partition

ready. It is the responsibility of the cluster's *master* node to apply the scheduling algorithm (policy) to allocate appropriate resources for each task on a *worker* node. After that, the task itself (map or reduce routine) is carried out by the worker. The scheduling algorithm may also choose to *preemt* a running task (stop its execution and re-allocate its resources) if it is in the interest of achieving the scheduling goal.

This is the traditional MapReduce flow. The Hadoop YARN implementation separates functionality even more, by dividing the task scheduling and MapReduce coordination between two entities in the system: the Resource Manager and the Application Master.

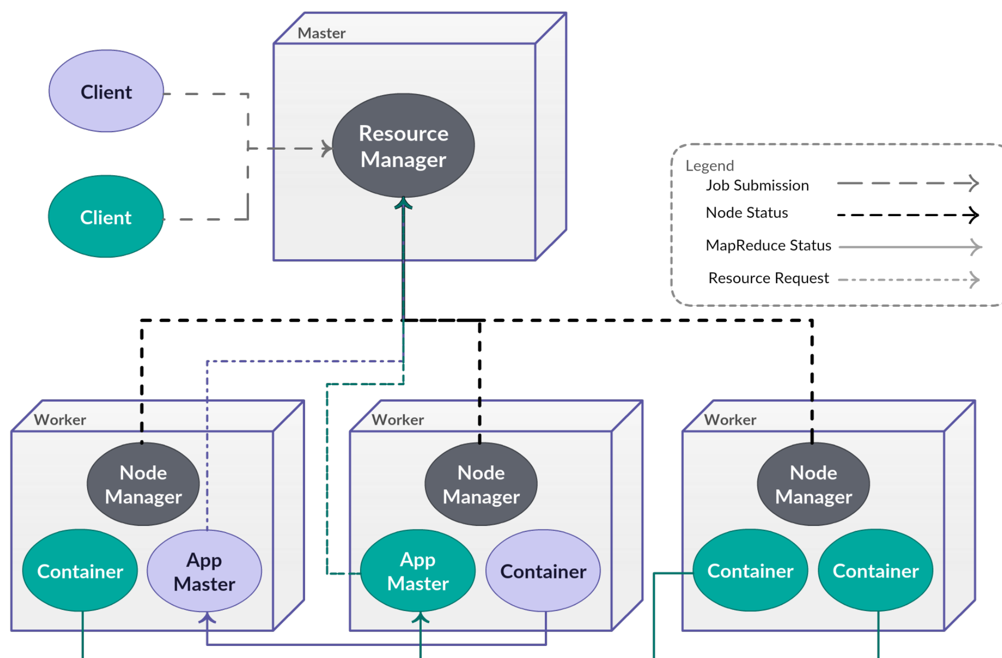


Figure 2.2: YARN architecture (reproduced from [7]).

As seen in Figure 2.2, the Resource Manager (RM) is the one that receives application submissions from the clients and node status updates from the Node Managers on the worker nodes. Its responsibility is to assign *containers* (a set of computation resources) on the worker nodes in order to run the clients' applications. When an application is submitted, the RM begins by assigning and launching its first and most important container: the Application Manager.

The Application Manager (AM) is the process that coordinates the execution of an application. In the case of MapReduce, this means running and keeping track of the progress of the job's tasks. In order to obtain the necessary resources for running the tasks, the AM sends resource requests to the RM. Once the RM's internal scheduling policy assigns a new container to the AM, the latter can launch the next appropriate map/reduce process.

An important aspect of MapReduce scheduling is achieving data locality. This refers to executing map tasks on the nodes of the DFS that have their input splits on their local storage. Achieving data locality avoids data transfer delays and leads to better performance. The schedulers that YARN comes equipped with all take this into account, in the sense that a job's resource requests are fulfilled in the order of their specificity: when containers become available on a node, an AM's requests for the specific node are honored first, then the requests for the same rack, and finally the ones for any node.

2.3. Survey of the State of the Art

This section contains a synthesis of works that constitute the state of the art in MapReduce scheduling, simulation and evaluation. These studies have inspired different parts of the following chapters from this thesis.

2.3.1. MapReduce Scheduling

Section 1.1.2 introduced examples of distinct performance metrics that schedulers optimize for. This segment surveys different scheduling approaches and their objectives.

The first MapReduce schedulers, i.e Google's initial implementation [17] and Hadoop's scheduler until 2008, operated on *First-In-First-Out (FIFO)* order, meaning that the jobs were scheduled in the order of their arrival. However, at task level, there was one modification required for boosting performance under scarce network bandwidth: achieving *data locality* for tasks whenever possible. Data locality refers to 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. Thus, the prioritization of tasks belonging to the same job is done by proximity to the node on which the freed slot is.

The issue of sharing data center resources later appeared, meaning a new aspect required attention: *fairness* amongst users. Each user or group of users needed to be guaranteed a fair share of the cluster, which resulted in the appearance of the Hadoop Fair Scheduler [46]. This new policy, which uses max-min fairness, allows the creation of *pools* of jobs that are each a minimum share of the cluster's slot capacity. Slots are awarded first to pools that are under their minimum share, and then evenly distributed to those that still have a demand. On the one hand, this prevents one user group from using up all the resources with a lengthy job, but on the other it may sometimes break data locality, resulting in longer runtimes for tasks. A more complex division of cluster resources was later provided by Hadoop's Capacity Scheduler that allows for a hierarchy of pools, each assigned custom soft and hard limits on their share.

Later on, the data locality constraint gained further attention, with the appearance of *delay scheduling* approaches. It was first applied to Hadoop Fair Scheduler [56] and later to its FIFO scheduler [24]. This policy relaxes the order of jobs in the queue (irrespective of the priority type applied) to achieve better data locality. Whenever a slot frees up, jobs that do not have local tasks are skipped, even if they have the highest priority, instead allowing tasks from other jobs to run if their data is local. To avoid starvation, each job can only be skipped a fixed amount of time. As a result, the time lost by waiting for a local slot to open up is made up for by speeding up the runtime of the task.

Cheriere et al. [15] 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 (jobs with a small input size and short execution times). Thus, they implement two versions of their *Shortest Remaining Time First* scheduling policy (hSRTF), to give higher priority to such jobs. The remaining time is calculated by estimating the runtime of the future map and reduce tasks based on the average runtime of finished map tasks. The authors use this priority first as a means to order jobs (a variant of FIFO), and as a weight for sharing the cluster's resources (a variant of the Fair Scheduler). They also encourage collocation of map and reduce tasks of the same job to minimize shuffle time, and consider the effects of using preemption with their policies. The results show significant improvement for the targeted jobs (small short ones) and deterioration of response times in large jobs.

In the attempt to find a more encompassing scheduler definition, the study of Nguyen et al. [40] uses a compound metric for determining job prioritization. Their scheduler, HybS, is a variant of the FIFO policy where the order of the jobs is given by a function $W^{\alpha} R^{\beta} N^{\gamma}$, where W is the wait time of the job compared to the average wait time in the system, R is the remaining map runtime compared to the average map runtime in the system, and N is the number of remaining tasks compared to the total number of tasks in the system queue. The powers α, β and γ are priority parameters and their values determine the intention of the scheduler (for instance $\alpha = 1$, $\beta = \gamma = 0$ give the FIFO policy and $\alpha = 0$, $\beta = -1$, $\gamma = 0$ result in Shortest Job First).

Yet another aspect of scheduling MapReduce is the presence of *stragglers*. Stragglers are tasks that are taking long not because of the required computation, but because of issues with machines or the network. These tasks have an even greater impact on MapReduce workflows that generated by higher level frameworks like Hive [3] and Pig [4] that run on top of Hadoop. Workflows consist of multiple sequential map and reduce phases with barriers between them: a phase cannot start before the previous phase has finished. This means that a straggler task at an early phase can considerably delay the entire job by causing unnecessary wait time. For this reason, both Google's initial implementation and Hadoop used speculation: starting copies of the task that is presumed to be straggling in the hope that one would finish sooner. However, this practice results in consuming slots, meaning the decision of when to start speculative copies and how many of them should run was the basis of extensive research. This trade-off is addressed by schedulers like LATE [55], SAMR [12] and later, with focus on approximation analytics, GRASS [2].

Apart from job runtime and user fairness, meeting specific Service Level Objectives (SLOs) for jobs is another direction of optimization for MapReduce schedulers. This translates mainly to having temporal deadlines for certain jobs which are part of a business pipeline and, thus, must complete in definite time. Deadline

[3] `http://hive.apache.org`
[4] `https://pig.apache.org`

scheduling is approached by Kc and Anyanwu [33] through the construction of a system that estimates the number of slots that the job needs to be allocated in order to make the indicated deadline. It accepts or rejects jobs based on whether this number is smaller than the available slots in the system. The task runtimes are required from the user in order for the model to work. Polo et al. [42] try to do the same calculation of required slots by predicting the runtime of future tasks from the performance of the finished tasks of currently running jobs. They focus on the tasks in the map phase, as they are the most time-consuming in their workload. Dong et al. [20] also incorporate reduce phase runtime in their predictor, which samples map tasks to compute their duration average and executes a dummy reduce task for estimating shuffle costs in the reduce phase. Verma et al. [48] [50], also consider both phases in designing their job profiler. However, this system is capable of predicting the task durations of a set of regularly run applications without waiting for any tasks to run, by analyzing the jobs' characteristics and past runtime behavior. Lim et al. [36] attempt to address strict deadlines for Service Level Agreements (SLAs), with their system based on constraint programming (CP), which again assumes task runtimes to be provided by the user.

In their study [35], Li et al. use their own job profiler to determine the necessary number of allocated slots for a job to make its deadline, in conjunction with DVFS (Dynamic Voltage and Frequency Scaling). The goal in this case is extended to also reducing energy consumption of the cluster, while still meeting SLAs. Another example of using scheduling techniques for achieving energy efficient resource utilization is the study of Yigitbasi et al. [54].

Further research exists for optimizing MapReduce runtimes on virtual machines by influencing scheduling at hypervisor level (e.g. Kang et al. [32] and Park et al. [41]), but such systems are beyond the scope of this work.

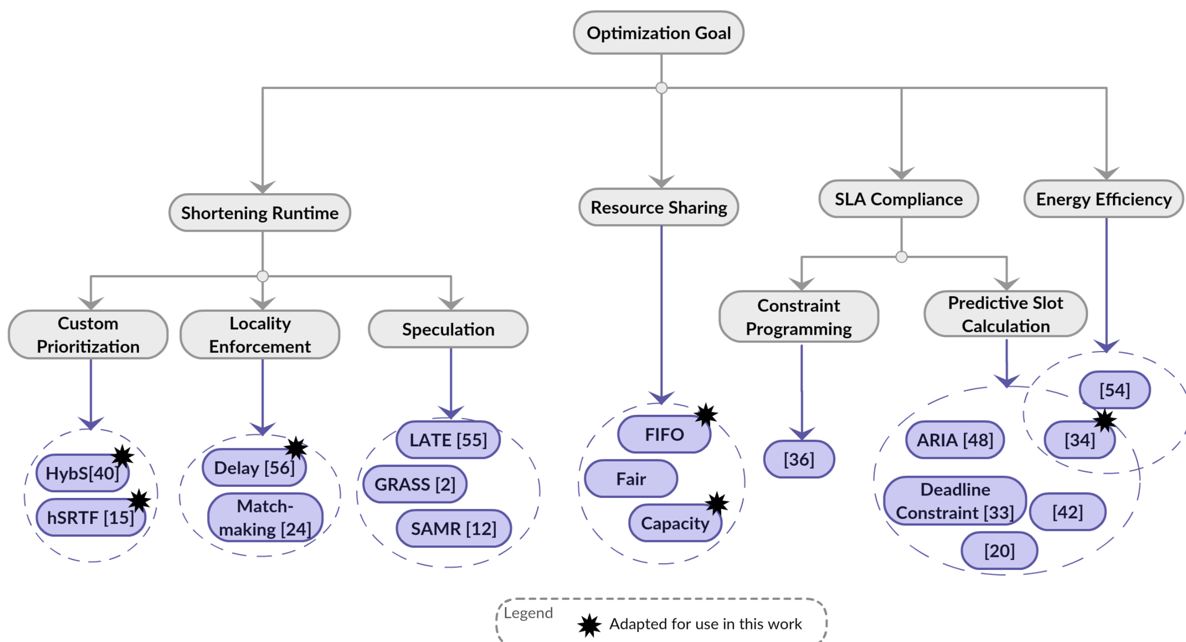


Figure 2.3: Goal-based scheduler taxonomy.

In conclusion, the are two underlying aspects that can be the basis of a taxonomy of MapReduce schedulers in literature. The first is the method used for enforcing priority: on the one hand, there are order-based scheduling policies. These are generally modelled as a single queue in which jobs are ordered by a specific prioritization mechanism. This is the case for the FIFO scheduler and all the ones that are derived from it. On the other hand, share-based schedulers strive to achieve a specific share distribution among pools, users or jobs. These are multi-queue implementations like the Fair and Capacity schedulers, along with the ones inspired by them. Of course, some scheduling ideas presented in the aforementioned studies can be applied to both types of priority management. This was done for the hSRTF scheduler, for instance.

Another classification driver is the optimization goal of a scheduler. Figure 2.3 shows an example hierarchy.

2.3.2. Simulator Types and Capabilities

As part of the desired portfolio scheduler is responsible for simulating the behavior of the cluster under a given load, this section is dedicated to clarifying the concept of a simulator and exploring existing solutions for simulation in the MapReduce context.

What is simulation? Robinson [45] defines it as the imitation of a system, and divides it into static simulation (which imitates a system at a point in time) and dynamic simulation (which imitates a system as it progresses through time). In the context of distributed computing, simulations are used as a means of conducting *in silico* (virtual) experiments in order to study and understand the behavior of applications running on distributed platforms [10]. Experimentation on such platforms of the *in vivo* (real) or *in vitro* (through emulation) kinds are considerably more expensive with regards to time, electrical power, and natural resource consumption.

It is for this purpose that domain-specific simulators have been developed targeting the areas of high performance, grid, volunteer, and peer-to-peer computing. Casanova et al. cover the state of the art of simulators in their study [10], and identify a common design backbone: (i) creation of simulation models; (ii) platform specification; (iii) application specification. Simulation models refer to the implementation of the simulated activities on the simulated resources in the simulated time. For example, an activity could be the reading of a chunk of data from a hard disk into memory. The resources involved would be the CPU, disk, and memory, while the simulated time could be measured in CPU cycles. Once all resource models have been chosen, each of them needs to be instantiated with appropriate parameters, and the interconnections of these resources need to be mapped out (platform specification). Application specification is done by describing how resources are used by the simulated application, either offline (by replaying of event traces or logs captured during real world execution), through formal (e.g. finite automata) description, or programmatically (through a set of functions that describe Concurrent Sequential Processes). Figure 2.4 shows a visual representation of these simulator characteristics.

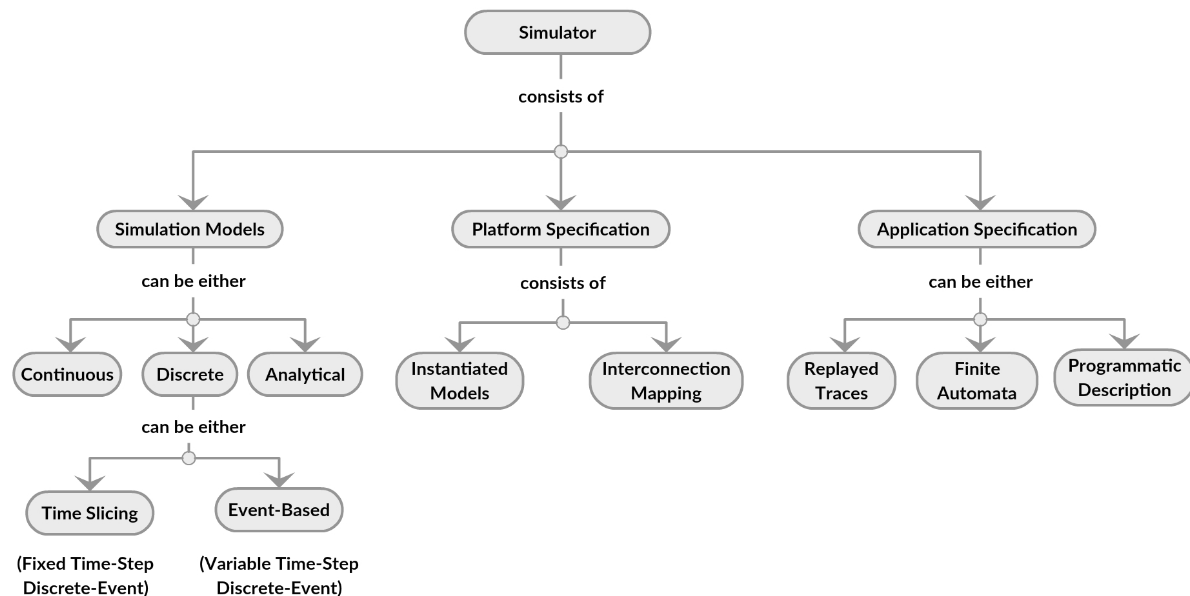


Figure 2.4: General simulator characteristics.

The way time passes in the simulation gives us several types of simulators [45]. In a *continuous simulator*, the state of the system changes continuously through time. This is more common in the fields for physics and natural sciences for simulating the movement of fluids, for instance, or chemical plants and oil refineries. However, in computing, time is generally a discrete function, since the period between two state changes is a multiple of one CPU cycle. Such is the case of *discrete simulators*. A common way of modelling a discrete simulator is time slicing. Time progresses by a constant time-step Δt. If Δt is small enough, one can approximate a continuous simulation, which is how computers can simulate continuous natural phenomena. However, if the state of the simulated system frequently remains unchanged over periods of several Δt steps, it can be inefficient to use this time model. *Discrete-event simulators* are to be preferred in this case. The basic three-phase discrete-event simulator regards each beginning or end of an activity as an event and

at each step progresses directly (phase A) to the time of the next *booked event* (event that was scheduled to occur at a certain time in the simulation) in its queue. After processing all events with that time-stamp (phase B), it processes all events that were generated as a result of the booked events. These are called *conditional events*, and the cycle restarts only after this phase (phase C) is complete. The time slicing approach can be considered a special case of a discrete-event simulator with a fixed time-step, where the passing of each time step constitutes a booked event.

Intuitively, simulation duration increases with the granularity of the time model. Thus, continuous simulators are generally less efficient than fixed-time-step (time slicing) simulators and the latter are slower than their variable-time-step (discrete-event) version. However, even when processing events only, the computation load for the simulator can be considerable. This is why analytical models are sometimes used to simulate resource usage. An analytical model is a series of mathematical equations that can directly generate the necessary measurements or state information as output, without a high-fidelity state transition sequence. This version is preferred for modelling the CPU in computing systems, for instance, since running full-fledged cycle-accurate simulation of computer instructions is extremely time consuming and its results might not be as accurate as intended [10]. As Casanova et al. observe, most distributed computing (DC) simulators estimate execution times by dividing a compute cost (such as the number of instructions required for an activity) by the compute speed (the frequency of instructions). When it comes to storage and network models, however, these simulators differ in the level of detail and simplification decisions, from block or packet-level discrete-event simulation, to analytical models based on average disk speed and bandwidth benchmarks.

Casanova et al. argue that the secret for designing an accurate yet scalable simulator for distributed platforms is versatility. Thus, they explain and evaluate the design of SimGrid, a versatile parallel (enabled by Concurrent Sequential Processes) discrete-event simulator (PDES), capable of adapting its largely analytical resource models to any target application domain within DC. And, indeed, it has been adapted to the MapReduce (MR) context by Kolberg et al. [34]. Their simulator, MRSG, uses SimGrid for simulating all the data transfers and task computations. MRSG models the logical flow of actions dictated by the MapReduce paradigm, while leaving the exact MR application specification to the user. The user can thus define functions for how to distribute data chunks through the cluster (DFS structure), how to choose which task to run next (scheduling), how to compute the cost in CPU cycles for each task type (map/reduce phases), and how to generate intermediate data from the mappers to the reducers (shuffle/sort phase). The result wished to be a versatile, yet resource accurate simulator for MR applications. MRPerf [52] is another instance of a MapReduce simulator written in C++ that aims for accurate resource modelling. It does packet-level simulation based on ns-2 [31], and uses a simple disk model that can also be substituted of a block-level model like DiskSim[21]. It also offers the same application specification possibilities as MRSG, with only a few more simplifications. A Java-based simulator called MRSim was developed for the same purpose by Hammoud et al. [23]. It is built on the SimJava [28] discrete-event engine and uses GridSim [8] to simulate the network communication. This simulator offers even less customization. For instance, it allows only for fixed task costs and seems to not vary the distribution algorithm or the scheduler, from what the short paper presents.

The above simulators have focused more on simulating the underlying resource usage of a MapReduce cluster, either ignoring or leaving the accuracy of the scheduling, data distribution and computation cost derivation to the user. However, there are simulators dedicated to mimicking the exact event-based flow of a MapReduce system, for the purpose of validating new schedulers or evaluating their behavior given a certain system load. A first example is Mumak [4], an event-driven simulator for Hadoop 1.x implementations that can replay traces collected with Rumen [5] from the execution logs of a cluster. It does not model any resource usage, but merely assumes that tasks and jobs behave according to their recorded history, and forwards all scheduling decisions to the scheduler class it received as input, with no modifications. An improvement over Mumak seems to be SimMR [49]. Verma et al. designed SimMR to be more accurate than Mumak by also modelling the shuffle/sort phase in the simulation, but also more efficient by not simulating the exact task trackers and the heartbeats between them. Moreover, SimMR can replay synthetic traces obtained either through statistical processing of actual traces, or by profiling past jobs according to their traces and modelling their resource requirements (as a follow-up of their research for ARIA [48]). Thus, artificial workloads, composed of variable-size instances of previously profiled application types can also be run on the simulator. The same can be said for the What-If Engine [26] of the self-tuning analytics system designed for Hadoop, called Starfish [27]. The engine creates profiles of previously run jobs using both black-box techniques based on supervised learning for estimating execution times for tasks and phases, as well as white-box (analytical) models for data flow during execution. As a result, the execution of a job or even entire workload execution can be simulated for any new network topology or resource distribution among nodes. The task sheduler is

a pluggable component whose default implementation is a lightweight discrete event simulator of the FIFO scheduler.

With the evolution of Hadoop to 2.x (YARN), other simulation solutions were required to fit the new execution model. The simulator that comes with the Hadoop distribution is called the Scheduler Load Simulator (SLS) [6] and it's purpose is similar to Mumak's. However, although it mimics the exact framework internals, it is implemented to work with fixed-time-steps equivalent to the initial heartbeat configurations of the trace. This means that the trace takes just as long to simulate as it did to be executed in real life. A more efficient and complex alternative is YARNSim [37]. Its creators, Liu and Sun based their simulator on the parallel discrete-event simulation package ROSS [9] which models each distinct component of the system as a logical process (LP) that communicates with other via timestamped messages. The disk and network usage is modeled using CODES [16], a package based on ROSS capable of low-level hardware simulation, while the CPU and memory are modeled analytically. Each Hadoop entity, such as the Resource Manager or Application Manager are represented by a separate LP that communicates realistically with its counterparts, using a pattern similar to the state-machine transitions that happen during real execution on the cluster. However, this simulator cannot be used for scheduler validation, as the scheduler is currently a fixed LP implementation of the standard FIFO scheduler. Also, in contrast with SLS, it cannot play traces, but needs instead a manual configuration of all the details of a specific job type before simulating it.

As the short survey above shows, each studied MapReduce simulator has different characteristics and targets different user requirements. Table 2.1 offers a summary of these properties. As all the simulators use some form of programmatic description as application specification, we noted to what degree it relies on recorded traces and to what level of detail the resource model is defined. Then we looked at the MapReduce-specific characteristics. Column eight contains a grade (from 1 meaning least to 10 meaning most) as an approximation of the simulator's time efficiency, based on the type of resource models that are used and the use of parallel processing. For MRPerf and MRSim, there was less indication provided in the paper with regard to efficiency, which leads to their grade being expressed as a range of possibilities. The final two columns refer to the availability of their implementations for use and extension. The cells highlighted in red represent characteristics that make the corresponding simulator incompatible with the use case presented in this thesis. Further explanation is offered in Chapter 3.

Simulator	Simulation (Time) Model	Application Specification		MR Characteristics			Estimated Time Efficiency (/10)	Code Availability	
		Resource Model	Simulation Input	MR Internals Model	Plug-in Scheduler	YARN Ready		Binaries	Sources
MRSG [34]	parallel discrete-event	coarse-grained	manual job configuration	no	yes	no	8	yes	yes
MRPerf [52]	discrete-event	fine-grained	manual job configuration	no	yes	no	2 - 6	yes	yes
MRSim [23]	discrete-event	fine-grained	manual job configuration	no	no	no	2 - 5	yes	yes
Mumak	discrete-event	no	trace	yes	yes	no	9	yes	yes
SimMR [49]	discrete-event	no	trace or learned profiles	yes	yes	no	10	no	no
Starfish [27]	discrete-event	coarse-grained	learned profiles	yes	yes	no	9	no (broken link)	no
SLS [6]	fixed-step (real-time)	no	trace	yes	yes	yes	1	yes	yes
YARNsim [37]	parallel discrete-event	fine-grained	manual job configuration	yes	no	yes	7	on-request	on-request
POSim (Chapter 3)	**parallel discrete-event**	**no**	**trace or learned profiles**	**yes**	**yes**	**yes**	**10**	**yes**	**yes**

Table 2.1: Extended characteristics of MR simulators.

2.3.3. MapReduce Task Runtime Prediction

An overview of the literature on MapReduce scheduling reveals three main areas of study where predictions need to be made on task execution: (1) straggler mitigation, (2) deadline-aware scheduling, and (3) simulation. The first centers around the concept of stragglers: tasks that take longer to finish than other similar tasks, even though they are doing the same amount of work. These types of tasks can have a considerable impact on the overall completion time of a job, especially in iterative MapReduce workloads, where later phases depend on the full completion of earlier ones. Studies such as those of Ananthanarayanan et al. [2] [1] focus on identifying stragglers and finding solutions for minimizing their effect through judicious scheduling of speculative copies of the task. In their work, task runtime prediction takes into account the data to be processed by the task and the average processing rate of past tasks of the job.

$$r_i = d_i \cdot \rho \cdot \lambda \tag{2.1}$$

, where:

$$\begin{aligned} r_i &= \text{runtime of task i,} \\ d_i &= \text{size of the data to process by task i,} \\ \rho &= \text{average data processing rate of completed tasks,} \\ \lambda &= \text{location factor (custom indicator of processing latency due to machine or network conditions).} \end{aligned}$$

Secondly, we have the research direction of scheduling jobs to meet specific SLOs (deadlines). The runtimes of tasks need to be estimated in advance, in order for the scheduling algorithm to know how many of a job's tasks it should schedule in parallel. Polo et al. [42] try to predict the number of resources (execution slots) needed by each job by operating under the assumption that all map tasks and all reduce tasks have a duration similar to the average duration of the previous maps and reduces of that job that have run on the cluster. As reduces run later than maps and data about them might not be available for estimation, a user can provide a ratio of the difference in runtime between the two task types in the job's configuration file.

$$r_i^M = \mu^M \quad , \quad r_i^R = \mu^R \tag{2.2}$$

, where:

$$\begin{aligned} r_i^M &= \text{runtime of task i of type map,} \\ r_i^R &= \text{runtime of task i of type reduce,} \\ \mu^M &= \text{average runtime of completed map tasks,} \\ \mu^R &= \text{average runtime of completed reduce tasks} \end{aligned} \tag{2.3}$$

Kc et al. [33] use a more detailed model of execution times, where the *input sizes* of the tasks are taken into account. For the map phase, the runtime of each task is calculated as the product between the size of the input it has to process and the task's observed *processing rate*. The same concept applies for the reduces, only the size of their input is an observed fraction of the input provided to the mappers (*map selectivity*). Also, the processing rate is split into two factors: the rate at which the data to be copied from the mappers to the reducers (*shuffle rate*) and the rate of processing of the reducing algorithm (*reduce rate*).

$$r_i^M = d^M \cdot \rho^M \quad , \quad r_i^R = f \cdot d^M \cdot \rho^{R_s} + f \cdot d^M \cdot \rho^{R_r} \tag{2.4}$$

, where:

$$d_M = \text{average map input data (split) size,}$$
$$f = \text{map selectivity,}$$
$$\rho^M = \sum_{i \in C^M} \frac{r_i^M}{d_i} = \text{average data processing rate of completed map tasks,}$$
$$\rho^{R_s} = \sum_{i \in C_R} \frac{t_i^{R_s}}{f d_i} = \text{average transfer rate in the shuffle phase of the reduce,}$$
$$\rho^{R_r} = \sum_{i \in C_R} \frac{t_i^{R_r}}{f d_i} = \text{average data processing rate of the reduce algorithm,}$$
$$t_i^{R_s} = \text{time required to copy the reduce input for task i (shuffle time),}$$
$$t_i^{R_r} = \text{time required by the reduce algorithm of task i to process the data.}$$

Dong et al. [20] approach their scheduling study for mixed deadline-constrained and batch job workloads with a similar principle. However, they compute the average map and reduce progress rates by sampling a given number of map tasks, running them and then running an extra reduce task that processes only a fraction σ of the output of those maps. The output of that reduce task is discarded, but the end result is that an estimation of the job's deadline is available much sooner than the completion of the map phase. The formulas are similar, but they add an extra constant time factor to the duration of a reduce task for the cost of merging the copied data on the reducer side (merge time).

$$r_i^M = \mu^M \quad , \quad r_i^R = (t_i^{R_s} + t_i^{R_r})/\sigma + t_i^{R_m} \tag{2.5}$$

, where:

$$\sigma = \text{reduce data sampling ratio,}$$
$$t_i^{R_m} = \text{time required by the merge operation of reduce task i.}$$

Verma et al.[50] devise a job profiler that can use past execution data of the cluster to model the behavior of future routine jobs and meet their deadlines. In their work, they take reduce task deconstruction even further by identifying a particularity of some MapReduce platforms such as Hadoop: reduce tasks are started before the map phase is complete. This means that the first reduce tasks that are spawned in parallel to the running maps start copying their data from the completed maps, but have to wait for all maps to finish before they can merge their input and run their reducing algorithm, making the shuffle phase of these early (first wave) reduce tasks abnormally large compared to later tasks. To overcome this skew in runtimes that is dependent on how many slots the scheduler assigns to the map tasks, their system keeps separate measurements for the *typical shuffle* durations (input dependent) and the *first wave shuffle* durations that are equal to the time needed to finish the shuffle after the map phase was complete.

$$r_i^M = \mu_M \quad , \quad r_i^R = \begin{cases} t_i^{R_s,T} + t_i^{R_r} & \text{if it is a typical reduce task} \\ t_i^{R_s,1} + t_i^{R_r} & \text{if it is a first wave reduce task} \end{cases} \tag{2.6}$$

, where:

$$t_i^{R_s,T} = \text{average duration of the typical shuffle in a reduce task,}$$
$$t_i^{R_s,1} = \text{average duration of the shuffle in a first wave task.}$$

Map task runtimes are considered stable irrespective of the total job input, as more data only results in more map tasks processing equally large chunks. However, since the number of reduce tasks can be chosen by the user, the input size of the reducer tasks can vary can differ accordingly. In order to compensate, Verma et al. use machine learning to model the variation and estimate durations for arbitrary data sizes.

$$t_i^{R_s,T} = \alpha^{R_s} \cdot d_i^R + \beta^{R_s}, \tag{2.7}$$
$$t_i^{R_r} = \alpha^{R_r} \cdot d_i^R + \beta^{R_r}, \tag{2.8}$$

where α^{R_s}, β^{R_s}, α^{R_r}, and β^{R_r} are linear regression coefficients calculated for the typical shuffle times and reduce times respectively.

The above deadline-oriented studies target heterogeneous environments where the processing rates of tasks and the cost of network transfers remain relatively steady throughout the workload and from machine to machine. However, when worker nodes are run as virtual machines in a multi-tenant cluster, the quantity and quality of the resources that run the MapReduce workload can vary. Zhang et al.[57] use a setup where both dedicated and shared nodes run the task computations. Instead of working with slots as the basic unit of resource, they use the residual capacity of a slot: the percentage of that slot's compute resources that can be used by the job. Their system compiles a regression model for each job type and can thus estimate the completion time of a task given a certain residual capacity. Again, tasks of the same type are considered uniform.

$$r_i(c) = \alpha \cdot e^{\beta \cdot c} + \gamma \cdot e^{\delta \cdot c} \tag{2.9}$$

where α, β, γ, and δ regression coefficients calculated in the job profiling phase and c is the residual capacity.

Finally, some form of prediction can be found in the systems designed for simulating MapReduce behavior that are discussed earlier in this chapter and in Section 2.3.2. Most of the simulators [34] [23] [52] [37] only carry out a simulation based on the job characteristics supplied in a manually-compiled configuration file that contains low-level statistics (such as CPU cycles and IO operations) gathered from a previous execution of the job on a cluster. The purpose of such simulators is not to estimate the job behavior, but to estimate the cluster's behavior given a specific job. Other simulators do not calculate runtimes at all, but rather replay past traces with a different scheduler. This is the case for Mumak [4] and SLS [6]. The only simulators that use dynamic job profiles to predict task durations are SimMR [49] and Starfish [27]. The former is based on the same profiler by Verma et al. [50] described earlier in this section. The latter uses an intricate mix of black-box (machine-learning based) and white-box (analytical) modelling detailed by Herodotou in his PhD thesis [25]. The prediction principle is the same, but the level of statistical detail and the composition of the models is more advanced. First, low-level statistics are gathered about the time cost of the different operations and phases of each task type, and the amount of data flowing through the different tasks and phases of the job execution. Then, regression models are compiled to extrapolate task behavior. These are later applied using the analytical data models to obtain concrete time estimates. This allows for a good estimation of task durations even after changing the underlying cluster structure and composition.

In conclusion, all MapReduce predictors follow the same general principle of uniformity among tasks of the same job (or even job type). However, among different prediction approaches, the level of detail at which each task type is deconstructed and modelled varies, from a simple average, to complex regression models.

3

Design of a Generic Simulator for Data Center-based MapReduce

This chapter explores the first research question (**RQ1**): How can predictions be made on how a sequence of MapReduce jobs will behave (with respect to a series of performance metrics) when executed on a homogeneous cluster under a specific scheduling policy? Our initial approach is to survey existing solutions for simulation of MapReduce workloads. However, we conclude that none of the existing simulators that we know of fulfill the entire set of requirements that a portfolio scheduler imposes. Therefore, we design our own simulator for data center-based MapReduce: POSim.

The rest of the chapter is structured as follows: we first formulate the requirements for the simulations that a portfolio scheduler needs to carry out to make informed policy application decisions (Section 3.1). In Section 3.2, we present the set of predictors that we designed, based on surveyed techniques, to provide the task runtime estimations needed by our proposed simulator. Section 3.3 illustrates the architecture and operation of POSim. In Section 3.4, we demonstrate how POSim can be integrated into the Hadoop YARN stack.

3.1. Requirements Analysis

Any portfolio scheduler needs to score its constituent policies according to a series of performance metrics in order to choose the one that could yield the best results for the targeted period of time. This is the purpose of its simulator. We identify the following requirements for this component:

R1 It should run the queued workload hypothetically, using each policy in the policy portfolio;

R2 It cannot be not omniscient: task behavior (i.e. runtime) is not known beforehand and should be predicted online;

R3 Simulations need to be as fast as possible, for the resulting policy change decision to still be relevant after the simulation has ended;

R4 It should be capable of calculating performance scores for the individual policy simulations and report the results;

R5 It should be openly available and adaptable to the target MapReduce stack (i.e. Hadoop YARN).

Section 2.3.2 contains a short survey of simulation techniques and current MapReduce simulators that were investigated for their fit with our proposed system. However, as Table 2.1 shows, none of them fully match our requirements. SLS [6], MRPerf [52] and MRSim [23] have very low-level resource models that result in heavy time costs, failing requirement **R3**. MRSG [34] and YARNsim [37] are designed to run a specific job based on a manually-constructed behavior configuration, and not an entire workload, invalidating **R1**. Mumak, SimMR and Starfish are not compatible with the YARN architecture and require upgrading, but only-Mumak is available for public use and modification, in accordance with **R1**. However, Mumak is not capable of running a new workload as **R2** demands. It only replays previous traces with a given scheduler. As a result,

we design and implement our own MapReduce simulator for use with the portfolio scheduler, the POrtfolio Simulator (POSim), based on techniques and results of previous work. The next sections follow that process.

3.2. Design of a Family of Task Behavior Predictors for Data Center-based MapReduce

In accordance with requirement **R2**, the portfolio simulator needs to predict future task behavior, not replay past traces. Section 2.3.3 surveys existing studies that analyze the prediction of task durations on MapReduce systems and proposes a family of task duration predictors to be experimented with in the portfolio scheduler. Section 3.3 details the design of a complete MapReduce simulator that matches the requirements mentioned above.

The prediction of the system's behavior at a certain time, given a certain load is a key component of any successful simulator. For the MapReduce context, this starts with being able to predict the duration of tasks in the system based on their characteristics and the state of the cluster. In this way, the long-term impact of early scheduling decisions can be better estimated.

Although tasks run the same map or reduce code, runtimes may vary due to the type and physical location of their input and the properties of the cluster. This is why this section contains an analysis of task duration variability, based both on research and on experiments carried out with our particular setup. The accumulated knowledge is used for the design and evaluation of a set of predictors that can be plugged into the system's simulator.

All the approaches mentioned in the previous section were taken into account for the design of the predictor. In order to better understand the effects of each added level of detail in the deconstruction of the execution patterns of MapReduce jobs, we have devised not just a single predictor, but three: a Basic Predictor, a Standard Predictor, and a Detailed Predictor. Each one builds on the previous and relaxes its assumptions through a more detailed time model. This approach also decouples the predictor from the rest of the portfolio scheduler and gives the end user better control over how much effort goes into the prediction, or whether to use their custom predictor to plug in.

The **Basic Predictor** is the closest version to the one used in the portfolio scheduler for scientific workflows [18], which calculated the average between the runtimes of the last two jobs run by the same user on the cluster. However, a few improvements were necessary to make the prediction more relevant for the MapReduce context. Our predictor calculates a task's runtime based on the average task 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" first and foremost if they have the same name, and only if no such jobs are found, then the system uses the jobs submitted by the same user. Since some MapReduce jobs are routine jobs that are run regularly, data about them is available and more relevant than that of other jobs sent by the user. The maximum number of past jobs that are used as reference is a configurable parameter in the system. The result is a predictor similar to what Polo et al. [42] use (Equation 2.2 applies).

The **Standard Predictor** refines the prediction by considering that past jobs might have different task durations because of the size of the data they had to process. Thus, historical *processing rates* are calculated and, in the case of map tasks, also their *selectivity*. The result is close to what Kc et al. [33] describe, without differentiating between the shuffle and reduce phases of reduce tasks. Moreover, historical reduce processing rates are used in favor of the job's current reduce processing rate, as the reduce phase can present greater variability than the map phase. The following equation applies:

$$r_i^M = \begin{cases} \mu_j^M & \text{if average map duration is known for this job j} \\ d^M \cdot \rho^{M_R} & \text{otherwise} \end{cases}, \quad r_i^R = f \cdot d^M \cdot \rho^R \tag{3.1}$$

, where

$$\begin{aligned}
\mu_j^M &= \text{average map task runtime of job j},\\
d^M &= \text{average map input data (split) size},\\
\rho^M &= \text{average data processing rate of similar completed map tasks},\\
f &= \text{map selectivity},\\
\rho^R &= \text{average data processing rate of completed reduce tasks},\\
r_i^M &= \text{runtime of task i of type map},\\
r_i^R &= \text{runtime of task i of type reduce},
\end{aligned} \tag{3.2}$$

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. 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.

The **Detailed Predictor** goes even deeper into the task internals and constructs a profile for each job that passes through the system. To factor in the observations of Ananthanarayanan et al. [1], 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, just as in the work of Verma et al. [50]. The resulting reduce estimation is similar to that in Equations 2.6 through 2.8, with the modification that merge time is considered and no regression is used (β disappears and α becomes the processing rate):

$$r_i^M = \begin{cases} d^M \cdot \rho^{M_L} & \text{if it is a local map task} \\ d^M \cdot \rho^{M_R} & \text{if it is a remote map task} \end{cases}, \tag{3.3}$$

$$r_i^R = \begin{cases} f \cdot d^M \cdot \rho^{R_s,T} + f \cdot d^M \cdot \rho^{R_m} + f \cdot d^M \cdot \rho^{R_r} & \text{if it is a typical reduce task} \\ t^{R_s,1} + f \cdot d^M \cdot \rho^{R_m} + f \cdot d^M \cdot \rho^{R_r} & \text{if it is a first wave reduce task} \end{cases}, \tag{3.4}$$

, where:

$$\begin{aligned}
\rho^{M_L} &= \text{average data processing rate of similar local map tasks},\\
\rho^{M_R} &= \text{average data processing rate of similar remote map tasks},\\
\rho^{R_s,T} &= \text{average data processing rate of similar typical reduce tasks},\\
\rho^{R_m} &= \text{average data processing rate of the merge step of a similar reduce task },\\
\rho^{R_r} &= \text{average data processing rate of the reduce step of a similar reduce task },\\
t^{R_s,1} &= \text{average duration of the shuffle in a similar first wave reduce task.}
\end{aligned}$$

None of the proposed predictors are designed for variation in the resource capability, as Zhang et al.[57] and Herodotou [25] account for. Since the cluster is considered homogeneous and its structure is stable, fluctuation in resources was not approached in this thesis. However, there can be contention on the disk or network due to tasks running on the same node that are IO intensive or have large data sets. Also, the cluster can be part of a multi-tenant setup where neighboring machines overuse sections of the network path between the MapReduce cluster's machines. We leave the exploration of these factors for future work. Another area of extension for this work regards using more complex prediction techniques that break down the

data read, data process, and data write phases of each task and run machine learning algorithms to model it. This approach would provide more accuracy for workloads containing jobs that do more data-invariant computation or that have runtimes which are dominated by output size instead of input size (data generators). However, it would come with the drawback of a higher time penalty and even impact data processing through the necessary instrumentation, as Herodotou et al. [26] reported.

Please refer to Section 5.4.1 for the experimental evaluation of all three predictor types.

3.3. Simulator Design

POSim is designed around the same message-based event handling mechanism that enables MapReduce frameworks, like Hadoop, to operate. Thus, the settings for heartbeat intervals and event queue usage come into play in the simulations the same way they would in real life. However, it can only be a discrete-event simulator, as real-time simulation is not an option for an online portfolio scheduler. For the same reason, resources are also not modeled explicitly to save computation time.

Simulations are triggered by sending a *start-simulation* message to the POSim process. The Simulator first copies the running job information from the main database view into a clone view. This way, changes in the cluster do not affect the simulations mid-run. Next, for each policy to be simulated, a Simulation Manager instance is created. This component is responsible for initializing its target policy and a "sandbox" environment for it, connected to a fresh database view. Each job that was running on the cluster is forcefully injected into the target policy, with the same state w.r.t running tasks (containers), remaining work and current resource requests. The remaining runtime of each task that was allocated resources is calculated using the configured predictor described in Section 3.2. The task is set to expire after this predicted interval, at which point a *task-finish* event will be triggered. The mock job implementation reacts to this by either requesting more resources from the target scheduling policy to run its remaining tasks, or signalling a *job-finish* event if there are none left. This process repeats until all the jobs in the queue have finished and the performance score can be computed from the information in the target policy's database view.

To implement this event-driven behavior, each main agent that is involved in the process of carrying out the submission, allocation and completion of tasks in the MapReduce cluster is identified and abstracted into logical components called *daemons*. Each daemon has a set of actions that it needs to do on startup, a set of actions that it carries out on each heartbeat, and a final set for shutting down and cleaning up its state. It is also associated an initial start time and a heartbeat interval, which it uses to keep track of the next moment it should be "woken up". All daemons are added to a run queue in ascending order of their next wake-up time, along with an additional Time Keeper daemon, responsible for keeping the current simulation time.

Figure 3.1 shows the operations that are carried out at each time step: the Time Keeper updates the simulation time to the closest wake-up time of the daemons that are waiting in the queue. All daemons that should be woken up at that time are triggered to run their step function and the Time Keeper waits for them to finish. During the execution of their step functions, daemons interact using an event queue that is monitored to trigger interactions with the database view of the current simulation. Thus, the data necessary for the operation of the scheduling policy and the final performance calculation is still recorded, even in the absence of a dedicated Cluster Monitor, keeping in line with **R4**. After completion of their step functions, each running daemon is re-added to the queue with an updated wake-up time. After all daemons have finished, the Time Keeper increments the simulation time to the wake-up time of the next daemon in the queue, and so on, until all applications in the simulation have finished their tasks.

Thus, the simulator mimics the same interactions that take place in the cluster, but within a fraction of the time, making it usable in online operation, like **R3** demands. It is left for future work to model more complex behavior, such as failures of tasks, daemons, or hardware.

3.4. Adapting the MapReduce Simulator to the Hadoop YARN Stack

To the fulfillment of requirement **R4**, the proposed simulation model fits perfectly with a system such as Hadoop, since each Node Manger and each Application Master can be abstracted into a daemon. The Node Manager daemons use the predictor to determine when their containers (tasks) should expire, while the Application Master daemons carry out the mock job logic of determining which tasks to run next and when to trigger the job-completion event. The Resource Manager is not a daemon in itself, but a stripped-down version of the same Resource Manager used by the cluster, with the sole responsibility of encapsulating the simulation's scheduling policy for the daemons to access.

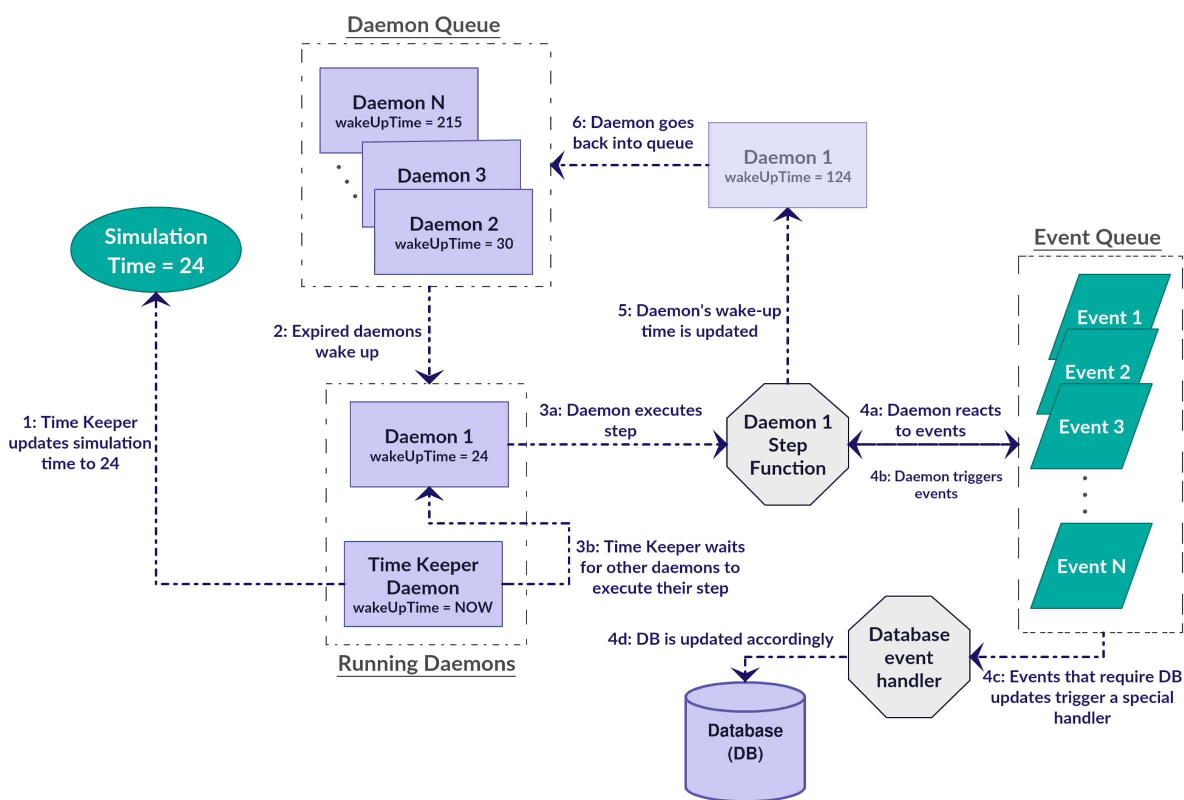


Figure 3.1: Simulator operation. Processing of one time step.

4

Design of a Generic Portfolio Scheduler for Data Center-based MapReduce

This chapter addresses the second research question (**RQ2**): how to design and develop a generic portfolio scheduler that supports MapReduce workloads? Our approach is to follow the same process of creating a portfolio scheduler illustrated in previous work on the subject, and adapt it to the MapReduce context, making sure to adhere to some pre-established requirements for our system. Our proposed solution is POSUM.

The remainder of this chapter is structured as follows: we start by formulating the requirements for the scheduler design (Section 4.1). Section 4.2 introduces POSUM and provides a broad overview of its architecture. We then describe the necessary adaptations made for the generic portfolio scheduler design process covered in previous work to apply to our target MapReduce environment (Section 4.3). The next section, 4.4, explains the specific components that make up POSUM and how they interact. The final section, 4.5, describes how POSUM offers monitoring and tuning capabilities to cluster administrators.

4.1. Requirements Analysis

The design process of the portfolio scheduler presented in this theses started out with several goals in mind:

R1 Include necessary conceptual elements of portfolio scheduling found in previous work;

R2 Take into account the characteristics of MapReduce processing;

R3 Adapt the resulting scheduler architecture to an actual, widely used MapReduce framework;

R4 Keep the design flexible so as to explore and compare different approaches;

R5 Follow or compare to state-of-the-art techniques wherever possible.

The first and second goals are demanded by our mission statement: applying portfolio scheduling to MapReduce workloads. Adapting the scheduler to an actual framework allows us to evaluate it on a real system, while also tackling more practical aspects that a high-level design might overlook. However, flexibility (**R4**) is a must if the design should also be applicable to other frameworks. It also eases maintainability and further improvement. The last goal refers to incorporating existing solutions if applicable. The benefits are two-fold: we take advantage of the lessons learned by other researchers in the domain and can also better compare to their work and results.

Addressing these requirements, we design POSUM (POrtfolio SchedUler for Mapreduce), an online meta-scheduler that can switch scheduling policies at runtime, based on real-time policy performance evaluations.

4.2. Architectural Overview of POSUM

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 **R3**, our implementation was integrated with the Hadoop (YARN) stack. More details about this can be found in Section 5.2.

The system follows the high-level architecture illustrated in Figure 4.1. There are three main processes that constitute POSUM. The Data Master is responsible for monitoring the system (both POSUM and the target cluster) and providing a coherent view of the stored statistics to the other processes. The POSUM Master is the orchestrator of the system. It coordinates simulations and decides when and how to change scheduling policies. The Simulator Master is a loosely coupled component that can simulate the outcome of a scheduling policy, given a certain queue composition, cluster state, and previous runtime statistics. It contains behavior predictors covered in Chapter 3. The scheduling policy is applied by the Portfolio Meta-Scheduler which is not necessarily a process on its own, but gets loaded as the designated scheduler of the framework, as is the Hadoop model.

Keeping the architecture modular achieves separation of concerns and enables both flexibility in approach exploration (demanded by **R4**) 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.

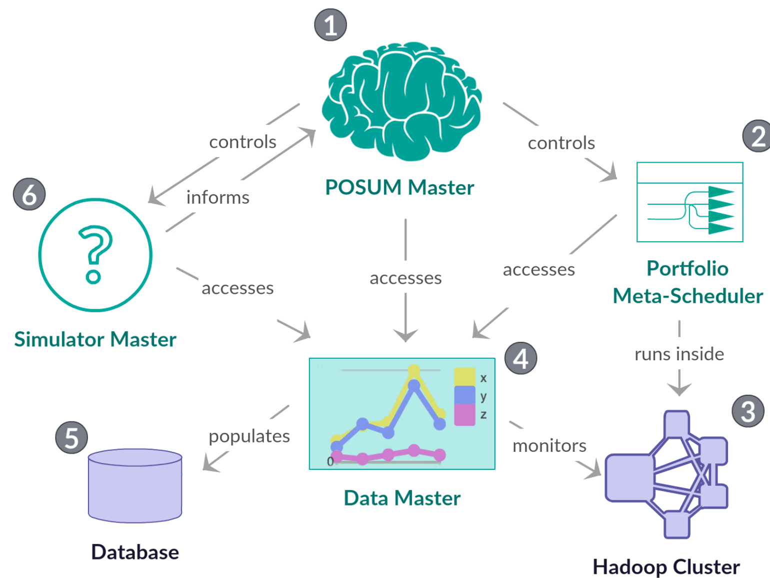


Figure 4.1: POSUM overview. We present the details of the architecture in 4.4

4.3. Portfolio Scheduler Adapted to MapReduce Workloads

To be in line with **R1**, we worked on the model constructed by Deng et al. [19], the design follows four steps, illustrated in Figure 4.2: creation (putting together the policy portfolio), selection (the process of scoring and selecting the appropriate policy for the next period), application (switching to the selected policy and monitoring its performance), and reflection (using the collected feedback to modify system configurations). Our new design allows for parameter variations within different steps.

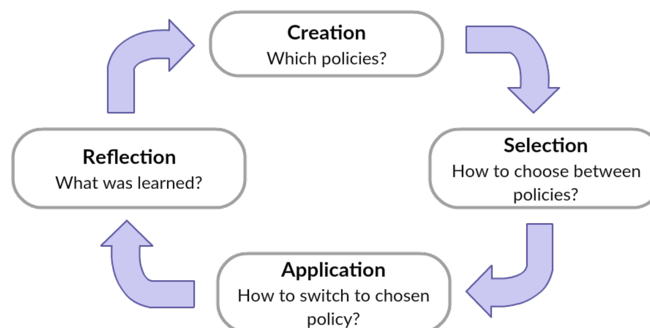


Figure 4.2: The process of designing a portfolio scheduler.

4.3.1. Portfolio Creation

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. This selection can be done by an expert, or through an automated process such as a comparative study of policies specific to one domain. 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 2.1 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 4.1 contains a short description of each of them. The survey in Section 2.3.1 served as inspiration for the policies of the first type. For deadline constrained (DC) jobs, the baseline policy is usually Earliest-Deadline-First (EDF) in literature [20, 33, 42]. 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 [18]. However, a form of prioritisation needs to be established between the two categories as well. Two solutions were adopted for this, resulting in the following pair of policies:

1. EDLS-Sh: 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;
2. EDLS-Pr: 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 $(1-\delta)$.

The δ parameter is configurable in the system (between 0 and 1) and represents the importance of DC jobs for the cluster owner.

These 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:

1. SRTF: the share-based version of the Shortest-Remaining-Time-First scheduler described by Cheriere et al. [15];
2. LOCF: a FIFO scheduler that enforces locality along the lines of what Zaharia et al. [56] and He et al. [24] implemented.

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 **R2**).

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.

4.3.2. Policy Selection

Achieving better performance over any single policy is complex, as the scheduler needs to switch policy if and only if the new one achieves better performance for the considered time period. The simulator detailed in Chapter 3 is used to anticipate how the system will operate under each scheduling method and output a score based on the metrics that the system is meant to optimize for.

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 $(1-\delta)$
SRTF	allocation	the share-based version of the Shortest-Remaining-Time-First scheduler described by Cheriere et al. [15]
LOCF	allocation	a FIFO scheduler that enforces locality along the lines of what Zaharia et al. [56] and He et al. [24] 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 4.1: A description of the policies used by POSUM.

The simulator caters to **R2**, **R4**, and **R5**. It follows the state-of-the-art closely with regard to the way it models the components of the system and how prediction is implemented. Its predictor is a decoupled component whose internal algorithm can be tailored to the cluster owner's needs. This flexibility is leveraged by varying the implementation of the predictor, as described in Chapter 3. Each implementation goes even deeper into MapReduce specifics to achieve its prediction.

4.3.3. Policy Application

Policy application starts immediately after the a policy has been selected. It consists not only of changing the way scheduling decisions are made (to include a proxy that delegates decisions to the current algorithm), but also of monitoring the performance of the policy, as opposed to what was predicted, for improving future scoring decisions (see next section).

For full flexibility to be achieved in policy application, some modifications might need to be made on how the policies operate in order for them to know more about the runtime state of the cluster, to use in their decisions. For a YARN setup, for instance, schedulers receive a request with the resources that an application needs and should base their prioritization solely on what queue the application is in, and which user submitted it. Giving the Portfolio Meta-Scheduler access to the Data Master makes it possible for more detailed information to be queried, even system-wide information. So policies can even be reactive to aspects that do not concern the applications themselves. Although this is a powerful concept, in this thesis we restrict the complexity of the scheduling policies so as to better analyze how the constant policy switch affects performance. The policies thus pull information only about the queued applications and use it to calculate slowdowns and remaining runtimes.

4.3.4. Reflection

Reflection on the performance of POSUM is carried out by a Feedback Module. Several features can be enabled and tuned for this component:

- Adjustment of the simulation interval (the time between consecutive simulations) to include burst or change-point detection according to the characteristics of the jobs/tasks that are currently in the queue;
- Interruption of an ongoing simulation when the change in properties of the system/ queue make-up have rendered it irrelevant ;
- Automatic policy application if a certain pattern of queued job types is detected and there are records of a particularly successful policy for that pattern;

- Analysis of the difference between simulated and observed cluster behavior as feedback for future improvement of POSUM components;

When analyzing the type of system load or queue make-up, the following dimensions are checked for patterns (based on the research of Chen et al. [13]):

- Distribution of job input/output/shuffle data sizes;
- Distribution of ratios of sizes;
- Distribution of map/reduce task runtimes;
- Job arrival (measured as averages of data sizes or number of jobs, peak to average ratios, diurnal patterns, etc.);
- Deadline constraints relative to data sizes.

However, the exact operational design of the Feedback Module is left for future work.

4.4. Key Processes and Components in POSUM

Considering the particular requirements for the system, there are three main processes that form the POSUM system, as illustrated in Figure 4.3: the POSUM Master (PM), the Data Master (DM), and the Simulator Master (SM). This separation was chosen as a means of isolating functionality and limiting the interference among the system's components. For instance, workload simulation can be very costly in terms of computation and memory and may introduce delays in the monitoring process of the DM. Conversely, the DM gathers/feeds large amounts of data from/to other system components, consuming network bandwidth. Since they are loosely coupled, these processes can be distributed to separate computation nodes if necessary and can be easily bypassed or replaced (e.g., upgrading the simulator to a more accurate one).

In our evaluation setup, the processes run together on their own node in the cluster, so as not to hinder the cluster's behavior. POSUM only interfaces directly with the MapReduce engine when gathering status data (e.g. via the exposed YARN REST API), and when a new node is added to the cluster and the resource manager needs to be informed and reconfigured to make use of it. In addiction, the scheduler that is loaded by the cluster is in fact a meta-scheduler that interacts with other POSUM processes and is devised to delegate all scheduling decisions to the current chosen scheduling policy.

The POSUM daemons communicate through an RPC mechanism. Each process starts its own server and exposes an interface through which its services can be invoked remotely. The interface exposed by the PM (POSUM Master Protocol) is used by all the components of POSUM, especially at startup, in order to register their services with the PM and become usable in the system. The Data Master Protocol is also used by all the components in order to access runtime or historical statistics.

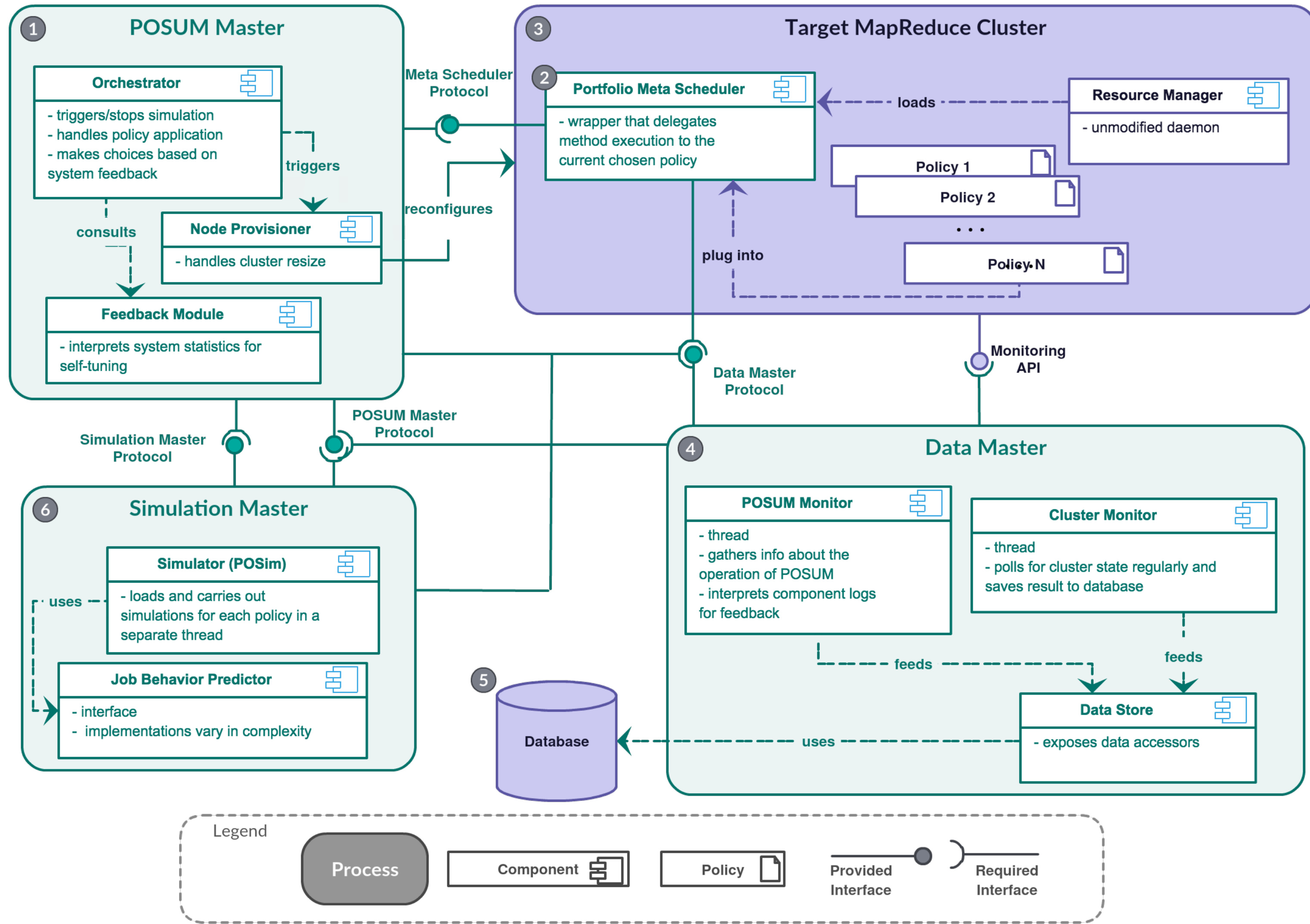


Figure 4.3: POSUM architecture.

The command center of POSUM is the **POSUM Master** process. All decisions regarding POSUM's operations are made by the *Orchestrator* component within. 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. In order to apply provisioning decisions, it uses the *Node Provisioner*, a component capable of reconfiguring the cluster to make use of or free recently added nodes. For analyzing performance and adapting to environmental changes, the *Feedback Module* regularly interprets system statistics and initiates self-tuning.

The **Simulation Master** process starts to simulate all the available scheduling policies concurrently, at the Orchestrator's command. The implementation of the *Simulator* component was already discussed in Chapter 3. The prediction logic used is contained under the umbrella of the *Job Behavior Predictor* component and its implementation is chosen according to the configuration file. Upon completion, the SM uses the POSUM Master Protocol to send the scores to the PM. It is the Orchestrator that makes the final policy decision.

The **Data Master** process stores all the data gathered by the two monitors of the system in a database. It consists of three main components: the Data Store, the Cluster Monitor, and the POSUM Monitor. The *Data Store* offers uniform and audited database access to all processes that require it. The *Cluster Monitor* component makes regular checks to get real-time information about applications and tasks that are running, or have finished running on the cluster. This information can be used in predictions and by schedulers that need constant updates on application progress for making decisions. The *POSUM Monitor* component, 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. It also interprets the component operation logs to supply information to the Feedback Module and the web interface. For instance, the Portfolio Meta-Scheduler logs each change in policy in the Data Store, but the calculations on overall policy durations are done periodically by this monitor.

The **Portfolio Meta-Scheduler** is only 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.

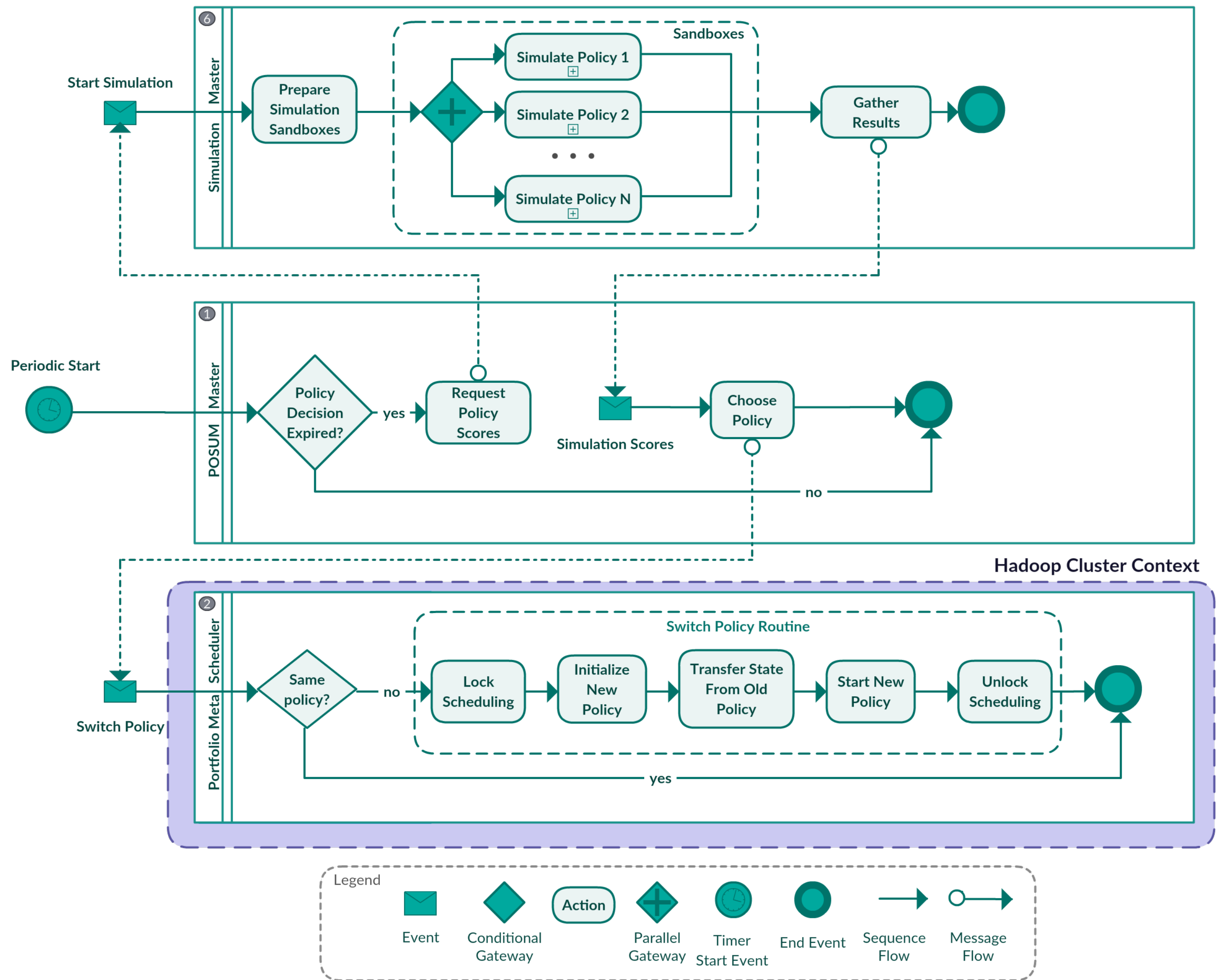


Figure 4.4: The process of switching policies.

Figure 4.4 shows the policy change process. The PM regularly checks if the conditions are met for the policy to expire and require a change decision. The PM sends a message to the SM to start simulating each policy in parallel and send back the resulting scores. The PM then sends a "Switch Policy" request to the Portfolio Meta-Scheduler. If the policy is different from the current one, the scheduler initializes the new policy, transfers the state (running applications, containers, resource consumption, etc.) from the old policy and starts with the new one. While this process takes place, all scheduling requests are forced to wait on a lock so as to avoid transition failures and maintain consistency.

If it is the case that the framework operation model abstracts all job-specific data away from the system scheduler (as is the case of YARN), policies that rely on these characteristics (for instance, a Smallest-Job-First or Earliest-Deadline-First algorithm) need to be implemented to access the running applications' profiles, which are stored in the database. This is done through the Data Master Protocol client that is made available by the Meta-Scheduler to each child policy.

4.5. The Administrator Dashboard

Although POSUM is autonomous in its policy application and self-management, it does need to be initially configured by system administrators, and eventually monitored during operation. For this, we believe it is of great value to provide a graphical interface for administration. The *POSUM Admin*, is a web application designed to offer visual real-time system information as well as tuning capability to administrators. It features several sections, including but not limited to:

1. Cluster statistics: the evolution of the number of running applications and containers and the node resource both free and in use;
2. Scheduler statistics: policy choices over time, aggregated policy usage statistics, and scheduler operation time costs;
3. System: memory and CPU usage of the POSUM processes and their health status;
4. Performance: the evolution of the performance metrics;
5. Controls: administrator tuning controls such as changing the weight of the individual metrics in the performance calculation, archiving old runtime data, refreshing the prediction model, and even a manual override of the current scheduler;
6. Analysis: a section with administrator tools for exploring the insights gathered by the Feedback Module and for creating new data and performance visualizations.

POSUM already monitors both the cluster and itself. General statistics are recorded in the database by default, but fine-grained reporting can also be enabled via a configuration parameter. The *POSUM Monitor* regularly logs these statistics to the database if persistence is enabled. Deliberate state changes (such as policy changes) are logged proactively by each process to the database. In this case, the POSUM Monitor acts as an interpreter and combines the data into meaningful reports for both the admin application and the Feedback Module.

To enable both the POSUM Monitor and the admin app to access process-specific information, all POSUM processes expose RESTful services of their own and offer on-demand real-time statistics such as JVM memory usage or the time cost of certain operations. These endpoints are polled regularly in the graphical user interface exposed by the POSUM Master and combined into one dashboard.

Section 5.3.4 contains an example implementation of this interface.

5

Experimental Evaluation of the Portfolio Scheduler

This chapter addresses the third research question (**RQ3**): how to demonstrate the capabilities and evaluate the performance of the proposed portfolio scheduler in realistic situations? In particular, does the portfolio scheduler perform better than its constituent policies? The scope of our empirical analysis is five-fold. First, we evaluate the accuracy of the three predictors. Second, we perform an analysis of the simulator. Third, we present a qualitative performance comparison between dynamic policy application (i.e., portfolio scheduling) and running each policy of the portfolio policies separately. Finally, we look at the stability of the system over multiple runs.

The main findings of our empirical analysis are:

MF1 Our predictors are efficient in estimating the runtime of both map and reduce tasks. However, due to the variability exhibited by the real-world workloads in our setup, reduces are more difficult to predict;

MF2 The simulator is roughly accurate when forecasting tasks which do not exhibit variability (e.g., maps), but the quality of the simulation degrades for reduce tasks which exhibit variability. Also, simulations and policy changes add little overhead to the system's functioning;

MF3 Portfolio scheduling cannot outperform policies that target single performance objectives. However, it balances mixed performance goals well, even in the absence of accurate task runtime predictions;

MF4 POSUM is light on resource usage: CPU load is generally below 40%, while memory utilization is not significant for modern hardware. Also, performance results fluctuate more than in the case of individual policies, as scheduling possibilities increase, but they do converge to a single value.

The remainder of this chapter has the following structure: we start by establishing the requirements for our evaluation in Section 5.1. We then define setup and objective of our evaluation in Section 5.2: system setup, workloads, and performance metrics. We then implement POSUM as a standalone module integrated into the Hadoop YARN stack (Section 5.3). Finally, we present the results for each type of analysis in Section 5.4.

5.1. Requirements Analysis

We now have a complete design of the proposed portfolio scheduler. In order to evaluate its performance and analyze its behavior, we have to design a clear experimental setup, i.e.:

R1 Set up the physical environment and the software structures that support the experiments;

R2 Define the workloads that will be used: types of jobs, arrival patterns, SLOs;

R3 Define the performance metrics that will be used both to score individual policies after simulation and to assess the POSUM's performance;

R4 Design a set of experiments for evaluating prediction accuracy and compare the three predictors detailed in 3.2;

R4 Design a set of experiments for validating the operation of the POSim simulator described in 3.3;

R5 Design a set of experiments for comparing the performance of POSUM with that of its constituent policies;

R6 Monitor POSUM and record operational data for reporting resource utilization, overhead, etc.

5.2. Experimental Setup

The environment is set up to coincide with the system model portrayed in 2.1. The workloads, composed of batch and deadline constrained jobs, are generated according to common practices in related literature and are run using a dedicated benchmarking solution. The performance goal is composed of two sub-goals, one for each job type. More details follow.

5.2.1. Hardware and Software Environment

All experiments are run on the TUDelft site of the DAS-5 [1]. 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 standard 1 Gbit/s Ethernet. In our experiments we used no virtualization and leased 1 node for running the YARN master, 1 node for running the three standalone POSUM processes and 8 nodes as YARN slaves. Each slave runs a maximum of 16 containers having 3 GB memory and 1 CPU core each.

5.2.2. Workload Generation

As production trace data is often lacking in real-time system information (node topology, data placement, bandwidth, disk speed, failures), the experiments are run on a synthetic workload with the help of a micro-benchmarking tool called BigDataBench [2] [53]. This suite was inspired by HiBench[29], but where HiBench can only run single application-type workloads, this tool makes it possible to prepare the data for and run simultaneously any combination of application types and data sizes, each with their own input and output directories.

The specific application types and their descriptions can be found in Table 5.1. The Sort and WordCount applications come with the Hadoop distribution and are representative of a large subset of real-world MapReduce jobs. The first does not modify the data itself, but only passes it through the system to be sorted and/or transformed from one representation to another. WordCount only counts the occurrences of words in texts, but it is analogous to any application that would parse large input data to extract a small amount of interesting information.

Application	CPU Utilization	I/O Utilization	Data Sizes
Sort	Low	High	8 - 56 GB
WordCount	High	Low	2 - 35 GB
Nutch Indexing	High	Medium	50K - 2M websites, 300 MB - 12 GB
Naive Bayes Classification	Low	High	2 - 18 GB

Table 5.1: Workload application types.

The Nutch Indexing application leverages the indexing component of the popular Apache Nutch [3] open-source search engine. We use Hibench's data generation tool to create the necessary site data. The map function of NutchIndexing is simply an identity function, and the reduce function generates inverted index files from the input texts.

[1] http://www.cs.vu.nl/das5/
[2] http://prof.ict.ac.cn/BigDataBench/
[3] http://lucene.apache.org/nutch

The Naive Bayes Classification application applies a previously created Naive Bayes data mining model [4] to a set of generated input data to classify it into one of two categories. The phases related to training the model are not included in our workloads, as job workflows are out of the scope of this study.

The engine for creating workloads and job deadlines, generating the input for them separately, running the jobs, and gathering the results is custom built [5]. Every workload is generated as a uniform combination of these application types, but with a data size factor drawn from an exponential distribution so as to favor smaller jobs over larger ones, as observed by [13]. Job arrival intervals are chosen from a Poisson distribution to be around a fixed value. This value, along with the rate of the size distribution determine the load that is put on the cluster. We choose a configuration that translates to a maximum of 10 to 20 jobs being in the system queue at any given time. The same workload can be re-run multiple times, as its definition is serialized to a file. Input data can be either be re-generated for a job or re-used based on the type and size of a job.

Deadlines are generated using a technique from literature [42] [36], i.e. the time it takes to run an application with the same data size alone on the cluster, multiplied by a random relaxation factor between 1 and 3. The proportion of deadline constrained jobs is fixed at an arbitrary 30%. However, deadlines are not directly supported in Hadoop. We implement two ways of specifying one: either as a job configuration parameter, or in a separate database collection. Since the jobs in our workloads come from closed executables in the benchmarking framework, we use the second option, the database, to communicate deadlines between the workload execution engine and POSUM.

5.2.3. Performance Objectives

In accordance with the problem description in Section 1.1, we choose the performance objective to be a combination of two sub-objectives: the accumulated SLO violation penalties and the total batch job slowdown. Operation cost was left out as cluster resizing was not implemented.

Since the the metrics operate in different value ranges and are difficult to normalize, the aggregated performance score will not be a scalar value, but a two-dimensional vector:

$$P = (V_D, S_B)$$

, where:

$$V_D = \frac{\sum_{i \in D} (max(0, r_i - d_i))^2}{|D|}, \tag{5.1}$$

$$S_B = \sum_{i \in B} \frac{r_i}{max(e_i, MinE)}, \tag{5.2}$$

$$D = \text{set of deadline-constrained (real-time) jobs in workload}, \tag{5.3}$$

$$B = \text{set of batch jobs in workload}, \tag{5.4}$$

$$r_i = \text{runtime of job i}, \tag{5.5}$$

$$e_i = \text{total execution time of the tasks of job i}, \tag{5.6}$$

$$MinE = \text{lower bound on the possible execution time of jobs}, \tag{5.7}$$

$$d_i = \text{desired runtime of job i (from submission to its deadline)}, \tag{5.8}$$

$$\tag{5.9}$$

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 any two performance vectors. To do this, and also provide absolute score values for easier plotting and comparison, we combine individual metric scores following formula:

$$P' = \alpha V_D + \beta S_B$$

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

[4] https://mahout.apache.org/users/classification/bayesian.html

[5] https://atlarge.ewi.tudelft.nl/gitlab/m.voinea/thesis-tools

5.3. Implementation of the POSUM Prototype

For the purpose of evaluation, POSUM is implemented in around 70,000 lines of code and integrated into the Hadoop (YARN) 2.7.1 stack through a GitHub repository fork [6]. It resides in the repository as a separate tool which can be run alongside the Hadoop cluster. With the exception of a few bug fixes, the rest of the original sources are not modified.

5.3.1. Processes and Communication

The POSUM daemons communicate following Hadoop's RPC model based on Google's Protocol Buffers [7].

In accordance with the Hadoop framework, the operating parameters of POSUM can be configured using an XML configuration file, and a script is supplied with the module in order to automatically deploy and destroy the necessary daemons.

Since the YARN operation model abstracts all job-specific data away from the system scheduler, policies that rely on these characteristics (for instance, a Smallest-Job-First or Earliest-Deadline-First algorithm) need to be implemented to access the running jobs' profiles, which are stored in the database. This is done through the Data Master Protocol client that is made available by the Meta-Scheduler to each child policy.

These profiles are put together by the Cluster Monitor component with information gathered through Hadoop's RESTful API. For those jobs that are not yet tracked by the Application Master, the job configuration directory is parsed directly from the HDFS, meaning that the information is always available in case a scheduling decision needs to be made. The same information is required by the simulator in order to stage and execute simulations based on the real-time data of jobs that are waiting in the queue.

For storage, a NoSQL solution, MongoDB, is chosen for its flexibility during development and its speed in updating and querying large volumes of data [39]. Since this system does not implement transactions out-of-the-box, the Data Store component acts as a transaction manager that can receive a set of operations through an RPC call and block all other access during execution. To restrict wait time considering this model, the database is split up into separate views that can be locked independently. So, for instance, simulations use a separate MongoDB database definition than the scheduler and monitors do. These views translate so separate MongoDB "database" definitions.

5.3.2. Policies

SRTF and LOCF are based on Hadoop's FIFO scheduler, rewritten to be pluggable into the Meta Scheduler. There is a single queue of jobs, but their priority depends on the policy logic. SRTF is built using the principle of the hSRTF policy designed by Cheriere et al. [15], which attempts to keep the ratio between the number of compute slots assigned to any two jobs equal to the inverted ratio of their estimated remaining work. It is common for many scheduler designs to be underspecified, that is, the technical specification of algorithms and formulas necessary to reproduce the scheduler behavior are lacking to the point where the expert needs some guesswork in trying to recreate the original scheduler [3]. What follows is an example of trying to recreate the hSRTF policy based on the formulas provided in the article. We start by considering a reference job 0 having remaining work w_0^r and allocated number of slots s_0:

$$\frac{s_i}{s_0} = \frac{w_0^r}{w_i^r}, \forall \text{ job } i, 1 < i < m, \text{ in the queue of } m \text{ jobs}$$

To determine an absolute formula for the desired number of slots each job should be allocated (s_i') we use the above dependency and sum up all the slots needed for each job, which should amount to the total number of slots in the system, denoted n:

$$\begin{aligned} n &= s_1' + s_2' + ... + s_m' \\ &= \frac{w_0^r}{w_1^r} \cdot s_0 + \frac{w_0^r}{w_2^r} \cdot s_0 + ... + \frac{w_0^r}{w_m^r} \cdot s_0 \\ &= s_0 \cdot w_0^r \cdot (\frac{1}{w_1^r} + \frac{1}{w_2^r} + ... + \frac{1}{w_m^r}) \end{aligned}$$

[6] https://atlarge.ewi.tudelft.nl/gitlab/m.voinea/hadoop
[7] https://developers.google.com/protocol-buffers

This means that s_0 can be expressed as:

$$s_0 = \frac{n}{w_0^r \cdot N}, \text{ where normalizer } N = \sum_{i=1..m} \frac{1}{w_i^r}.$$

Thus, s_i can be obtained as:

$$\begin{aligned} s_i' &= \frac{w_0^r}{w_i^r} \cdot s_0 \\ &= \frac{w_0^r}{w_i^r} \cdot \frac{n}{w_0^r \cdot N} \\ &= \frac{n}{w_i^r \cdot N} \end{aligned}$$

Cheriere et al. use an additional factor to avoid starving long-running jobs of resources in favor of short ones. This is equal to the ratio between the estimated final runtime of a job given its current resource allocation (s_i) and what its runtime would be if it were running alone on the cluster:

$$\sigma_i = \frac{t_i^e + \frac{w_i^r}{s_i^a}}{\frac{w_i^t}{n}}, \text{ where } t_i^e \text{ is the job's elapsed runtime and } w_i^t \text{ is its total work.}$$

Finally, we consider the priority of each job in the queue to be its resource deficit (Δ_i), i.e. the difference between the number of slots that the job is currently allocated (s_i) and the desired number of slots (s_i'), accounting for starvation:

$$\Delta_i = s_i - s_i' \cdot \sigma_i$$

By job work, we mean the cumulated execution time of all the job's tasks. We consider maps and reduces separately and multiply the number of tasks of each type by the average task duration for that task type. If no reduces have finished yet, we calculate the average map processing rate and multiply it with the estimated reduce input size to obtain the reduce task runtime estimation, similar to Equation 3.1 in Section 3.2.

The LOCF policy does not calculate priorities for jobs per say. When a Node Manager signals that it has available slots, the queue of jobs is traversed in the order of their arrival until one is found that has tasks local to that node. A job can only be skipped a number of times equal to the number of nodes in the cluster, to avoid starvation of jobs.

The EDLS-Sh and EDLS-Pr policies are based on Hadoop's Capacity Scheduler, orchestrated through Java Reflection API to comply with and be manipulated by the Meta Scheduler. The Capacity Scheduler allows for the definition of a hierarchy of queues, where each queue can be allocated a specific proportion of the cluster's resources, called the *capacity* of the queue. The sum of the capacities of a parent queue's children is always equal to that of the parent queue. For both EDLS policies, we define two children of the root queue: one queue for deadline-constrained jobs, having capacity δ), and the other for batch jobs, with capacity $1-\delta$. The δ parameter is configurable, but in our experiments it is fixed at 70%.

The difference between the two policies lies in the mechanism of prioritizing the queues. For EDLS-Sh, the default implementation of the Capacity Scheduler is used, which is based on resource deficit calculations. EDLS-Pr, on the other hand overrides this default sorting algorithm of the queues. Instead, it chooses the next random number between 0 and 1 from a uniform distribution and puts the DC queue first if the number is smaller than δ, and the BC queue first otherwise.

Besides the implementation of each scheduling algorithm listed above, we implement the framework that permits switching policies in a single-scheduler system like Hadoop. The solution is based on a pseudo-scheduler, the Meta Scheduler, which does not keep any state related to jobs or queues or resources. Its purpose is to provide the mechanism of instantiating a sub-policy, transferring state between policies of different types, and delegating scheduling decisions to the currently active sub-policy, without allowing inconsistencies in data or loss of events. The portfolio of policies provided with POSUM is extensible and fully configurable, as long as the all policy classes implement the same Plugin Policy interface and can import and export their state for other policies to take over.

5.3.3. Simulator

The simulator is be based on an existing simulator for MapReduce workloads, the Hadoop Scheduler Load Simulator [8] but with the modification that it does not already know a job's runtime and resource consumption, but tries to predict it, according to the results of research question 1. For this, the implementation of the Node Manager that is used in simulations is changed to not rely on a previously established duration for every container, but rather operate with an instance of the configured predictor.

Another drawback of SLS that POSUM mitigates is the time needed to run a simulation. The original simulator implements event-based communication, in the sense that the simulated Node Managers and Application Masters (called daemons) rely on the same events to communicate with the Resource Scheduler as they would on the real cluster. However, the simulation is fixed-step, since they are implemented as schedulable recurrent tasks that wait in a queue to be executed until enough real system time elapses for their heartbeats to expire. This causes an hour-long simulation to need an hour to run. Since our simulator is online, it needs to make decisions in a matter of seconds, not in real cluster time.

To transform SLS into an event-based simulator, we make two major changes: we force the expiration of each daemon to be triggered based an artificial timer, and we create a new type of schedulable task: the Time Keeper. As depicted in Figure 3.1 of Section 3.3, this daemon increments the artificial timer to the timestamp of the next event after the other daemons' interactions in each time step are over. For this, the default execution queue of SLS is augmented to provide fine-grained synchronization with the Time Keeper. Also, a real event queue is created for interaction between Node Managers and Application Master, as opposed to the direct invocation implemented in SLS. This allows us to monitor the exchanges for the purpose of updating the system state in the database view of the current simulation. Without this feedback, runtime predictions would be incorrect and we would not be able to compute evaluation scores.

Both the simulator's predictors and the simulator itself could be interchanged with other implementations as they are loosely coupled in POSUM.

5.3.4. Monitoring

The POSUM statistics web application (Figure 5.1) is a browser interface designed to offer visual real-time system information as well as tuning capability to administrators. It is implemented as a simple Bootstrap-enabled landing page, with several tabs containing dashboards. It uses POSUM's REST API to get information and the Plotly [9] Javascript library for plotting diagrams. This library offers out-of-the-box manipulation capabilities for the diagrams, like enabling and disabling traces, zooming and panning, even exporting to the Plotly platform for advanced data manipulation. Refer to Appendix B for screenshots of each tab. Most of the functionality described in Section 5.3.4 is already available. However, the analytics tab is currently only used in development mode, for manually constructing overviews.

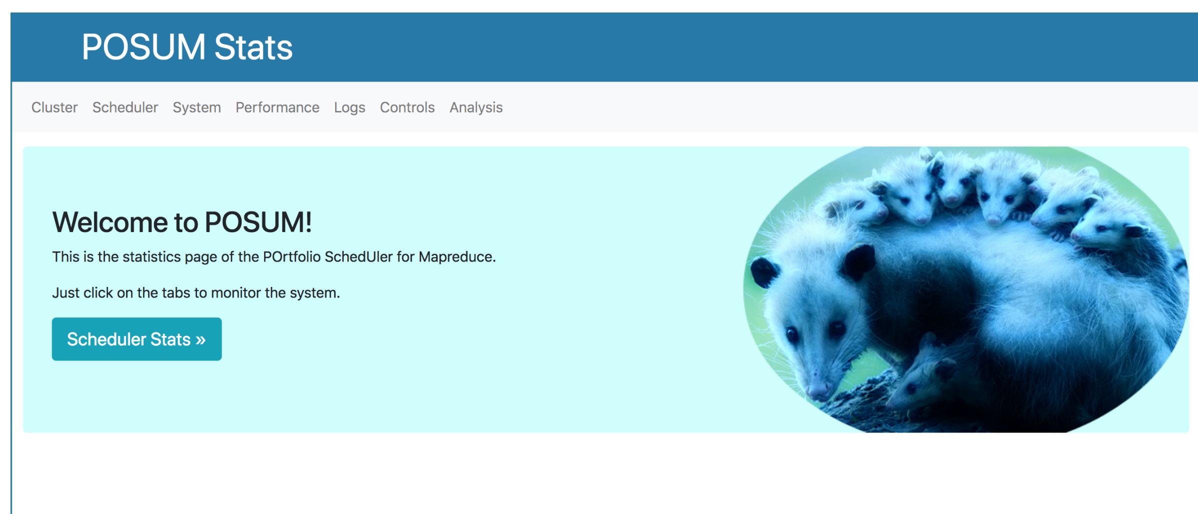


Figure 5.1: The POSUM web application.

[8] https://hadoop.apache.org/docs/r2.4.1/hadoop-sls/SchedulerLoadSimulator.html
[9] https://plot.ly/javascript/

Predictor	Task Type	Average Error (Seconds)	Average Relative Error (Error/Runtime)
Basic	Map	32	0.911
Standard	Map	34	0.698
Detailed	Map	35	0.570
Detailed	Local Map	47	0.696
Detailed	Remote Map	32	0.663
Basic	Reduce	248	1.845
Standard	Reduce	214	0.999
Detailed	Reduce	1501	4.569

Table 5.2: Average prediction accuracy for initial experiments (3 hour-long runs).

5.4. Experimental Results

The experiments are targeted at evaluating three main behavioral aspects of our implemented design: task runtime prediction accuracy, system performance compared to using any single scheduler, and system stability.

5.4.1. Prediction Accuracy

For setting up these experiments, we need to cater to the fact that predictions evolve along with the historical and job runtime data that they input into their model. Thus, the best way to monitor the accuracy of simulations during the operation of POSUM is to run each predictor alongside the system and record predictions periodically, using actual cluster runtime data. For this purpose, the POSUM Monitor's periodic data collection function is modified to also fetch each task currently waiting or running in the system, predict its runtime using each of the three predictors detailed in 3.2, and persist the results to the database.

We run the experiments in two phases. An small set of prediction experiments are first run for the purpose of choosing the best predictor to use in the performance experiments that are illustrated in the next section. Then, we use the same setup while carrying out the performance experiments themselves, to constantly monitor the accuracy of the predictions that the simulator uses in its operation, and compare them to the results that the other predictors would obtain on the same data.

The **initial set of experiments** consists of three medium workloads with hour-long job arrivals. The results can be seen in Table 5.2. We average both the total error and the relative error (error divided by the runtime of the task) of each predictor over all the tasks of the same type in all three workloads.

Several preliminary findings derive from analyzing the results. Firstly, considering map locality when predicting map task runtimes does not add value in our setup. This presents in a lower prediction error when using the detailed predictor without specifying task locality (detailed map in Table 5.2). Although the average runtime of the local tasks of a job is generally lower than that of the remote tasks, the difference is only a few seconds and is outweighed by the variability in runtime caused by other factors in the system. For local tasks, higher runtimes could be observed when many data-local tasks were scheduled simultaneously on the same node, which can cause thrashing on the disk. For remote tasks, varying network load when scheduling several remote tasks on the same network segment could cause the same effect.

However, a slight improvement can still be seen between the detailed predictor and standard predictor for map predictions. Checking Equation 3.1, we see that the standard predictor defaults to the average map duration if at least one map task was complete for a given job, on the assumption that map task input sizes are generally uniform. However, it seems this assumption is false. Calculating input processing rates always adds value for map tasks.

Another observation is that reduce tasks are even more susceptible to runtime variability. Even splitting prediction on different phases of the reduce process is not enough. The average relative error of detailed reduce predictions is lower than that of the standard ones for two out of the three workloads. However, the third workload generated some abnormal predictions, resulting a higher overall error for the detailed predictor.

With these points in mind, the performance experiments featured in Section 5.4.3 are carried out using the standard predictor, as it gives more consistent results for reduce predictions. However, a slight modification is made to it for map predictions: it always calculates runtimes based on the exact input each tasks needs to process and the previous map processing rate for the job (or similar past jobs if unavailable). Also, to

Predictor	Task Type	Average Error (Seconds)	Average Relative Error (Error/Runtime)
Basic	Map	5	0.679
Standard	Map	4	0.350
Detailed	Map	4	0.350
Detailed	Local Map	3	0.381
Detailed	Remote Map	6	0.374
Basic	Reduce	234	1.126
Standard	Reduce	284	5.472
Detailed	Reduce	3,016	72.458

Table 5.3: Final average prediction accuracy.

limit the impact of reduce prediction errors even more, we set the *mapreduce.job.reduce.slowstart.completed maps* parameter to 1, to disable starting reduce tasks before all map tasks of a job have finished. This should force the first shuffle duration ($t^{R_s,1}$) from Equation 3.4 to always be 0, making reduce runtime predictions independent of when maps finish.

The second, **more extensive prediction analysis** is then performed. Between the two workloads described in Section 5.4.3, the prediction results of a total of 70 hour-long runs are recorded. These results can be seen in Table 5.3.

As mentioned before, the accuracy of predictions varies in time as the prediction model changes. To gain insight into the actual distribution of errors in time, given the entire task pool, we plot the 99th percentile of the relative errors of all task runtime predictions for a representative workload run, grouped by the time the predictions are made: figures 5.2 through 5.4. The bottom part of each chart is the magnified interval 0-1.5 for better visualization of the median values. In the case of the basic predictor, the median relative error is around 0.4 (or 40%) of the total task runtime for map tasks, and 0.6 (60%) for reduce tasks. While map errors never differ by more than a few seconds, erroneous reduce predictions are up to 50 times the actual duration of the task. The standard and detailed predictors both use the same algorithm for locality-agnostic map predictions, which yields a median of 0.35 relative error, with outliers of up to 4 (400%). When considering locality, the median relative error for local map tasks is generally lower than the locality-agnostic one by 0.1, while remote map task predictions are usually worse by 0.1. But the local map predictions have more disperse outliers that go up to 35, bringing the average relative error up. For reduce, standard predictions have a median of 0.7, with much more significant outliers, even over 300. The detailed predictor takes a while to gather enough data for more accurate predictions, after which the median error stabilizes at around 0.8. However, the most inaccurate of detailed predictions exceed 500, bringing the average relative error extremely high.

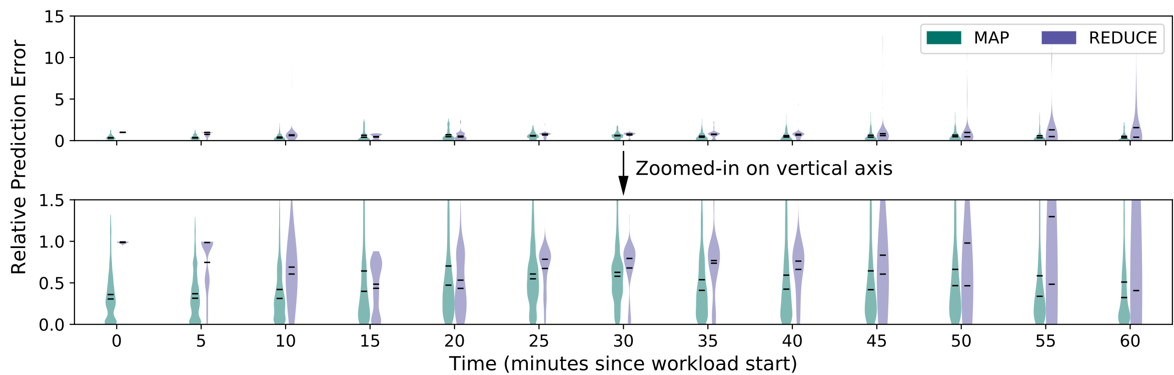


Figure 5.2: Relative prediction errors by task type for Basic Predictor.

Next we look at the same distribution of prediction errors for the standard predictor, but separating by job type. It appears that WordCount and Naive Bayes have the most predictable maps (0.1-0.2 median relative error) and the rest follow the general trend from Figure 5.3. For reduce, we see that Sort and Index seem to have rather low relative errors, while the outliers of Figure 5.3 are given by the predictions WordCount and

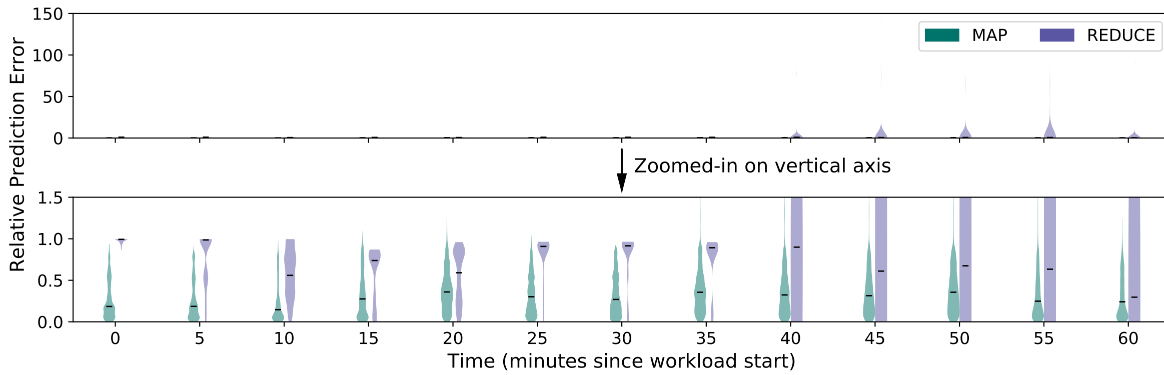


Figure 5.3: Relative prediction errors by task type for Standard Predictor.

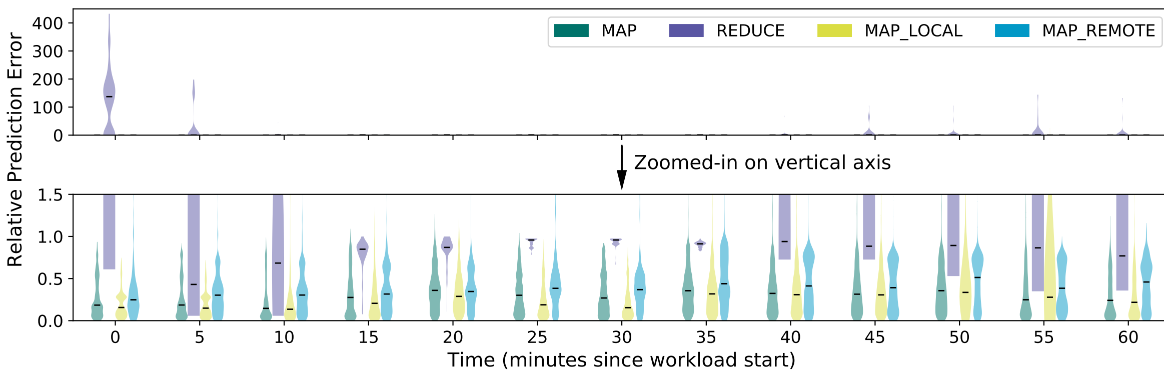


Figure 5.4: Relative prediction errors by task type for Detailed Predictor.

Naive Bayes. We, thus, inspect these jobs individually and look at the time each reduce phase takes. For Naive Bayes, merge and reduce times are relatively constant, while shuffle times fluctuate but not necessarily with input size. For WordCount, merge and reduce times seem to increase with input size, but the variability of shuffle times has more impact on the final task durations.

This leads us to analyze the runtime variability for each reduce type and size. All of our jobs suffer considerably from runtime variability, primarily in the shuffle phase. For instance, Figure 5.5 shows how the reduces for sort vary in runtime at different sizes. This impacts our reduce prediction results in two ways. Errors for long-running reduce tasks like Sort and Index, although not that significant relative to their total runtime, weigh in on the average absolute prediction error. While errors for short tasks, like those of WordCount and Naive Bayes, are not that significant in absolute values, but weigh in on the average relative error.

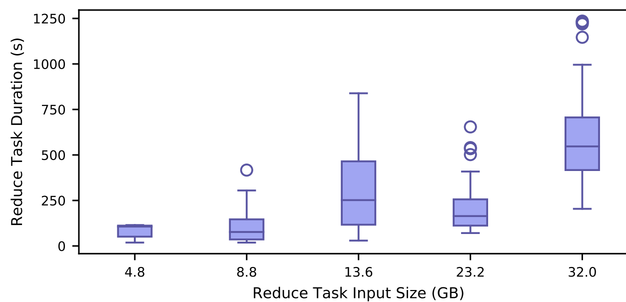


Figure 5.5: Reducer performance variability for Sort.

In conclusion, the results of the extended analysis are in line with the initial findings and support **MF1**.

Map tasks can be pretty accurately predicted, with a average relative error of about 35%, but differentiating by locality does not add value. Reduces are much more volatile, with an average relative error over 500%. And even disabling Hadoop's slow start mechanism (starting reduces even before all maps finish) does not lead to better results for certain job types. For these, a prediction model that takes into both a size-dependent and a constant component would be a better fit, along the lines of Equation 2.8, but calculated separately for each of the three phases of the reduce tasks. However, even such a model would suffer from the variability that our workloads exhibit. In the work of Verma et al. [51] all task types have an almost constant runtime for a given input size, meaning that their model would have yielded very different results for our use-case. Moreover, such models would need to train on many more samples than are available in our hour-long workloads to produce reliable results for single-reduce jobs like WordCount and Naive Bayes. It is left for future work to explore how such a model should be used and to what extent it could improve predictions.

5.4.2. Simulator Validation

Validating the accuracy of the simulator is not straightforward, as it needs to hypothesize about task-to-node allocation given a large number of tasks, fluctuating prediction accuracy, and a continuously evolving internal state of the simulated scheduling policy. While all this is hard to capture with a single evaluation function, we can, however create a basic visualization of the state of the cluster over time, comparing reality to the periodic simulations. For this we used a small workload of 7 mixed jobs arriving all at once. The system only runs and simulates one policy: Shortest-Remaining-Time-First. For prediction we use the Standard Predictor (with the modifications mentioned in Section 5.4.1), trained on the trace of a previous run of the same workload on the real cluster.

Figure 5.6 shows the evolution of the number of running tasks on the cluster, grouped by job. We can see that the first simulation starts only a few seconds into the run. The distribution of tasks in the beginning of the simulation follow roughly the same pattern as on the real cluster, however, the predictor underestimates the total time it would take Job 2 to finish, giving it priority over the rest of the jobs. In the second simulation (roughly ten minutes into the run), prediction is more accurate for Job 2, sending it to the end of the job queue, but Job 1 is expected to finish much earlier than in reality. This causes Job 6 to take over the cluster's slots a whole 2 minutes before this happens in the real cluster, and all following jobs to start earlier. Also, the final reduce tasks of Job 2 are grossly underestimated for the entire run of the workload, creating a large discrepancy between the next simulations and reality, with respect to the finish time of the workload.

Even though this analysis gives no quantitative evidence of the performance of the simulator, we believe that our visualization supports **MF2**. It shows that the allocation decisions of the simulated policy will be similar to reality, as long as task runtime predictions are accurate enough. We consider this result to be sufficient for carrying out the performance experiments that follow.

Next, we looked at the overhead of simulations and policy application. Figure 5.7 shows the 99th percentile for our results. Simulations last for an average of 14 seconds on the hour-long workloads that we experimented with. After the policy decision is made, it takes the meta-scheduler generally under a second to transfer state to the next policy. These values are not significant with respect to the average runtime of a job in the cluster (6.5 minutes).

5.4.3. System Performance

As the performance goal of our design is a combined score of metrics that have different measuring units and scales, there is no absolute value that can indicate how well the implementation is performing. Thus, we evaluate it through comparison on the same score function with each of its constituent policies. Two medium workloads are run with each of the allocation policies mentioned in Table 4.1, as well as with the dynamic policy switching enabled. Three different sets of values are chosen for the α and β normalization factors of the compound score formula (Equation 5.2.3) to measure the sensitivity of the system to administrator tuning, as seen in Table 5.4. The first favors batch jobs (BC Optimization), the second deadline constrained jobs (DC Optimization), and the third targets a mixture of the two (Mixed Optimization).

Figures 5.8 and 5.9 show the results of running two different workloads obtained through the method described in 5.2.2. The lower the score, the better the performance. The charts have been cut off at 2500 for better visualization. The first four groups on the x-axis represent the scores obtained by running the target workload with each of the constituent policies of the Portfolio Meta-Scheduler, listed in Table 4.1. The last three represent the scores obtained by the meta-scheduler's dynamic policy application which is based on the simulation evaluation modes from Table 5.4.

For the first workload, the SRTF policy gives the best results when optimizing for batch job slowdown,

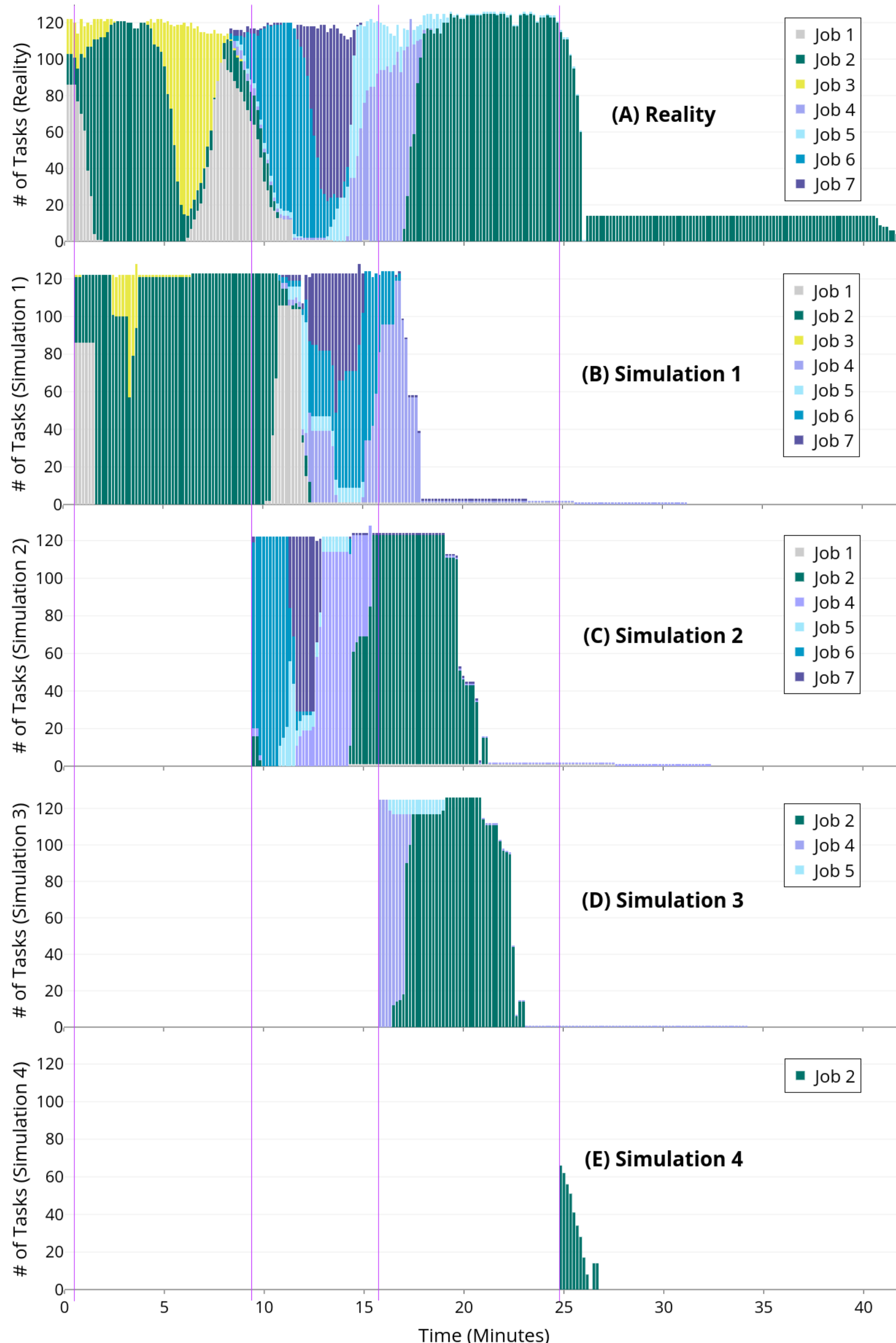


Figure 5.6: Number of running tasks on the cluster by job. Reality (A) is compared with the periodical simulations (B, C, D, and E).

followed by the batch-favoring DYN_BC mode of the meta-scheduler and then LOCF. The the deadline-aware policies, EDLS-Sh and EDLS-Pr give less importance to slowdown in favor of meeting job deadlines, resulting in much higher values for this type of score. As EDLS-Sh reserves an entire section of the cluster for deadline-constrained jobs that remains idle while only batch jobs are running in the system, its results are particularly poor in this respect. When considering the score function which prioritizes deadline-constrained jobs, however, EDLS_Sh has the best results of all, followed closely by EDLS-Pr. All three dynamic scheduling modes also perform better on this goal than SRTF and LOCF, but there is no clear distinction between the modes. The most surprising result is that, for the mixed optimization goal, it is the DYN_BC meta-scheduler mode

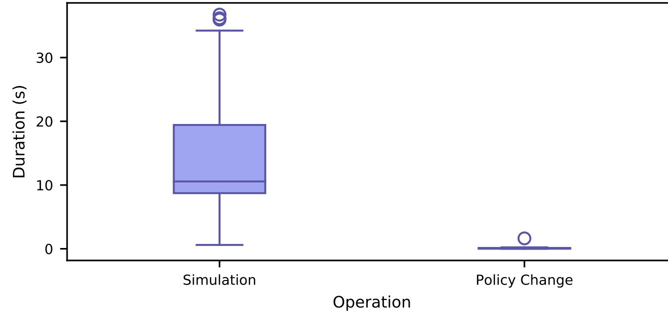


Figure 5.7: Time cost of simulations and policy changes.

Score Function	Slowdown Normalization Factor (α)	Deadline Violation Normalization Factor (β)	Meta-Scheduler Mode
BC Optimization	1000	1E-6	DYN_BC
DC Optimization	1E-6	1000	DYN_DC
Mixed Optimization	100	1	DYN_MID

Table 5.4: The types of score functions and their normalization factors.

that performs best, not the specialized DYN_MID, which comes in 3rd place, after SRTF.

On the second workload, POSUM has better overall performance. The meta-scheduler still cannot outperform SRTF in slowdown optimization or EDLS-Sh in reducing deadline violations, but it does comes in second place. While for the mixed optimization score, DYN_MID gives the best results out of all the other schedulers. Again interesting to mention is that DYN_MID seems to have a better performance than even DYN_BC with respect to batch job slowdown.

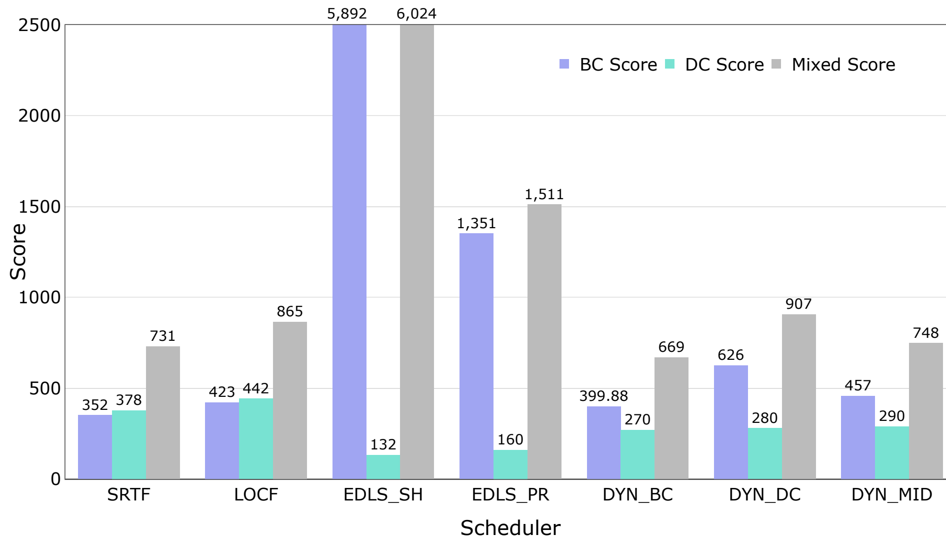


Figure 5.8: Experimental results on Workload 1.

The scores illustrated in the previous figures are averages over multiple runs of the same workload (see Section 5.4.4 for details). We choose representative runs of the first workload to do a deeper analysis into the behavior of the meta-scheduler in the three optimization modes. Figure 5.10 shows the overall policy usage of the Portfolio Meta-Scheduler during execution. We can see that each of the constituent policies are used. In each of the three cases, SRTF is the most used policy, but it is most predominant (65%) in optimizing for batch job slowdown, which, as we saw earlier is what SRTF does best. LOCF is also used a large portion of

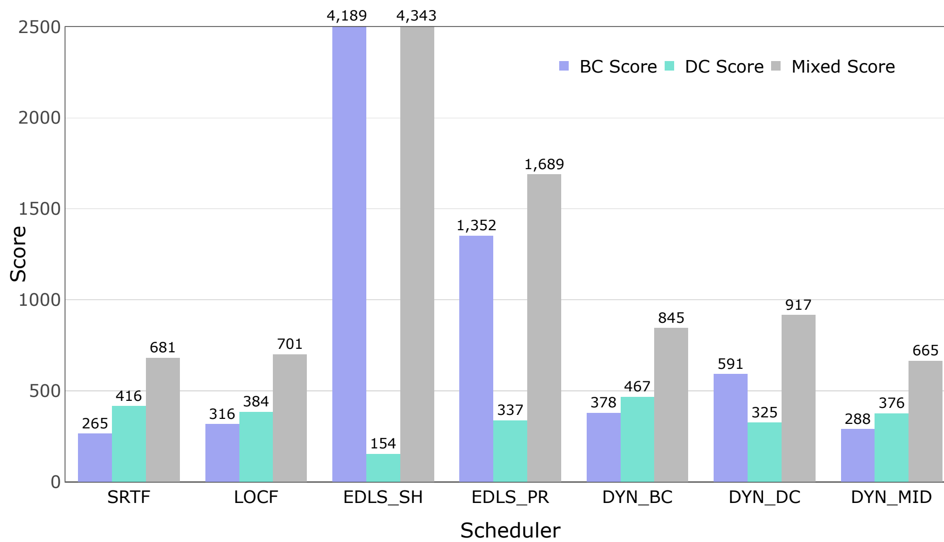


Figure 5.9: Experimental results on Workload 2.

the time. In DYN_DC mode, we see a more aggressive use of EDLS-Sh and EDLS-Pr, which try to minimize deadline constraint violations. In the mixed mode, policy shares are, as expected, a middle-ground between those presented by the other two modes.

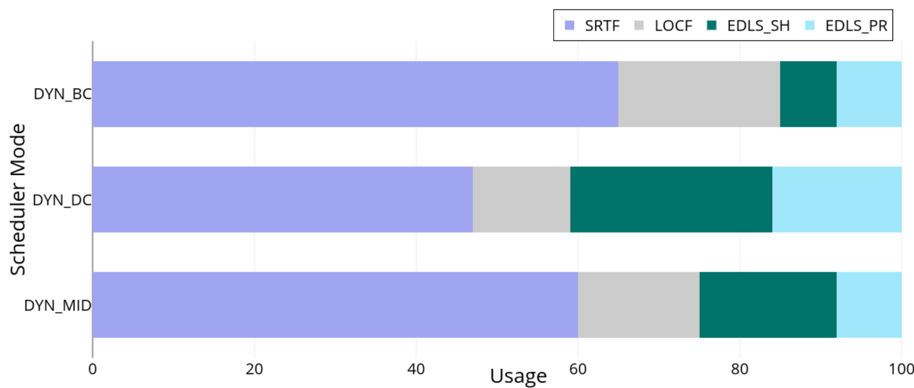


Figure 5.10: Portfolio Meta-Scheduler policy usage in different modes.

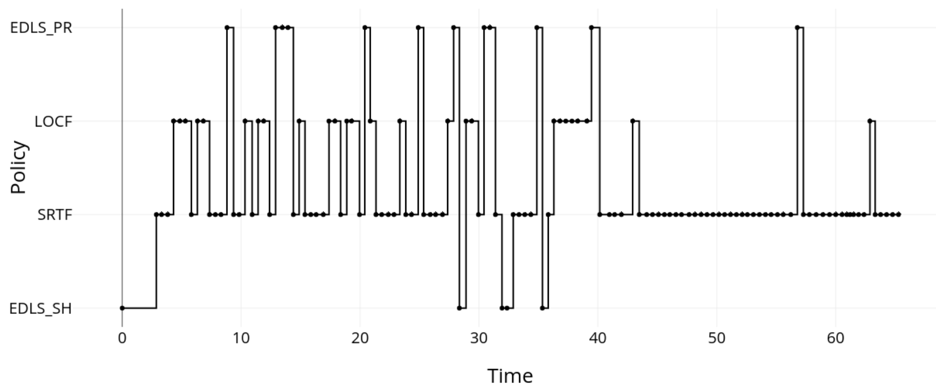


Figure 5.11: Meta-Scheduler policy choices in DYN_BC mode.

Thus, we see that policy usage does indeed adapt to the configured goal. However, SRTF seems to be very specialized in optimizing for slowdown, especially given a workload of predominantly small jobs that

incur a great slowdown penalty if left waiting. Any deviation from this, even a few incorrect choices during runtime will still penalize the batch optimization score. Figure 5.11 shows the policy choices for the DYN_BC run. For the first part of the run, the scheduler alternates between all policies. In the second half, it chooses predominantly SRTF. If we inspect the evolution of the slowdown readings, we see a very mild slope during the first half, and an abrupt increase in slowdown at the end of the run. This points towards the fact that the simulation scores for the first half of the run are close enough for a chance distribution of tasks on nodes to yield a slightly lower overall score for other policies than SRTF. Towards the end, when the slowdown penalty is about to become more significant, simulations are more clearly in favor of using SRTF. If we compare this to the evolution of slowdown for an SRTF-only run, the plot has the same shape, but with a slightly milder slope in the first half hour. This leads us to conclude that the penalty in slowdown is incurred due to small discrepancies between the simulated score and the actual slowdown score of the chosen policies, which are most likely due to prediction errors.

The same is valid for EDLS-Sh and the second performance goal. The bulk of the deadline violation penalty incurred by DYN_DC is due to a single job that finishes around 20 minutes into the run, while the meta-scheduler has been running mostly SRTF. The only way SRTF would be chosen over EDLS-Sh or -Pr is if none of the simulations predicted that the job would not finish in time to respect its deadline.

In conclusion, the results support **MF3**. The Portfolio Meta-Scheduler cannot out-perform policies that target a single performance objective specifically. However, it does a good job at balancing performance goals, even without flawless task runtime predictions. Another observation is that the end score of the workload run does not always reflect the exact configuration of the importance of slowdown versus deadline violations, as DYN_BC and DYN_MID have sometimes outperformed each other on their specific targets. So achieving the exact balance of BC job to DC job priority is not trivial.

5.4.4. System Stability

Finally, we look at the stability of the system: how much load it can handle and in what way its performance can vary.

During Workload 1, presented in Figure 5.8 we measured the resource utilization incurred by the POSUM components: Data Master, Orchestrator, Simulator, and Scheduler. Our main finding is that POSUM is scalable and does not utilize an excessive amount of resources. Figure 5.12 plots the CDF of the CPU utilization of the four components, while Figure 5.13 plots the memory usage. We notice that for the majority of the runtime, the four POSUM components use less than 50% CPU, and for 90% of the running time POSUM exhibits less than 30% CPU utilization. Moreover, the memory utilization of the four POSUM components is always below 1.5 GB. This is considered low memory utilization for all modern hardware. Our findings support **MF4**.

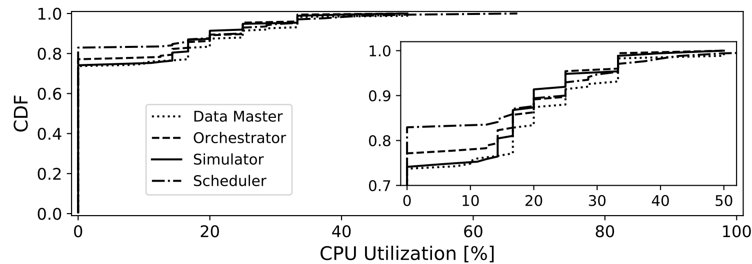


Figure 5.12: CDF of the CPU load for the POSUM components during runtime.

Performance-wise, substantial variability can seen from the first few runs of the first workload, not only for the dynamic modes, but also for single-policy runs. Consequently, we run each workload multiple times, calculating confidence intervals for each score type. The numbers in Figures 5.8 and 5.9 are only the resulting score averages. Table 5.5 shows the number of times each workload is run with each scheduler, the standard deviation, and the margin of error for the average scores given a confidence level of 90%.

We see that **MF4** holds true. Score averages for schedulers generally converge more easily for the scores that they optimize (e.g. SRTF for batch job slowdown). While, for the mixed optimization score, even the constituent policies seem to have higher error margins, especially when running the second workload.

We also look at POSUM's behavior under different levels system of load (i.e. number of jobs in the run queue). For lower loads, POSUM's dynamic policy application does not have many options for reprioritizing

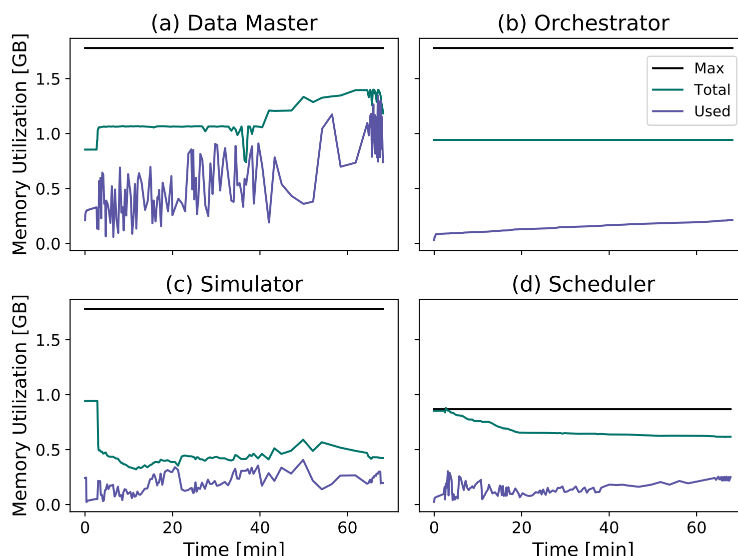


Figure 5.13: The memory usage of the four POSUM components during runtime as reported by their underlying JVM.

			BC Score			DC Score			Mixed Score		
Workload	Scheduler	Runs	Avg	Error Margin	StdDev	Avg	Error Margin	StdDev	Avg	Error Margin	StdDev
W1	SRTF	4	352	21	18	378	85	72	731	94	80
	LOCF	4	423	63	53	442	59	50	865	91	77
	EDLS_SH	4	5,892	992	843	132	43	37	6,024	1,034	879
	EDLS_PR	4	1,351	559	475	160	27	23	1,511	586	498
	DYN_MID	9	457	69	111	290	43	69	748	95	153
	DYN_DC	7	626	137	187	280	68	93	907	179	244
	DYN_BC	8	399	55	83	270	34	51	669	82	122
W2	SRTF	4	265	10	8	416	130	110	681	139	118
	LOCF	4	316	18	16	384	133	113	701	149	126
	EDLS_SH	4	4,189	791	672	154	27	23	4,343	787	669
	EDLS_PR	4	1,352	531	451	337	195	165	1,689	522	444
	DYN_MID	5	288	26	27	376	175	183	665	197	207
	DYN_DC	4	591	343	291	325	53	45	917	375	318
	DYN_BC	4	378	102	87	467	260	221	845	362	307

Table 5.5: Averages and error margins workload runs.

jobs, meaning scores across simulations seem to be more uniform, decisions become less clear and overall SRTF tends to perform best on both BC and mixed optimization scores, since deadlines are met more easily. For higher loads, up to 50 concurrent jobs in the system, deadlines become harder to meet and a more intensive usage of EDLS-based policies can be observed. With over more than 50 jobs in the system, the simulator can no longer handle the load. However, this does not mean that POSUM cannot add value under different load levels. It would just need configuration changes: higher thread and memory thresholds can be increased to support a larger number of jobs, more relaxed deadlines, adapted normalization factors, etc.

6 Conclusion and Future Work

MapReduce is a widely used programming paradigm that enables data processing to scale horizontally and without need of sophisticated hardware. In this thesis, we presented one of the drawbacks of some of the most popular MapReduce frameworks, such as Hadoop: using one scheduling policy to address all the functional and non-functional requirements of the cluster's users. This can be hard to achieve if the makeup of the workloads that run on the system is diverse, or if there are several metrics that play a role in the performance of the policy. We proposed a different solution: portfolio scheduling. By alternating schedulers from a set of complementary policies, the algorithmic complexity can be kept in check and the system can better adapt to the current conditions of the environment.

6.1. Conclusion on Main Research Questions

In this thesis, we have answered three main research questions resulting in the three main contributions of POSim, POSUM, and their experimental evaluation:

RQ1 *How can predictions be made on how a sequence of MapReduce jobs will behave (with respect to a series of performance metrics) when executed on a homogeneous cluster under a specific scheduling policy?*

We surveyed a number of existing schedulers for MapReduce systems and found neither of them directly applicable to our use case. Thus, we designed our own, based on approaches for task runtime prediction that we found in literature. The proposed solution, POSim, is an event-based online simulator that integrates with Hadoop. It uses simplified models of its internal components, like Application Masters and Node Managers, and mimics their inter-communication. We also designed a set of three task runtime predictors that can be plugged into POSim in order to assess their accuracy, given our use case.

RQ2 *How to design and develop a generic portfolio scheduler that supports MapReduce workloads?*

This aforementioned simulator, along with other standalone processes for data retrieval and orchestration, and a scheduler abstraction for online policy switching, make up the design of our complete portfolio scheduler for MapReduce workloads: POSUM. We found it best to keep the architecture modular and separated from the cluster itself, to be more maintainable and upgradable, and to have as little impact as possible on normal cluster operation. The system gathers information about the MapReduce cluster and its job queue and stores it in a database. Both historical statistics and the current state of the cluster are used as input by POSim to score individual policies based on the configured evaluation function. The system then makes a decision of which policy is best to use for the following time period. Since MapReduce frameworks do not support seamless switching of schedulers at runtime, we designed a meta-scheduler which can be plugged into the cluster like any other single scheduler, but actually delegates the scheduling decisions to the currently running policy.

RQ3 *How to demonstrate the capabilities and evaluate the performance of the proposed portfolio scheduler in realistic situations? In particular, does the portfolio scheduler perform better than its constituent policies?*

We chose to implement the proposed system as part of the Hadoop YARN stack and then evaluate it using a set of real-world experiments on synthetic workloads containing a mix of both deadline-constrained and batch jobs. The goal was optimize for two performance objectives: minimizing deadline violations and reducing batch job slowdown as much as possible. The experimental assessment of the simulator's predictors has shown that map task runtimes can be predicted with a low error based on the average input processing

rates of past map tasks. Splitting these averages between local and non-local tasks, on the other hand, leads to a higher average error than considering predictions location-agnostic. For reduce tasks, we have found that some of our predictions can deviate considerably from reality when task runtimes do not scale proportionally with reduce input sizes. But even with this accuracy, POSUM still managed to perform for its intended purposes. It cannot out-perform policies that target a single performance objective specifically. However, it does a good job at balancing performance goals. Another observation was that the end score of the workload run does not always reflect the exact configuration of the importance of slowdown versus deadline violations. So achieving the exact balance of batch job to deadline-constrained job priority is not trivial.

6.2. Lessons Learned

This project started out as a somewhat straightforward application of portfolio scheduling to the MapReduce context. However, several aspects of the target environment caused the amount of effort required to complete this challenge to greatly exceed initial estimations. Firstly, literature on MapReduce is extensive and most studies focus on the older version of Hadoop, before the abstractions that YARN brings forth. This made it difficult to judge the current limitations of scheduling, to understand what data is available to the scheduler about the jobs that it is managing, and to recreate schedulers depicted in literature. Secondly, it was assumed that a compatible simulator could be found that would only require adaptation and configuration to be used with POSUM. This was not the case, and a full prediction and simulation approach had to be contrived and implemented. Thirdly, the Hadoop system is already a complete software product with a large source code base and a highly concurrent operational paradigm. Considerable effort was put into understanding its architecture and inter-operating with it at runtime. Finally, workload generation and execution for a MapReduce environment is not trivial: benchmarking tools needed to be adapted, a separate system was devised for injecting deadlines, as they are not directly supported by the framework, and data generation and transfer for experiments was time consuming and error prone. All in all, software implementation alone took 12 person-months, and another 3 person-months were spent on testing, validation, and the experimental analysis.

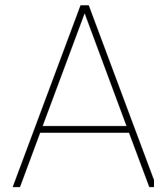
As a result of these circumstances, this thesis is only an introductory analysis of the capabilities of portfolio scheduling for big data processing. The following section proposes several directions for improvement in the future.

6.3. Recommendations for Future Research

With respect to **RQ1**, the least favorable results were obtained for reduce prediction accuracy. It is worth investigating a more complex model to handle instances when task runtimes do not scale proportionally with reduce input sizes. Although linear regression was the method of choice in related work, we did observe a considerable degree of variability even on a simple algorithm like Sort, which would be tricky to capture in such a model. Moreover, it is not trivial to gather enough training data (both in quantity and quality) with little overhead and where job types and characteristics constantly change.

Regarding the design of POSUM itself (**RQ2**), we have observed that the evaluation function's scaling factors should be tuned periodically either by a human administrator or an adaptive algorithm that can account for changes in the order of magnitude of the constituent evaluation metrics. The latter, along with other features of the Feedback Module mentioned in this thesis could make an interesting subject for future work. Another area of development is the exploration of cluster resizing policies in combination with the allocation policies to optimize for a third performance objective: cumulated node lease cost.

Finally, our work is but a glimpse into the applicability of portfolio scheduling for big data processing (**RQ3**). We would like to explore a more varied workload composition and possibly different performance score definitions, to offer more insight into the limitations of portfolio scheduling for big data workloads. Also, we recently notice a trend for big data processing ecosystems to migrate towards more efficient, in-memory frameworks, such as Spark [15]. Due to its generality and modularity, we believe that POSUM could be adapted to such systems as well.



Glossary

batch job a job that is not constrained by a specific deadline

BC job batch job

client the software entity that sends a request to the system

cloud an internet-based computing model that uses abstracted hardware resources from one or more datacenters

cluster a group of interconnected computers

commodity hardware hardware that is widely accessible and affordable for private consumers

datacenter a facility used to house a large number of interconnected computer systems

DC job deadline constrained job

DFS Distributed File System; a file storage system where each file is divided into chunks (blocks) and distributed over several computers

FIFO First-In-First-Out; a prioritization mechanism for jobs where priority increases with wait time

heterogeneous cluster a cluster on which the amount and quality of computation resources (RAM memory, CPU cores, disk storage space and speed) differs from node to node, and/or there is variable background load on the machines

homogeneous cluster a cluster on which the amount and quality of computation resources (RAM memory, CPU cores, disk storage space and speed) are the same on each node and the nodes do not have any background load

hypervisor a software program that creates, runs and monitors virtual machines

job execution time the time spent only in executing mapper/reducer code (wait times excluded)

job runtime / makespan / turnaround time / response time the time from when a client submits a MapReduce application for execution, until it is finished

LJF Largest-Slowdown-First a prioritization mechanism for jobs where priority increases with the ratio of a job's wait time in the system over its execution time

locality a scheduling goal where involving the execution of map tasks on the DFS's nodes which have their input splits already stored locally

mapper the software entity that runs the map code in a MapReduce system

MapReduce system a collection of worker nodes and a coordinating framework on which applications following the MapReduce model can be executed

metadata data about data; file metadata is the information about a file's chunks and their location

metric a standard of measure of a degree to which a software system or process possesses some property

multi-tenant the property of a cluster or datacenter to host several entities (users, organizations) that need to share their computational resources

POSUM POrtfolio SchedUler for Mapreduce

preempt to stop the execution of a task in the interest of re-allocating its resources to a higher priority job

reducer the software entity that runs the reduce code in a MapReduce system

redundant the property of being resistant to failures, mainly through replication

resource a quantity of hardware capacity (e.g. 1 GB RAM, 1 CPU cores, 1 TB HDD space)

RPC Remote Procedure Call; a client-server protocol that is established between two remote software processes in order to communicate as if the caller is invoking the target locally

node a (virtual) computer in a cluster/datacenter

scalability (for hardware) the ability of a system to grow or shrink in size over time; (for software) the ability of a system to adapt in a deterministic and linear (non-bursty) fashion to a change in input size or load

scheduling the allocation of hardware resources (slots/containers) to jobs

shuffle the phase in the execution of a job in which the intermediary data is sorted by keys and data is transferred from the mappers to the reducers

SLA Service Level Agreement; the contract between a service provider and the end user that defines the level of service expected from the service provider

SLO Service Level Objective; a specific value of a performance metric that can be specified in an SLA

slot a logical group of hardware resources that can be used to run a single task/container in a MapReduce system

speculation the process of starting another copy of a straggling task in the hope that it will finish sooner and the straggler can be forcefully killed

split a chunk of the input for a MapReduce application; its maximum size is determined by a configuration parameter in the framework

SRTF Shortest-Remaining-Time-First; a prioritization mechanism for jobs where priority increases with the decrease in the remaining processing time of a job

straggler a running task that is taking unexplainably longer to finish than other similar tasks

trace the recorded history of a workload that was run on a MapReduce system

uptime the total time a computer (or VM) is powered on and leased for use within the cluster

virtual machine (VM) an emulation of a physical computer that abstracts its hardware resources

workload a series of jobs running / to be run on a MapReduce system

B
POSUM Admin

Figure B.1: The cluster tab displays the number of running applications and containers, and the available resources.

Figure B.2: The scheduler tab displays the information about policy usage and the time cost of scheduler operations.

Figure B.3: The system tab displays the resource usage of each process in the environment.

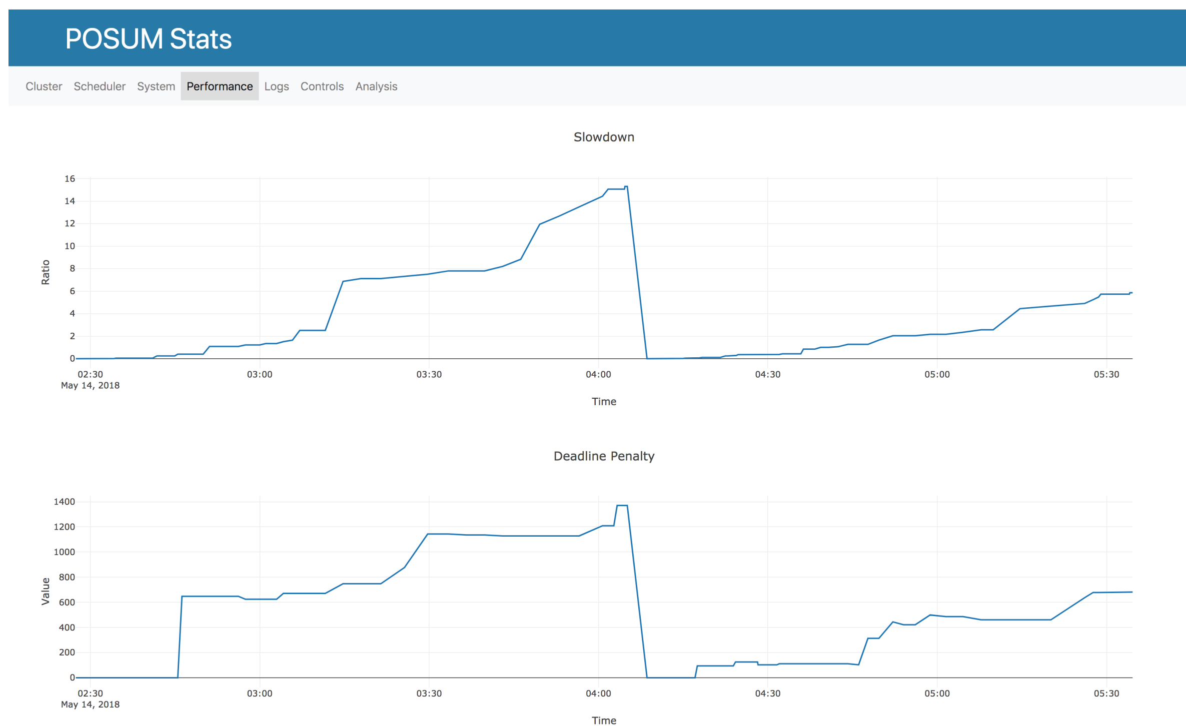


Figure B.4: The performance tab displays the readings for each active performance metric.

POSUM Stats

Cluster Scheduler System Performance Logs Controls Analysis

Time	Message
05-14 02:27:27	Simulation was not performed
05-14 02:27:56	Simulation was not performed
05-14 02:28:26	Simulation was not performed
05-14 02:28:56	Simulation was not performed
05-14 02:29:26	Simulation was not performed
05-14 02:29:56	Simulation was not performed
05-14 02:30:28	Starting simulation for EDLS_SH
05-14 02:30:28	Starting simulation for SRTF
05-14 02:30:28	Starting simulation for LOCF
05-14 02:30:28	Starting simulation for EDLS_PR
05-14 02:30:42	Score for simulation of SRTF: policyName: "SRTF" score { slowdown: 0.016511127063890883 penalty: 0.0 cost: 0.0 }
05-14 02:30:42	Score for simulation of EDLS_SH: policyName: "EDLS_SH" score { slowdown: 0.03194544149318019 penalty: 0.0 cost: 0.0 }
05-14 02:30:42	Score for simulation of LOCF: policyName: "LOCF" score { slowdown: 0.019023689877961235 penalty: 0.0 cost: 0.0 }
05-14 02:30:42	Score for simulation of EDLS_PR: policyName: "EDLS_PR" score { slowdown: 0.02189519023689878 penalty: 0.0 cost: 0.0 }
05-14 02:30:42	Simulation results: [policyName: "EDLS_SH" score { slowdown: 0.03194544149318019 penalty: 0.0 cost: 0.0 }, policyName: "LOCF" score { slowdown: 0.019023689877961235 penalty: 0.0 cost: 0.0 }, policyName: "EDLS_PR" score { slowdown: 0.02189519023689878 penalty: 0.0 cost: 0.0 }, policyName: "SRTF" score { slowdown: 0.016511127063890883 penalty: 0.0 cost: 0.0 }]
05-14 02:30:42	Changed scheduling policy to SRTF
05-14 02:30:58	Starting simulation for EDLS_PR

Figure B.5: The logs tab displays real-time operating logs.

POSUM Stats

Cluster Scheduler System Performance Logs Controls Analysis

Evaluation Scale Factors

Alpha 1

The weight of slowdown in score comparison.

Beta 0.000001

The weight of deadline violations in score comparison.

Gamma 1

The weight of cost in score comparison.

Save

System Reset

Press this button if you want to erase all data and prediction models.
This will also stop any ongoing simulations.

Reset

Current policy

Choose a specific scheduler, or enable dynamic switching:

DYNAMIC

Change

Figure B.6: The controls tab allows administrators to tweak performance normalization factors, to reset the system, and to force the application of a specific policy.

Bibliography

[1] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G. Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, pages 265–278, 2010. URL http://www.usenix.org/events/osdi10/tech/full_papers/Ananthanarayanan.pdf.

[2] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. GRASS: trimming stragglers in approximation analytics. In *NSDI*, pages 289–302, 2014. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/ananthanarayanan.

[3] Georgios Andreadis, Laurens Versluis, Fabian Mastenbroek, and Alexandru Iosup. A reference architecture for datacenter scheduling: Extended technical report. *arXiv preprint arXiv:1808.04224*, 2018.

[4] Apache. Mumak: Map-reduce simulator. https://issues.apache.org/jira/browse/MAPREDUCE-728,.

[5] Apache. Rumen. https://hadoop.apache.org/docs/r1.2.1/rumen.html,.

[6] Apache. Yarn scheduler load simulator (sls). https://hadoop.apache.org/docs/r2.4.1/hadoop-sls/SchedulerLoadSimulator.html,.

[7] Apache. Apache hadoop yarn. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html,.

[8] Rajkumar Buyya and M. Manzur Murshed. Gridsim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, 2002. doi: 10.1002/cpe.710. URL http://dx.doi.org/10.1002/cpe.710.

[9] Christopher D. Carothers, David W. Bauer, and Shawn Pearce. ROSS: A high-performance, low-memory, modular time warp system. *J. Parallel Distrib. Comput.*, 62(11):1648–1669, 2002. doi: 10.1016/S0743-7315(02)00004-7. URL http://dx.doi.org/10.1016/S0743-7315(02)00004-7.

[10] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *J. Parallel Distrib. Comput.*, 74 (10):2899–2917, 2014. doi: 10.1016/j.jpdc.2014.06.008. URL http://dx.doi.org/10.1016/j.jpdc.2014.06.008.

[11] Hsinchun Chen, Roger HL Chiang, and Veda C Storey. Business intelligence and analytics: From big data to big impact. *MIS quarterly*, 36(4):1165–1188, 2012.

[12] Quan Chen, Daqiang Zhang, Minyi Guo, Qianni Deng, and Song Guo. Samr. In *10th IEEE International Conference on Computer and Information Technology, CIT 2010, Bradford, West Yorkshire, UK, June 29-July 1, 2010*, pages 2736–2743, 2010. doi: 10.1109/CIT.2010.458. URL http://dx.doi.org/10.1109/CIT.2010.458.

[13] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy H. Katz. The case for evaluating mapreduce performance using workload suites. In *MASCOTS*, pages 390–399, 2011. doi: 10.1109/MASCOTS.2011.12. URL http://dx.doi.org/10.1109/MASCOTS.2011.12.

[14] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *PVLDB*, 5(12):1802–1813, 2012. URL http://vldb.org/pvldb/vol5/p1802_yanpeichen_vldb2012.pdf.

[15] Nathanaël Cheriere, Pierre Donat-Bouillud, Shadi Ibrahim, and Matthieu Simonin. On the usability of shortest remaining time first policy in shared hadoop clusters. In Sascha Ossowski, editor, *SAC*, pages 426–431. ACM, 2016. doi: 10.1145/2851613.2851626. URL http://doi.acm.org/10.1145/2851613.2851626.

[16] Jason Cope, Ning Liu, Sam Lang, Phil Carns, Chris Carothers, and Robert Ross. Codes: Enabling co-design of multilayer exascale storage architectures. In *EST*, pages 303–312, 2011.

[17] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004. URL http://www.usenix.org/events/osdi04/tech/dean.html.

[18] Kefeng Deng, Junqiang Song, Kaijun Ren, and Alexandru Iosup. Exploring portfolio scheduling for long-term execution of scientific workloads in iaas clouds. In *SC*, pages 55:1–55:12, 2013. doi: 10.1145/2503210.2503244. URL http://doi.acm.org/10.1145/2503210.2503244.

[19] Kefeng Deng, Ruben Verboon, Kaijun Ren, and Alexandru Iosup. A periodic portfolio scheduler for scientific computing in the data center. In *JSSPP*, pages 156–176, 2013. doi: 10.1007/978-3-662-43779-7_9. URL http://dx.doi.org/10.1007/978-3-662-43779-7_9.

[20] Xicheng Dong, Ying Wang, and Huaming Liao. Scheduling mixed real-time and non-real-time applications in mapreduce environment. In *ICPADS*, pages 9–16, 2011. doi: 10.1109/ICPADS.2011.115. URL http://dx.doi.org/10.1109/ICPADS.2011.115.

[21] Greg Ganger, Bruce Worthington, and Yale Patt. The disksim simulation environment (v4.0). http://www.pdl.cmu.edu/DiskSim/.

[22] Bogdan Ghit, Nezih Yigitbasi, Alexandru Iosup, and Dick H. J. Epema. Balanced resource allocations across multiple dynamic mapreduce clusters. In *SIGMETRICS*, pages 329–341, 2014. doi: 10.1145/2591971.2591998. URL http://doi.acm.org/10.1145/2591971.2591998.

[23] Suhel Hammoud, Maozhen Li, Yang Liu, Nasullah Khalid Alham, and Zelong Liu. Mrsim: A discrete event based mapreduce simulator. In *FSKD*, pages 2993–2997, 2010. doi: 10.1109/FSKD.2010.5569086. URL http://dx.doi.org/10.1109/FSKD.2010.5569086.

[24] Chen He, Ying Lu, and David Swanson. Matchmaking: A new mapreduce scheduling technique. In *CloudCom*, pages 40–47, 2011. doi: 10.1109/CloudCom.2011.16. URL http://dx.doi.org/10.1109/CloudCom.2011.16.

[25] Herodotos Herodotou. *Automatic tuning of data-intensive analytical workloads.* PhD thesis, Duke University, 2012.

[26] Herodotos Herodotou and Shivnath Babu. A what-if engine for cost-based mapreduce optimization. *IEEE Data Eng. Bull.*, 36(1):5–14, 2013. URL http://sites.computer.org/debull/A13mar/herod.pdf.

[27] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, pages 261–272. www.cidrdb.org, 2011. URL http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper36.pdf.

[28] Fred Howell and Ross McNab. Simjava: A discrete event simulation library for java. *Simulation Series*, 30:51–56, 1998.

[29] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Workshops Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 41–51, 2010. doi: 10.1109/ICDEW.2010.5452747. URL http://dx.doi.org/10.1109/ICDEW.2010.5452747.

[30] Bernardo A Huberman, Rajan M Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.

[31] ISI. The network simulator - ns-2. http://nsnam.sourceforge.net/wiki/index.php/User_Information.

[32] Hui Kang, Yao Chen, Jennifer L. Wong, Radu Sion, and Jason Wu. Enhancement of xen's scheduler for mapreduce workloads. In *HPDC*, pages 251–262, 2011. doi: 10.1145/1996130.1996164. URL http://doi.acm.org/10.1145/1996130.1996164.

[33] Kamal Kc and Kemafor Anyanwu. Scheduling hadoop jobs to meet deadlines. In *CloudCom*, pages 388–392, 2010. doi: 10.1109/CloudCom.2010.97. URL http://dx.doi.org/10.1109/CloudCom.2010.97.

[34] Wagner Kolberg, Pedro de B. Marcos, Julio C. S. dos Anjos, Alexandre K. S. Miyazaki, Cláudio Fernando Resin Geyer, and Luciana Arantes. MRSG - A mapreduce simulator over simgrid. *Parallel Computing*, 39(4-5):233–244, 2013. doi: 10.1016/j.parco.2013.02.001. URL http://dx.doi.org/10.1016/j.parco.2013.02.001.

[35] Ping Li, Lei Ju, Zhiping Jia, and Zhiwen Sun. Sla-aware energy-efficient scheduling scheme for hadoop YARN. In *ICESS*, pages 623–628, 2015. doi: 10.1109/HPCC-CSS-ICESS.2015.181. URL http://dx.doi.org/10.1109/HPCC-CSS-ICESS.2015.181.

[36] Norman Lim, Shikharesh Majumdar, and Peter Ashwood-Smith. A constraint programming based hadoop scheduler for handling mapreduce jobs with deadlines on clouds. In *ICPE*, pages 111–122, 2015. doi: 10.1145/2668930.2688058. URL http://doi.acm.org/10.1145/2668930.2688058.

[37] Ning Liu, Xi Yang, Xian-He Sun, Jonathan Jenkins, and Robert B. Ross. Yarnsim: Simulating hadoop YARN. In *CCGrid*, pages 637–646, 2015. doi: 10.1109/CCGrid.2015.61. URL http://dx.doi.org/10.1109/CCGrid.2015.61.

[38] Michael Mattess, Rodrigo N. Calheiros, and Rajkumar Buyya. Scaling mapreduce applications across hybrid clouds to meet soft deadlines. In *AINA*, pages 629–636, 2013. doi: 10.1109/AINA.2013.51. URL http://dx.doi.org/10.1109/AINA.2013.51.

[39] Inc. MongoDB. MongoDB Architecture Guide. Technical Report MongoDB 3.2, MongoDB, Inc., 2015. URL https://www.mongodb.com/collateral/mongodb-architecture-guide.

[40] Phuong Nguyen, Tyler A. Simon, Milton Halem, David Chapman, and Quang Le. A hybrid scheduling algorithm for data intensive workloads in a mapreduce environment. In *UCC*, pages 161–167. IEEE Computer Society, 2012. doi: 10.1109/UCC.2012.32. URL http://dx.doi.org/10.1109/UCC.2012.32.

[41] Jongse Park, DaeWoo Lee, Bokyeong Kim, Jaehyuk Huh, and Seungryoul Maeng. Locality-aware dynamic VM reconfiguration on mapreduce clouds. In *HPDC*, pages 27–36, 2012. doi: 10.1145/2287076.2287082. URL http://doi.acm.org/10.1145/2287076.2287082.

[42] Jorda Polo, David Carrera, Yolanda Becerra, Malgorzata Steinder, and Ian Whalley. Performance-driven task co-scheduling for mapreduce environments. In *NOMS*, pages 373–380, 2010. doi: 10.1109/NOMS.2010.5488494. URL http://dx.doi.org/10.1109/NOMS.2010.5488494.

[43] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, pages 13–24, 2007. doi: 10.1109/HPCA.2007.346181. URL http://dx.doi.org/10.1109/HPCA.2007.346181.

[44] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SOCC*, page 7, 2012. doi: 10.1145/2391229.2391236. URL http://doi.acm.org/10.1145/2391229.2391236.

[45] Stewart Robinson. *Simulation: the practice of model development and use*. Palgrave Macmillan, 2014.

[46] Yongcai Tao, Qing Zhang, Lei Shi, and Pinhua Chen. Job scheduling optimization for multi-user mapreduce clusters. In *PAAP*, pages 213–217, 2011. doi: 10.1109/PAAP.2011.33. URL http://dx.doi.org/10.1109/PAAP.2011.33.

[47] Vincent van Beek, Jesse Donkervliet, Tim Hegeman, Stefan Hugtenburg, and Alexandru Iosup. Self-expressive management of business-critical workloads in virtualized datacenters. *IEEE Computer*, 48(7):46–54, 2015. doi: 10.1109/MC.2015.206. URL http://dx.doi.org/10.1109/MC.2015.206.

[48] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *ICAC*, pages 235–244, 2011. doi: 10.1145/1998582.1998637. URL http://doi.acm.org/10.1145/1998582.1998637.

[49] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Play it again, simmr! In *CLUSTER*, pages 253–261, 2011. doi: 10.1109/CLUSTER.2011.36. URL http://dx.doi.org/10.1109/CLUSTER.2011.36.

[50] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Resource provisioning framework for mapreduce jobs with performance goals. In *Middleware*, pages 165–186, 2011. doi: 10.1007/978-3-642-25821-3_9. URL http://dx.doi.org/10.1007/978-3-642-25821-3_9.

[51] Abhishek Verma, Ludmila Cherkasova, Vijay S. Kumar, and Roy H. Campbell. Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle. In *NOMS*, pages 900–905, 2012. doi: 10.1109/NOMS.2012.6212006. URL http://dx.doi.org/10.1109/NOMS.2012.6212006.

[52] Guanying Wang, Ali Raza Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *MASCOTS*, pages 1–11. IEEE Computer Society, 2009. doi: 10.1109/MASCOT.2009.5366973. URL http://dx.doi.org/10.1109/MASCOT.2009.5366973.

[53] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. Bigdatabench: A big data benchmark suite from internet services. In *HPCA*, pages 488–499, 2014. doi: 10.1109/HPCA.2014.6835958. URL http://dx.doi.org/10.1109/HPCA.2014.6835958.

[54] Nezih Yigitbasi, Kushal Datta, Nilesh Jain, and Theodore Willke. Energy efficient scheduling of mapreduce workloads on heterogeneous clusters. In *GCM*, page 1. ACM, 2011.

[55] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, pages 29–42, 2008. URL http://www.usenix.org/events/osdi08/tech/full_papers/zaharia/zaharia.pdf.

[56] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278, 2010. doi: 10.1145/1755913.1755940. URL http://doi.acm.org/10.1145/1755913.1755940.

[57] Wei Zhang, Sundaresan Rajasekaran, Timothy Wood, and Mingfa Zhu. MIMP: deadline and interference aware scheduling of hadoop virtual machines. In *CCGrid*, pages 394–403. IEEE Computer Society, 2014. doi: 10.1109/CCGrid.2014.101. URL http://dx.doi.org/10.1109/CCGrid.2014.101.