# Balanced Resource Allocations Across Multiple Dynamic MapReduce Clusters

Bogdan Ghiţ[§], Nezih Yigitbasi[†], Alexandru Iosup[§], Dick Epema[§$]
[§]Delft University of Technology, the Netherlands
[$]Eindhoven University of Technology, the Netherlands
[†]Intel Labs, USA
b.i.ghit@tudelft.nl, nezih.yigitbasi@intel.com
{a.iosup,d.h.j.epema}@tudelft.nl

## ABSTRACT

Running multiple instances of the MapReduce framework concurrently in a multicluster system or datacenter enables data, failure, and version isolation, which is attractive for many organizations. It may also provide some form of performance isolation, but in order to achieve this in the face of time-varying workloads submitted to the MapReduce instances, a mechanism for dynamic resource (re-)allocations to those instances is required. In this paper, we present such a mechanism called Fawkes that attempts to balance the allocations to MapReduce instances so that they experience similar service levels. Fawkes proposes a new abstraction for deploying MapReduce instances on physical resources, the MR-cluster, which represents a set of resources that can grow and shrink, and that has a core on which MapReduce is installed with the usual data locality assumptions but that relaxes those assumptions for nodes outside the core. Fawkes dynamically grows and shrinks the active MR-clusters based on a family of weighting policies with weights derived from monitoring their operation.

We empirically evaluate Fawkes on a multicluster system and show that it can deliver good performance and balanced resource allocations, even when the workloads of the MR-clusters are very uneven and bursty, with workloads composed from both synthetic and real-world benchmarks.

## Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications; D.2.8 [**Metrics**]: Performance measures

## General Terms

Experimentation, Measurement, Performance

## Keywords

MapReduce clusters; Scheduling; Dynamic provisioning; Performance; Fairness; Datacenters; Multicluster systems.

## 1. INTRODUCTION

MapReduce and similar computing frameworks are now widely used by institutes and commercial companies (e.g., Google [9], Facebook [33], Yahoo! [7]) because of their ability to efficiently use large sets of computing resources and to analyze large data volumes. MapReduce workloads may be very heterogeneous in terms of their data size and their resource requirements [18], and mixing them within a single instance of a computing framework may lead to conflicting optimization goals. Therefore, *isolating* MapReduce workloads and their data while dynamically *balancing* the resources across them is very attractive for many organizations. In this paper, we present the design and analysis of Fawkes[1], a scheduling system for *dynamic* resource provisioning of multiple MapReduce instances in single large-scale infrastructures.

Running multiple MapReduce frameworks concurrently within the same physical infrastructure enables *four types of isolation.* First, different (groups of) users each working with their own data set may prefer to have their own MapReduce framework to avoid interference, or for privacy and security reasons, thus requiring *data isolation.* A second type of isolation is *failure isolation*, which hides the failures of one MapReduce framework from the applications running in other concurrent MapReduce frameworks. Third, with the multiple MapReduce frameworks approach, *version isolation*, with different versions of the MapReduce framework running simultaneously, may be achieved as well. Finally, it can enable *performance isolation* between streams of jobs with different characteristics, for instance, by having separate MapReduce frameworks for large and small jobs, or for production and experimental jobs.

Whereas the first three forms of isolation are easily enforced by a resource manager that can deploy multiple instances of the MapReduce framework along with their corresponding filesystems on disjoint sets of nodes, performance isolation is more difficult to achieve (and define)—as the workloads of the instances may vary considerably over their lifetimes, deploying them on static partitions of the system may lead to an imbalance in the levels of service they receive. To dynamically provision multiple framework instances at runtime, Fawkes defines a new abstraction of the MapReduce framework called the *MR-cluster.* An MR-cluster is initially deployed (along with its filesystem) on a system

[1] Fawkes is a phoenix bird which in Greek mythology is reborn from its own ashes just like our MapReduce clusters grow and shrink.

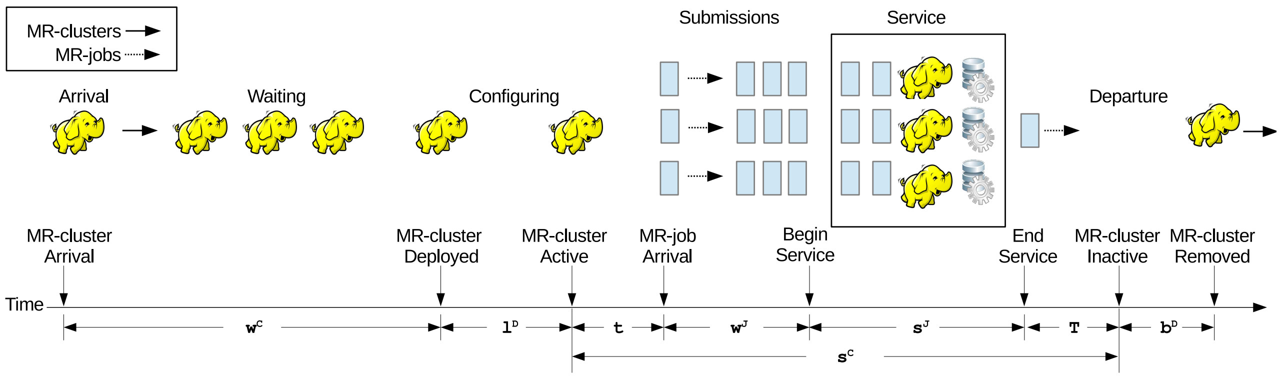


Figure 1: An overview of the queuing system of FAWKES. The system handles requests for deploying MapReduce frameworks (MR-clusters) with a global queue, and each active MR-cluster manages an internal queue of MapReduce jobs (MR-jobs). The Hadoop logo is taken from [2].

partition of a certain minimum size consisting of *core* nodes when its first job is submitted, and it will remain active as long as it receives additional jobs. The allocation of an MR-cluster can grow (and later shrink) by adding (removing) *transient* or *transient-core* nodes that don't store any data or only output data, respectively, thus breaking the standard MapReduce data locality assumption but allowing fast reconfiguration. FAWKES implements three types of policies for setting and periodically adjusting the *weights* of the active MR-clusters that indicate the shares of the resources they are entitled to, and to resize their allocations accordingly. These policies try to assess the load conditions of the MR-clusters by considering their queue lengths, their resource utilizations, or their performance (in terms of, e.g., job slowdown) when setting the weights. The most important performance metric we use to assess the actual performance of the MR-clusters is the average job slowdown.

Another possible solution for provisioning multiple MapReduce instances is to share the distributed file system across all frameworks and to employ two-level scheduling by delegating the scheduling control to the frameworks, as is done in Mesos [19]. There, a high-level resource manager initiates resource offers to the frameworks, which need specific policies to decide whether to accept or reject these offers. Mesos achieves near-optimal data locality when resources are frequently released by the frameworks. Instead, our solution targets performance isolation for time-varying workloads, but breaks the data locality assumptions to enable fast framework reconfigurations. Whereas Mesos has an offer-based scheduling system, FAWKES employs a feedback mechanism and balances the allocations of multiple frameworks by monitoring their operation.

The contributions of this paper are as follows:

1. We define the abstraction of the MR-cluster which is a set of resources that can grow and shrink, that has MapReduce installed on its core in the usual way, but that relaxes the MapReduce data locality assumptions for nodes outside its core (Section 3).

2. We provide a comprehensive taxonomy of policies for provisioning multiple MR-clusters that take into account their dynamic load conditions as perceived from their queue lengths, the utilizations of the resources allocated to them, or the performance they deliver (Section 4).

3. With a set of micro-experiments in a real multicluster system, we analyze, among other aspects of FAWKES, the benefit of trading data locality for dynamicity (Section 6). We find that the performance penalty induced by a relaxed data locality model in MapReduce is not prohibitive.

4. With a set of macro-experiments in the multicluster system, we evaluate the weighting policies of FAWKES for balancing the allocations of multiple MR-clusters. We show that our system delivers good results even for unfavorable workloads (Section 7).

## 2. BACKGROUND

MapReduce [9] is a programming model that exploits the parallelism in applications processing large, regular data sets. With the open-source implementation of MapReduce Hadoop [2], a new software stack emerged for big data processing on inexpensive hardware. Hadoop provides a distributed file system (HDFS), which accommodates very large files, divided into chunks or data blocks of 64 or 128 MB in size. Data blocks are replicated at different compute nodes (DataNodes) and their locations are provided by a master node (NameNode).

The flow of a MapReduce computation can be split into three phases. First, each data block of the input data set is assigned to a *map task*, which generates key-value pairs as specified by a user-defined map function. Secondly, during the *shuffle phase*, the key-value pairs are sorted by key and divided among a number of *reduce tasks*. Finally, the reduce tasks, after receiving all key-value pairs with the same key, execute a user-defined reduce function.

To execute a MapReduce computation, a master node (JobTracker) assigns tasks based on the location of their input data blocks to different compute nodes (TaskTrackers). Despite the simplicity of the MapReduce programming model, today it is also common to create MapReduce programs from SQL-like higher-level programming languages (Pig Latin [25]). Although MapReduce can be employed for large jobs processing terabytes of data, most of the jobs in production clusters are short (minutes) SQL-like queries, with small input sizes (GBs) [7], [28].

# 3. SYSTEM MODEL

In this section, we propose a scheduling and provisioning structure for deploying multiple concurrent MapReduce frameworks, which can be resized dynamically at runtime.

## 3.1 Scheduling and Provisioning Structure

The system architecture assumed by MapReduce has resources where processors and storage are co-located. We assume these resources to be organized in multiple physical clusters that operate as a single distributed computing system managed by the resource manager FAWKES, which decides how resources are to be balanced across multiple MapReduce frameworks.

In Figure 1 we show the queuing system managed by FAWKES. FAWKES receives two types of requests, one for activating new MR-clusters, and one for executing MapReduce jobs (MR-jobs) that identify the MR-cluster in which they have to be executed. These requests are serviced in multiple queues, all using the FIFO scheduling discipline. The system queues requests for new MR-clusters in a global queue managed by FAWKES. The time required to activate a new MR-cluster consists of the time the MR-cluster has to wait to be deployed on physical resources ($w^C$) and the time required to load the input data set from an external persistent storage ($l^D$). Each active MR-cluster maintains an internal queue of MR-job requests targeted at it. The response time of an MR-job is equal to the sum of its waiting time in the MR-cluster's queue ($w^J$) and its execution time ($s^J$). If the weight of an MR-cluster in the FAWKES mechanism is equal to 0 indicating no task execution for at least a certain duration $T$ (see Section 4.2), FAWKES marks it as inactive, deallocates its resources, and removes it from the system. Prior to the removal of an MR-cluster, its (output) data is saved on the persistent storage in the time interval of length $b^D$.

## 3.2 Dynamic MR-Clusters

In order to balance resources across the active MR-clusters, FAWKES has to be able to resize them by growing and shrinking the number of resources allocated to them at runtime. In the traditional static deployment of the MapReduce frameworks, the data of the HDFS are replicated and distributed uniformly across the nodes. Then, techniques like delay scheduling [33] can maximize the number of tasks that achieve data locality. FAWKES has the ability of changing the allocations, and has a relaxed data locality model, through a new abstraction of the MapReduce framework that we call *dynamic MR-cluster*, which comprises three types of nodes.

The most important requirement for FAWKES is to provide reliable data management so that when nodes are removed from an MR-cluster and the number of replicas is small, no data are lost. However, creating numerous replicas is not desired because of the increased usage of storage space. Thus, when removing nodes from an active MR-cluster, FAWKES needs to replicate the data they store. As data-intensive applications are processing large volumes of data, its replication makes the resizing of the MR-cluster slow. To enable fast reconfigurations, the removed nodes of the MR-cluster should store relatively small amounts of data.

Similar to the static deployment of MapReduce, FAWKES permanently allocates to an MR-cluster an initial set of *core nodes* used for both executing jobs and storing (input and output) data. During the time an MR-cluster is active, FAWKES may temporarily increase its capacity by provisioning *transient* or *transient-core* nodes, which break the traditional model for data locality. The former are instantiated without local storage such that the jobs they execute read and write input/output data from/to core nodes. The latter are different from the core nodes only by the lack of input data, thus the jobs they execute also need to transfer input data from core nodes, but they can use the local storage to write output data. As a consequence, FAWKES can grow the size of an MR-cluster fast as no data movement will be involved at all. Shrinking the size of an MR-cluster by removing transient-core nodes does require saving the output data stored on them, the amounts of which is usually very small in comparison to the input data set distributed on the core nodes. Nevertheless, large fractions of these nodes may saturate the network (both) or increase the contention on the physical disks of the core nodes (especially the transient nodes). When to resize an MR-cluster, and whether then to use transient or transient-core nodes, is explained in Section 4. Different from our dynamic MR-cluster approach, Amazon Elastic MapReduce [1] does not support data redistribution when shrinking the size of the cluster.

# 4. BALANCED SERVICE LEVELS

In this section, we derive a fairness metric to determine the imbalance between concurrent MR-clusters (Section 4.1). Our scheduling system, FAWKES, targets weighted fair allocations, such that different MR-clusters converge to fractions of the system resources proportional to their weights (Section 4.2). FAWKES dynamically updates the weights based on different metrics exposed by the MR-clusters at runtime. We design three classes of weighting policies which consider the input to the MR-clusters (*demand*), the operation of the MR-clusters (*usage*), and the output of the MR-clusters (*runtime performance*) (Section 4.3).

## 4.1 Fairness

Fairness is a major issue in resource provisioning and job scheduling, and is an optimization target that may conflict with performance metrics such as response time [21], [33]. Although there exists a large volume of literature that analyzes the notion of fairness in communication systems [16], there is no generally agreed upon measure of the fairness between jobs. The fairness of a queuing system has been defined either as a measure of the time spent waiting [5], possibly with respect to job size [31]. A fairness metric which accounts for both job arrival times and sizes is proposed in [26]. We adapt the latter for evaluating the fairness of a provisioning policy.

The key element is the assumption that, at any moment of time $t$, the MR-clusters may be entitled to shares of the total data center capacity $C$, proportionally to their given weights. For an MR-cluster $i$, the difference between the fraction $c_i(t)$ of resources it currently has and the share of resources it should have based on its weight $w_i(t)$ (see Section 4.2.2) at moment $t$ is defined as its *temporal discrimination*

$$d_i(t) = c_i(t) - w_i(t). \qquad (1)$$

We define the *discrimination* of MR-cluster $i$ during a time interval $[t_1, t_2]$ by

$$D_i(t_1, t_2) = \int_{t_1}^{t_2} (c_i(t) - w_i(t))dt. \tag{2}$$

Setting $t_1 = d_i$ and $t_2 = r_i$ with $d_i$ and $r_i$ the moments of the request for the deployment and the removal of the MR-cluster, respectively, we obtain the *overall discrimination* of the MR-cluster.

When all the resources of the datacenter are occupied all the time, every positive discrimination is balanced with negative discrimination, and so $\sum_i D_i(t_1, t_2) = 0$ for any time interval $[t_1, t_2]$, which makes the expected mean value of the discrimination $E[D] = 0$. The fairness (or balance) of the system is given by the variance of the discrimination, which we call the *global discrimination factor*:

$$Var(D) = E[D^2] - E[D]^2 = E[D^2] \tag{3}$$

We consider the allocations of the MR-clusters to be *imbalanced* or *unfair*, when the global discrimination factor is larger than a predefined parameter $\tau$.

## 4.2 The Fawkes Mechanism

We want to provision resources to multiple MR-clusters in a single datacenter or multicluster system to give MR-clusters similar levels of service. To achieve this, we want to assign each MR-cluster a dynamically changing weight that indicates the share of the resources it is entitled to.

### 4.2.1 Admission Policy

For each MR-cluster $i$ Fawkes assumes that there is a minimum number of core nodes $m_i$ (and a corresponding minimum share), which may be set by a system administrator or computed based on the amount of data the cluster has to process. The system guarantees the minimum share of an MR-cluster as long as it has running jobs, even if according to its current weight it is only entitled to a smaller share.

If on the arrival of a new MR-cluster, the sum of its minimum share and of the minimum shares of the active MR-clusters exceeds 1, the new MR-cluster is queued (in FIFO order). Otherwise, the system gives it its minimum share of the resources within a time interval $T$ from its arrival, by shrinking the active MR-clusters which are above their minimum shares proportionally to their current weights (but not going below their minimum shares). When later an active MR-cluster finishes its workload and releases the resources it holds, Fawkes checks to see if the MR-cluster at the head of the queue fits. After a new MR-cluster receives its minimum share, the system monitors its state along with the states of the other active MR-clusters. The weights of the active MR-clusters are periodically updated after every interval of length $T$.

### 4.2.2 Changing Shares

To ensure that MR-clusters with different workloads experience similar service levels (e.g, job slowdown), we propose three complementary mechanisms, employed by Fawkes, which target either a subset or the entire set of the active MR-clusters, and operate at different timescales.

The MR-clusters collect periodically samples of different aspects of system operation, such as demand $d(t)$, resource utilization $r(t)$, or actual performance $p(t)$. Fawkes *monitors* a specific metric related to these aspects and sets the weight (*weighting mechanism*) $w_i(t)$ of MR-cluster $i$ at time $t$ to the average value of the samples $y_i$ collected during the last time interval $T$:

$$w_i(t) = \frac{y_i(t)}{\sum_{k=1}^{n} y_k(t)}, \tag{4}$$

where $n$ is the number of active clusters at time $t$.

After updating the weights, the temporal discriminations of the MR-cluster are determined as well. When the MR-clusters are imbalanced that is, the global discrimination factor exceeds the predefined threshold $\tau$, Fawkes changes their shares proportionally to their dynamic weights. To resize an MR-cluster, Fawkes employs the *grow and shrink mechanisms* based on provisioning temporary nodes, which can be either transient or transient-core (see Section 3).

The shrinking mechanism guarantees that the MR-clusters with positive discrimination reach their fair shares by releasing the surplus of resources they hold. Based on the type of nodes which are removed, we distinguish two possible ways of shrinking an active MR-cluster, *instant preemption* (IP) or *delayed preemption* (DP). The former is suitable for transient nodes and simply kills the currently running tasks, which are later re-scheduled by the MR-cluster. The latter applies to transient-core nodes, which besides removing their running tasks also require the replication of their local data. Fawkes removes transient-cores nodes in non-decreasing order of the amount of data they locally store.

The growing mechanism ensures that MR-clusters with negative discrimination achieve their fair shares by extending their current shares. To do so, the MR-cluster is grown either with transient (TR) or with transient-core (TC) nodes. The former have good performance only for compute intensive workloads, which generate small amounts of data. The local storage of the latter type of nodes significantly improves the performance of highly disk intensive workloads (Section 6). The type of growing employed by Fawkes is a predefined system parameter.

## 4.3 Weighting Policies

To balance the allocations of multiple MR-clusters, we investigate a comprehensive design space, which covers the input to the system, the state of the system, and the output (runtime performance) of the system. We focus, respectively, on the demand of the workloads submitted to MR-clusters, on the usage of resources allocated to them, and on the runtime performance. For each of these, we propose three exemplary policies.

For all policies we investigate in this work, the weights of the MR-clusters and the global discrimination are recomputed after every interval of length $T$. Only when the global discrimination exceeds a threshold $\tau$ are the allocations of the MR-clusters actually changed according to the new weights.

### 4.3.1 Demand-based Weighting

Demand-based weighting policies take into account the input to the system (the demand). They establish the fair shares of the MR-clusters as proportional to the sizes of the workloads submitted to their queues. As we do not assume any prior knowledge about the workloads, we identify three ways of defining the size of a workload at time $t$, viz. with respect to the number of waiting jobs, the size of the input data of the jobs, and the number of waiting tasks:

1. **Job Demand (JD).** The JD policy sets the demand of the MR-cluster to the total *number of jobs* waiting in the queue.

2. **Data Demand (DD).** The DD policy sets the weight of the MR-cluster to the total *input data volume* of the jobs waiting in the queue.

3. **Task Demand (TD).** The TD policy gives an estimate of the MR-cluster demand at finer granularity than JD, by taking into account the total *number of tasks* waiting in the queue.

Although each of these policies is inherently inaccurate, for example JD because the duration of jobs ranges from minutes to hours, we expect demand-based weighting policies to lead to better system performance than no policy.

#### 4.3.2 Usage-based Weighting

Usage-based policies monitor the state of the system; here, we propose policies that monitor the utilization of the physical resources currently allocated to MR-clusters. We identify two main resources to monitor, processor usage and disk usage, and derive three policies:

1. **Processor Usage (PU).** The PU policy sets the usage at time $t$ to the fraction of *utilized processing units* (cores or slots) from the total configured capacity of the MR-cluster.

2. **Disk Usage (DU).** The DU policy sets the usage at time $t$ to the ratio between the total *output data* generated by the MR-cluster and its current storage capacity.

3. **Resource Usage (RU).** The MU policy combines the previous two policies by accounting for *both compute and storage resources*, processor and disk, as follows:

$$u_i(t) = \psi \cdot u_i^P + (1 - \psi) \cdot u_i^D, \qquad (5)$$

where $u_i^P$ and $u_i^D$ are the (normalized) resource usages as computed by the PU and DU policies, respectively, and the parameter $\psi \in (0, 1)$ reflects the relative importance of the two resources.

#### 4.3.3 Performance-based Weighting

The performance-based policies assign the fair shares of the MR-clusters based on the performance of the system at runtime, so that MR-clusters with poor performance receive larger fractions of resources and, thus, improve their performance. We use in this work two performance metrics, slowdown (low values are ideal) and throughput (high values are ideal) to calculate the weights of the MR-clusters as follows:

1. **Job Slowdown (JS).** The JS policy calculates the slowdown of each running job at time $t$ as the ratio between the elapsed time since the job started and the job execution time on a reference static MR-cluster (only for this policy, assumed known at the start of the job in the MR-cluster). We consider the weights of the MR-clusters to be all equal and positive at time $t = 0$. The weight of the MR-cluster $i$ at time $t > 0$ is set to the *average job slowdown* of all jobs $s_i$ which are waiting in the queue.

2. **Job Throughput (JT).** The JT policy considers the performance of MR-cluster $i$ at time $t$ to be the ratio $q_i$ between the number of *jobs* completed and the total number of jobs waiting in the queue. The weight is, then:

$$p_i(t) = a^{-q_i(t)} \qquad (6)$$

where $a > 1$ is a constant (we set $a = 2$). The share of an MR-cluster $i$ is entitled to increases inversely proportional with the measured throughput from $C/a$ ($q_i \to 1$) to $C$ ($q_i \to 0$).

3. **Task Throughput (TT).** The TT policy is similar to the JT policy. The TT policy uses a throughput computed as the ratio between the number of *tasks* completed and the total number of tasks waiting in the queue. Eq. (6) still holds, with the ratio $q_i$ now referring to tasks instead of jobs.

We compare our policies with two *baselines*, **No policy (None)** and **Equal Shares (EQ)**. The former makes the MR-clusters run permanently on their minimum shares. For the latter, the available resources are always equally divided between the active MR-clusters.

## 5. EXPERIMENTAL SETUP

In this section, we present the experimental setup for assessing the performance of several aspects of system operation (Section 6) and of the full FAWKES mechanism for balancing resources across MR-clusters (Section 7). The main differences between our and previous experimental setup are the use of a comprehensive set of representative MapReduce applications (including a real, complex workflow), the design of five workloads (including several unfavorable cases), and the use of a multicluster testbed (only in one experiment). The total time used for experimentation exceeded 3 real months and over 60,000 hours system time.

### 5.1 Clusters

We run experiments on the Dutch six-cluster wide-area computer system DAS-4 [3]. The system has in total roughly 200 dual-quad-core compute nodes with 24 GB memory per node and 150 TB total storage, connected within the clusters through 1 Gigabit Ethernet (GbE) and 20 Gbps QDR InfiniBand (IB) networks. The compute nodes from different clusters communicate over dedicated 10 Gbps light paths provided by SURFnet. The largest cluster in terms of the number of nodes, situated at the VU Amsterdam, has roughly 70 nodes divided into 4 racks. The GbE interconnect is based on two 48-ports 1 GbE switches (symmetric, backplane cabled). The IB network is enabled by six 36-ports InfiniBand switches, organized in a fat tree, with 4 access switches, 2 at root. This architecture, which is useful for both data processing and high-performance computing, currently services about 300 scientists.

In our experiments, we restrict ourselves to well-connected datacenters and we use a standard setup of Hadoop-1.0.0 over InfiniBand. We configure the HDFS on a virtual disk device (with RAID0 software) that runs over 2 physical devices with 2 TB storage in total per node. The data are stored in the HDFS in blocks of 128 MB with a default replication factor of 3. With 8 cores per node enabled (no hyperthreading), we configure the TaskTrackers with 6 map slots and 2 reduce slots.

| Job | Type | Data | Input | Output |
|---|---|---|---|---|
| WC | compute | Random | 200 GB | 5.5 MB |
| ST | disk | Random | 200 GB | 200 GB |
| PR | compute | Random | 50 GB | 1.5 MB |
| KM | compute,disk | Random | 70 GB | 72 GB |
| TT | compute | BitTorrent | 100 GB | 3.9 MB |
| AH | disk,compute | BitTorrent | 100 GB | 90 KB |

Table 1: A summary of the MapReduce applications used in our experiments.

Real-world experimentation was greatly facilitated by the DAS-4 system. However, as the system is shared between many users, we also encountered practical restrictions. We have completed the set of micro-experiments presented in Section 6 reserving 20 up to 30 nodes for a week. As we explore a large design space experimentally, we summarize for the large experiments in Section 7 only results from single executions. It took more than 2 months to complete the macro-experiments we have designed in this paper.

## 5.2 MapReduce Applications

The choice of MapReduce applications is crucial for a meaningful experimental evaluation. We use both simple, synthetic *applications* from a popular MapReduce benchmark, HiBench [20] and a complex, real MapReduce-based *logical workflow*, BTWorld [17], [32]. Table 1 gives a high-level summary of these MapReduce applications, which we extend with a detailed description in this section.

### 5.2.1 The HiBench Benchmark

HiBench includes a suite of simple synthetic benchmarks for data transformation, web search, and machine learning, along with automatic tools for data generation:

1. **Wordcount (WC)** counts the number of occurrences of each word in a given set of input files. The map tasks simply emit key-value pairs with the partial counts of each word, and the reduce tasks aggregate these counts into the final sum. Wordcount is mostly compute intensive and shuffles a small number of bytes from map to reduce tasks.

2. **Sort (ST)** is a disk-intensive application in which the identity function stands as both map and reduce functions and the actual sorting is executed while the data is shuffled from map to reduce tasks.

3. **PageRank (PR)** is a link analysis algorithm widely used in web search engines to calculate the ranks of the web pages based on the number of reference links. The MapReduce implementation of the workload consists of three compute-intensive jobs which iteratively compute the ranking scores of all pages.

4. **K-Means (KM)** is a data mining clustering algorithm for multi-dimensional numerical samples. The workload employs two MapReduce jobs which resemble the characteristics of Wordcount and Sort. The former is mostly compute-intensive and iteratively computes the centroid of each cluster, thus swallowing a large fraction of the input. The latter is disk-intensive and reorders the data by assigning each sample to a cluster.

### 5.2.2 The BTWorld Workflow

BTWorld is a complex, real-world MapReduce-based logical workflow for processing the data collected periodically over many years from the global-scale peer-to-peer system BitTorrent [32]. The data set contains per tracker statistics (*scrapes*) stored in a multi-column layout which includes the identifier for the BitTorrent content (*hash*), the URL of the BitTorrent tracker (*tracker*), the time when the status information was logged (*timestamp*), the number of users having the full and part of the content (*seeders* and *leechers*), and the number of downloads at the moment of sampling (*downloads*).

A MapReduce-based workflow which currently consists of 14 high-level queries expressed in Pig Latin processes data and leads to understand the evolution over time of the global BitTorrent system. The queries expressed in this MapReduce workflow cover a broad range of SQL-like operators (e.g., join, aggregation, filtering, projection), break down into more than 20 MapReduce jobs, and exhibit three levels of data dependency: inter-query (when the input of the query needs to be generated by another query), inter-job (when a query is divided into several MapReduce jobs), and intra-job (between map and reduce tasks). The workflow combines both compute and disk intensive jobs with small ($10^{-6}$) and high ($10^{2}$) job selectivities, where job selectivity is defined as the ratio between the output and input sizes. Thus, this real workflow is very challenging for MapReduce-based data processing.

In our experiments, we use not only the complete workflow, but also two single queries individually:

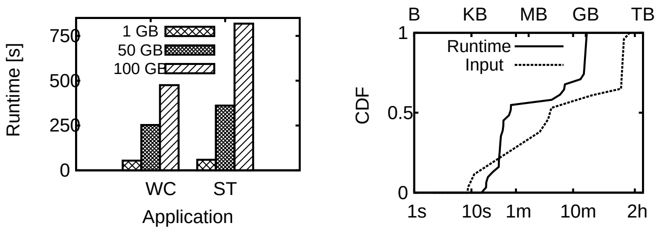
1. **TrackerOverTime (TT)** groups the input data set by tracker, sorts it by timestamp field, and applies different aggregation functions (e.g., count, avg, sum) on the remaining fields of the records. The query translates into a single, compute-intensive *map-heavy* job, and its output is 5 orders of magnitude smaller than the data set size.

2. **ActiveHashes (AH)** determines the number of active hashes in the system at every moment of time. The query is split in two MapReduce jobs, one disk-intensive with high (1) selectivity, and the other compute-intensive with very small ($10^{-6}$) selectivity. The first job emits all distinct hash and timestamp pairs to a second job, which further counts the number of unique hashes at every moment of time.

## 5.3 MapReduce Workloads

We consider workloads that cover many aspects (e.g., job types, data sizes, submission patterns) identified in synthetic benchmarks, production clusters, and BTWorld [20], [7]. To this end, we design 3 categories of workloads, based on which we generate 19 different workloads which we use in our micro- and macro-experiments:

1. **Single job - Single size (SS).** The SS workloads contain a number of identical synthetic or real-world jobs presented in Table 1. We use 6 such workloads of size one (one job) with fixed input data sizes (see Table 1) in Section 6. In Sections 7.1 and 7.2 we use SS workloads with 50 and 100 jobs, respectively, in which we employ the same submission pattern with all jobs submitted at once (*batch*), which is also used in many synthetic benchmarks.



(a) HiBench jobs (b) BTWorld jobs

Figure 2: The runtime performance of the jobs in HiBench and BTWorld on a 10-node static MR-cluster used as reference for the job slowdowns.

2. **Multiple jobs - Single size (MS).** The MS workloads combine several types of jobs (e.g., WC and ST) with the following input data sizes: 1 GB (small), 50 GB (medium), and 100 GB (large). We generate 3 workloads of this type based on WC and ST (Section 7.3, where we also describe the job arrival process) which have hundreds of small jobs.

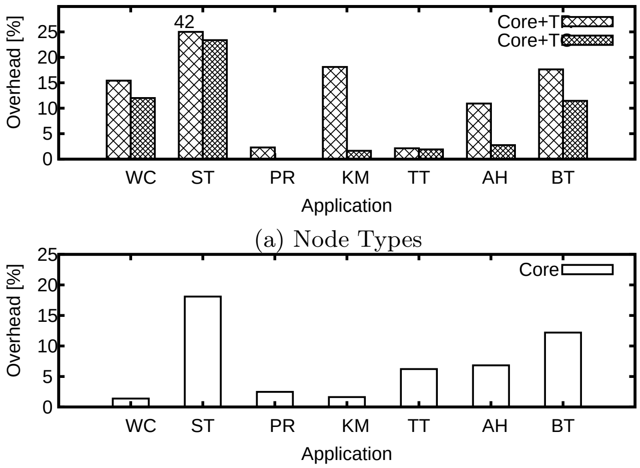
3. **Multiple jobs - Multiple sizes (MM).** The MM workloads combine several job types with different input data sizes which are summarized in Figure 2. The jobs in HiBench (e.g., WC and ST) have the same input sizes as in the MS workloads. BTWorld employs 26 jobs with 13 distinct input sizes. We use 3 instances of this type of workloads in which small jobs prevail (Section 7.3, including the arrival process).

Given the high imbalance between the workloads we use, the discrimination threshold does not have a significant impact, thus we set $\tau$ to a small value (10 in our experiments). In our workloads, most of the jobs take between 1 and 4 minutes to complete (Figure 2) and a couple of them arrive every minute. Thus, we set the weight update interval $T$ to a value in the order of a few minutes (1 or 2 minutes in our experiments).

We design the evaluation of Fawkes in two steps. First, we design four *micro-experiments* using SS workloads to assess different aspects of system operation (see Section 6). Then we design five *macro-experiments* using instances of all types of workloads to assess the performance of Fawkes. Towards this end, we combine highly imbalanced workloads to create extreme conditions of variable load across distinct MR-clusters (Section 7).

# 6. MICRO-EXPERIMENTS

In this section, we present the results of four experiments that each address a separate aspect of the performance of single MR-clusters. We investigate the performance of several MapReduce applications in single MR-clusters with different configurations with respect to the types of nodes (Section 6.1) and whether a single or multiple physical clusters are used (Section 6.2), and we assess the performance of growing (Section 6.3) and shrinking (Section 6.4) single MR-clusters at runtime. We measure the overhead of these configurations relative to a static MR-cluster with only core nodes. For all jobs in the micro-experiments we use the input data set sizes defined in Table 1.



(a) Node Types

(b) Multicluster Deployment

Figure 3: The overhead of running a single MR-job on 20-node MR-clusters with equal fractions of core and transient, and of core and transient-core nodes (a), and with resources evenly co-allocated from two physical clusters (b).

## 6.1 Node Types

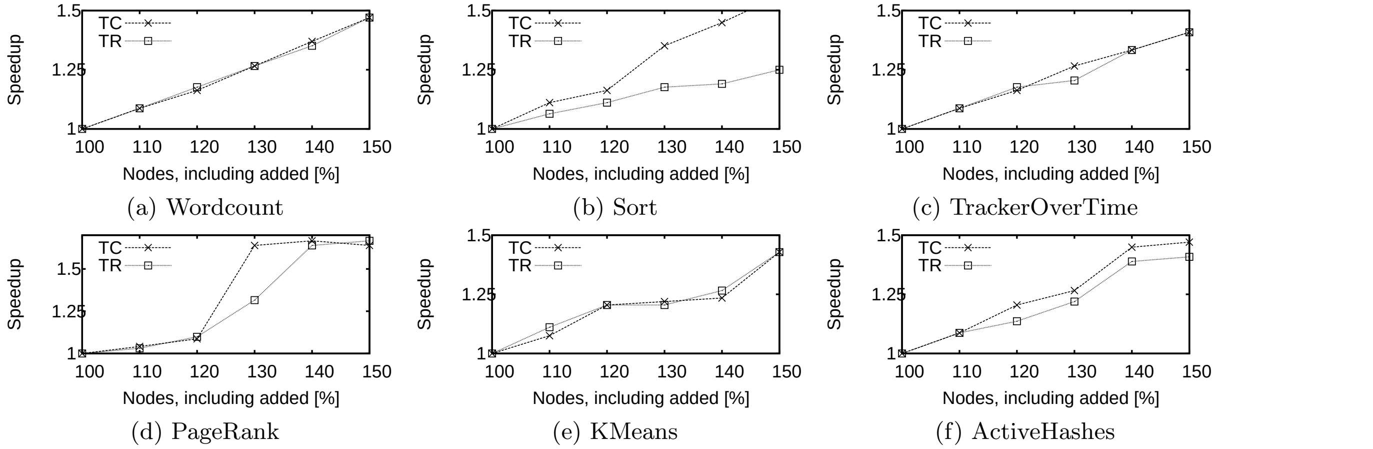
We assess the impact on the runtimes of jobs of using the three types of MR-cluster nodes presented in our system model in Section 3. In our previous work [12], we have found that the execution time of disk-intensive jobs increases with the ratio between transient and core nodes, while the performance of compute-intensive jobs is independent of the types of nodes.

**Using transient-core nodes instead of transient nodes reduces the overhead for disk-intensive jobs considerably (Figure 3a).** We set up static 20-node MR-clusters with only core nodes and with equal numbers of core and transient/transient-core nodes. In the former configuration, the input data set is distributed across all nodes of the MR-cluster, while in the latter two configurations, the input data set is distributed on 50% of the MR-cluster nodes. Figure 3a shows that with transient-core nodes instead of transient nodes, the overhead for disk-intensive jobs relative to the job execution on only core nodes is much smaller. In particular, Sort shows a significant improvement, decreasing the overhead from 40% with transient nodes to 23% with transient-core nodes, as do KMeans and ActiveHashes.

In our model, dynamically provisioning MR-clusters by means of a grow-shrink mechanism at runtime comes at the expense of poor data locality, as the tasks executed on transient or transient-core nodes need to transfer their input across the network. Nevertheless, we have shown here, at least in a cluster with a high-bandwidth network, that the impact of running non-local tasks can be limited by using transient-core nodes.

## 6.2 Multicluster Deployment

We assess the impact on job execution time of deploying single MR-clusters by co-allocating resources from different physical clusters in our multicluster system with high-speed wide-area connections. Previous work [10] has shown that co-allocation of parallel applications in multicluster systems is beneficial because of reduced job wait times if the overhead due to the slower wide-area communication is less than 25%.



(a) Wordcount

(b) Sort

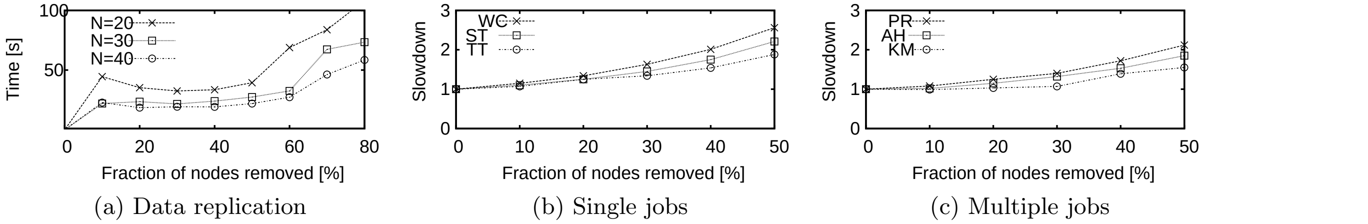
(c) TrackerOverTime

(d) PageRank

(e) KMeans

(f) ActiveHashes

Figure 4: The job speedup relative to the execution time measured on a static 20-node MR-cluster, when growing the MR-cluster with different fractions of transient (TR) or transient-core (TC) nodes.



(a) Data replication

(b) Single jobs

(c) Multiple jobs

Figure 5: The average per-node shrinking time of MR-clusters with $N$ nodes in total (a). The job slowdown relative to the execution time measured on a static 20-node MR-cluster, when shrinking a 20-node MR-cluster (b),(c).

**MapReduce jobs can run with low to moderate overhead in co-allocated MR-clusters over a high-speed interconnect (Figure 3b).** We set up 20-node static MR-clusters, with nodes co-allocated evenly from two physical clusters located at two universities in Amsterdam. Figure 3b shows that most of the applications exhibit low overhead when they run on co-allocated MR-clusters. For the complete BTWorld workflow and Sort, which are mostly composed by disk-intensive jobs, a co-allocated MR-cluster increases their execution times by less than 20% relative to the single physical cluster deployment.

Although we have shown here that MR-clusters may be provisioned with co-allocated resources, we design the remaining experiments within a single physical cluster.

## 6.3 Growing MR-clusters

We measure the speedup of single jobs when the MR-cluster grows with different fractions of transient-core or transient nodes before the job starts. The conveniently parallel layout of MapReduce applications [9] with only a single predetermined synchronization point between the map and reduce phases, in principle makes them malleable applications [11] that can benefit from dynamic resource provisioning at runtime [8].

**The execution time of MapReduce jobs can be improved with a growing mechanism at runtime by relaxing the data locality constraints.** We set up dynamic MR-clusters with 20 core nodes which we extend at runtime with different fractions of transient or transient-core nodes.

Despite the lack of data locality, transient (TR) nodes show good performance with one exception. For Sort, which is a highly disk intensive job, large fractions of TR nodes increase the contention on the physical disks of the core nodes (Figure 4b), thus limiting the speedup. With the relaxed data locality model of the transient-core (TC) nodes, jobs may write the data they generate on their local storage. This explains the linear increase of the applications speedups with the number of transient-core nodes (Figure 4). The supralinear speedup of PageRank is an anomaly due to the non-deterministic convergence of the iterative MapReduce jobs.

We can improve the performance of a broad range of MapReduce applications by relaxing the data locality model. Moreover, even in more extreme cases of no locality, transient nodes show good performance for applications which generate small amounts of data.

## 6.4 Shrinking MR-clusters

In this section, we investigate the overhead of reorganizing the data within HDFS (Figure 5a) and the job slowdown when different fractions of transient-core nodes are removed from the MR-cluster at the moment the job starts running (Figure 5b, 5c). Although in practice the transient-core nodes store less data than the core nodes, we assume in this micro-experiment a *worst-case scenario* in which both types of nodes store the same amounts of data.

**When resizing an MR-cluster to 50% of its size, the time overhead of reorganizing the data in HDFS increases linearly with the number of nodes removed.** We set up MR-clusters with different numbers of core and

|  |  |  |  |  | Job Types |  |  | Job Arrivals |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| Sec. | Workload | Nodes | Weight | Apps. | c-1 | c-2 | c-3 | c-1 | c-2 | c-3 |
| 7.1 | WKLD-A | TR | JD | WC | 50 x small |  |  | batch |  |  |
| 7.2 | WKLD-B | all | TD | WC | 90 x small | 5 x medium | 5 x large | batch |  |  |
|  | WKLD-C | all | TD | ST | 90 x small | 5 x medium | 5 x large | batch |  |  |
| 7.3 | WKLD-D | TC | all | WC,ST | 165 x small | 188 x all | 555 x small | average | bursty | average |
|  | WKLD-E | TC | TD | WC,ST,BT | 359 x all | 26 x all | 559 x small | average | sequential | average |

Table 2: The design space coverage of the macro-experiments presented in Section 7. For each experiment, the table summarizes the provisioning policy (node type and weighting policy) employed by the resource manager, and the workload instances (application type and job types, sizes, and arrival pattern) submitted to three concurrent MR-clusters.

transient-core nodes. The former represent 20% of the cluster size and each node of the cluster stores 10 GB of data. There are no running jobs while the MR-clusters are resized.

We find the average per-node removing time is constant when the MR-cluster is shrunk with up to 50% of its total size, and increases exponentially for larger fractions of transient-core nodes removed, as more data are replicated on fewer nodes (Figure 5a).

**When shrinking an MR-cluster at runtime, the job runtime is determined by the total size of the replicated data.** We set up 20-node MR-clusters with 10 core nodes which we shrink at runtime by different fractions of transient-core nodes. Figures 5b and 5c show that shrinking MapReduce applications at runtime increases the job slowdown linearly with the number of transient-core nodes removed. However, we observe that less compute-intensive jobs (e.g., ST and WC), which run on 200 GB, have higher slowdown than more computational intensive jobs (e.g., TT, PR), which run on less than 100 GB.

As MapReduce is usually employed for data-intensive applications, it is important to reduce the overhead of data replication by limiting the frequency of MR-cluster reconfigurations and by removing nodes with smaller data volumes.

# 7. MACRO-EXPERIMENTS

In this section, we evaluate Fawkes's resource provisioning and balancing mechanisms. Towards this end, we design a comprehensive set of scenarios, summarized in Table 2 w.r.t. to both system operation (e.g., nodes and weights) and experiment instrumentation (e.g., applications and workloads). We show how Fawkes effectively provisions newly arriving MR-clusters (Section 7.1) and achieves good balancing when the workloads are imbalanced (Section 7.2). Moreover, even under extreme imbalance and unfavorable conditions, we show evidence of up to 25% improvement of average job slowdown (Section 7.3).

## 7.1 Arriving MR-clusters

In this section, we show how Fawkes balances idle resources across the active MR-clusters and gracefully shrinks them to make space for new MR-cluster deployments.

Fawkes **effectively uses its grow and shrink mechanisms to dynamically provision multiple arriving MR-clusters.** Given 60 resources, Fawkes receives requests for 3 MR-clusters, at intervals of 100 s. All MR-clusters store 50 GB of data on their minimum shares of 10 core nodes. Fawkes uses transient (TR) nodes and employs the JD weighting policy (see Section 4.2). The weights are updated every $T = 60$ s. We combine 3 instances of the SS workload (see Section 5.3) into WKLD-A (see Table 2).

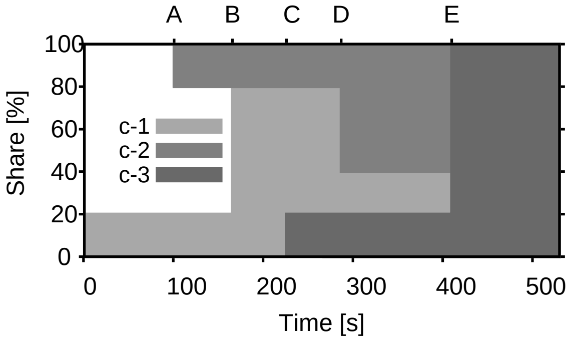


Figure 6: The distribution of the resources across three MR-clusters (c-1, c-2, c-3) arriving at different moments in time. (Points A-E explained in Section 7.1.)

In Figure 6, Fawkes initially provisions 10 core nodes to c-1. While c-1 loads its data, c-2 arrives and receives 10 core nodes from remaining 50 idle resources (A). MR-cluster c-1 starts running jobs when c-2 is still loading the data. Thus, all 40 remaining resources are allocated to c-1 (B). Later MR-cluster c-1 is successively shrunk to make space for c-3 (C) and to allow the share of c-2 to grow (D). When both c-1 and c-2 finish their workloads, c-3 grows to the full capacity of the system (E).

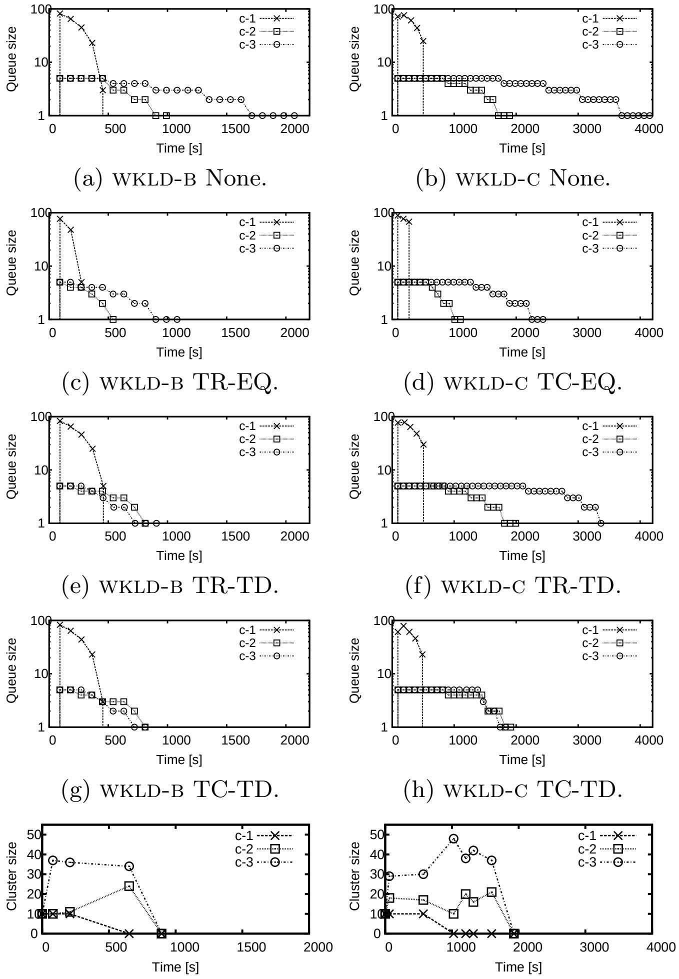
With a static partitioning approach, when the system is fully utilized, requests for new MR-clusters need to wait for active MR-clusters to complete their workloads and release the resources. Dynamic provisioning allows new MR-clusters to be deployed even when the active MR-clusters use the entire system capacity.

## 7.2 Growing and Shrinking MR-clusters

In this section, we show the impact of the type of nodes (transient or transient-core) and the type of workload on Fawkes's balancing mechanism.

Fawkes **is able to balance the allocations for disk intensive workloads with TC growing and DP shrinking, but fails to do so when using TR growing and IP shrinking (all defined in Section 4.2).** We consider 60 resources in total, which Fawkes uses to deploy simultaneously 3 concurrent MR-clusters with 10 core nodes to which we submit workloads WKLD-B and WKLD-C which differ by the growing (TC or TR) and the application (WC or ST) type (see Table 2). The weights are updated every $T = 60$ s. In Figure 7, we show the queue sizes of the three MR-clusters over time for None, EQ and TD policies (all defined in Section 4.3).

Apparently, neither the queue size nor the makespan of MR-cluster c-1 with small jobs is affected by growing or shrinking. Whereas Fawkes balances the medium and large



(a) WKLD-B None. (b) WKLD-C None.

(c) WKLD-B TR-EQ. (d) WKLD-C TC-EQ.

(e) WKLD-B TR-TD. (f) WKLD-C TR-TD.

(g) WKLD-B TC-TD. (h) WKLD-C TC-TD.

(i) WKLD-B TC-TD. (j) WKLD-C TC-TD.

Figure 7: The queue and cluster sizes of three MR-clusters running workloads WKLD-B and WKLD-C (Table 2) for different growing and weighting types.

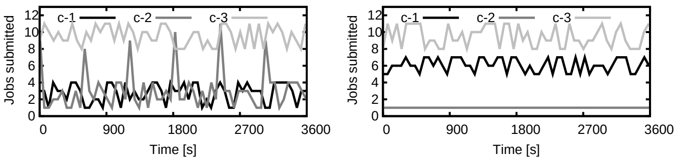
WC workloads with both TR and TC nodes (MR-clusters c-2 and c-3 in Figures 7e and 7g), the mechanism is not effective for ST-based workloads with TR nodes (MR-cluster c-3 in Figure 7f). For the latter scenario, the data volumes shuffled by the large ST jobs (MR-cluster c-3) increase the execution overhead as we have shown in Section 6. With TC growing, FAWKES balances the allocations even for highly disk intensive workloads (MR-clusters c-2 and c-3 in Figure 7h).

Without the dynamic growing and shrinking, the resources released once an MR-cluster executes its workload remain idle. Instead, FAWKES allocates the unused capacity to provision the active MR-clusters according to their weights, thus reducing the imbalance and the makespan.

## 7.3 Weighting MR-clusters

In this section, we assess the balancing properties of FAWKES in two scenarios with three MR-clusters running extremely imbalanced workloads.

FAWKES **balances the allocations to workloads even in very unfavorable situations.** In both scenarios, we reserve 48 resources to deploy simultaneously three MR-clusters with 10 core nodes and 200 GB of data each. FAWKES updates the weights of the MR-clusters every $T = 120$ s and provisions them with TC nodes. The two scenarios we analyze use workloads WKLD-D and WKLD-E (Table 2) with the arrival processes depicted in Figures 8a and 8b. WKLD-D mixes WC and ST jobs in two MS instances submitted to c-1 and c-3 and one MM instance submitted to c-2 and has bursts with large jobs in cluster c-2. WKLD-E submits the complete BTWorld workflow to c-2 as an MM instance and mixes WC and ST jobs submitted to c-1 and c-3 as MM and MS instances, respectively, with c-3 permanently having a much higher load than c-1 and c-2. Both workloads are imbalanced, with the ratio between the average number of tasks executed in the clusters with the highest and lowest loads being 3 and 8, respectively.



(a) WKLD-D. (b) WKLD-E.

Figure 8: Arrival processes (numbers of submissions per 2-minute intervals) for the two highly imbalanced workloads WKLD-D and WKLD-E (see Table 2).

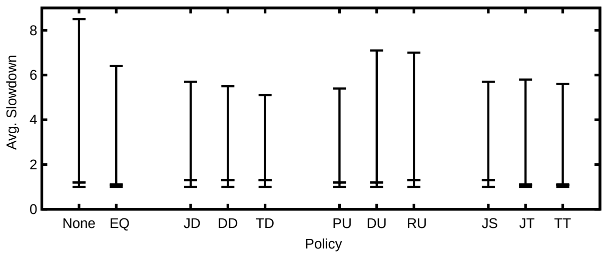
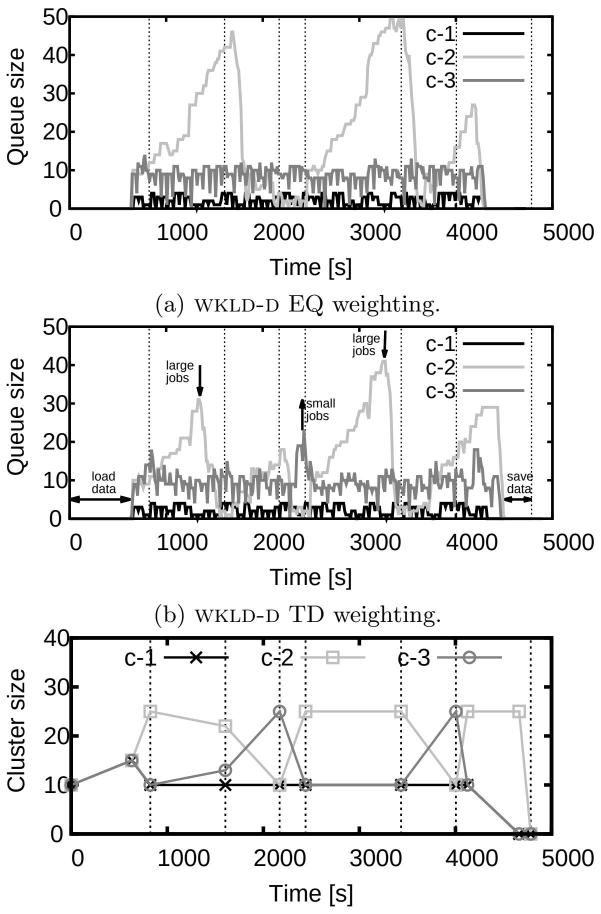


Figure 9: The minimum, the maximum, and the median of the average job slowdowns of three MR-clusters with workload WKLD-D for all weighting policies.

In Figure 9 we compare in the first scenario the weighting policies w.r.t. the average job slowdown measured for each MR-cluster running WKLD-D. For the demand-based policies (JD, DD, TD), the finer the granularity of calculating the queue sizes (number of tasks with TD) is, the more balanced the workloads of the MR-clusters are (25% improvement of job slowdown compared to None). FAWKES achieves the best improvement of the average job slowdown of 25% with the TD policy. Furthermore, we observe that FAWKES reduces the average job slowdown in the most loaded cluster (c-2) without significant impact on the performance of the low demand clusters (c-1 and c-3). From the usage-based policies (PU, DU, RU), only PU performs reasonably well. RU ($\psi = 0.5$) and its derivative DU ($\psi = 0$) are not effective because of some small jobs which generate large amounts of data, yet are completed relatively fast (see ST in Figure 2a). Counterintuitively, the performance-based policies (JS, JT, TT) do not outperform the demand-based policies. The reason is that in WKLD-D, small WC and ST jobs, which have the same runtimes (see Figure 2), prevail. Therefore, the weights assigned with the performance-based policies are similar and do not reflect the actual imbalance between the workloads.

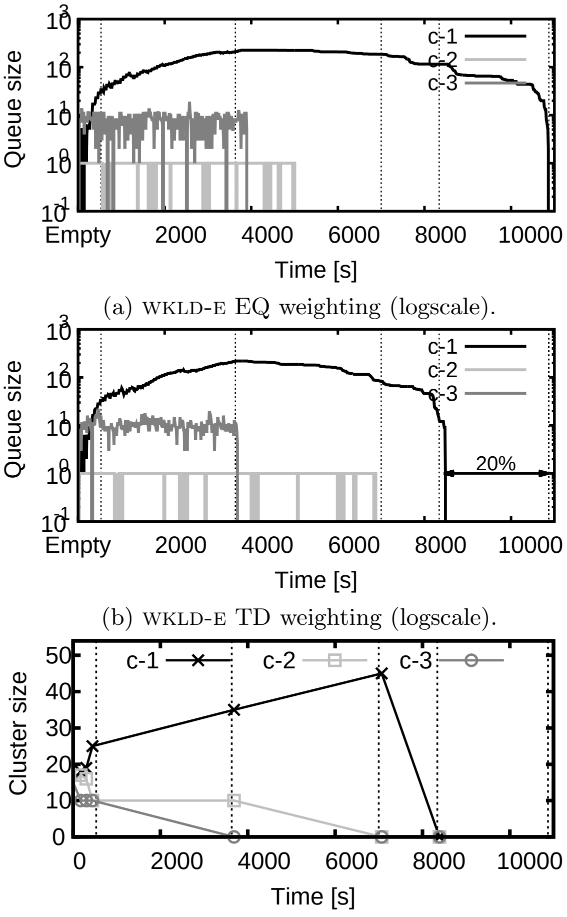


(a) WKLD-D EQ weighting.

(b) WKLD-D TD weighting.

(c) WKLD-D TD weighting.

Figure 10: The performance of TD weighting for WKLD-D.



(a) WKLD-E EQ weighting (logscale).

(b) WKLD-E TD weighting (logscale).

(c) WKLD-E TD weighting.

Figure 11: The performance of TD weighting for WKLD-E.

Figures 10 and 11 show the queue and cluster sizes with the weighting policy that performs best in the first scenario, TD, and the EQ baseline, for scenario 1 and 2, respectively. FAWKES makes 16 and 5 reconfigurations (excluding MR-cluster deployments and deallocations) in the two scenarios, see Figure 10c and 11c. For WKLD-D we notice that MR-clusters are reconfigured when bursts of large jobs arrive at c-2 or many small jobs are submitted to c-3. For WKLD-E, the most loaded MR-cluster (c-1) acquires most of the system resources, thus reducing the overall makespan of the experiment. In both cases, FAWKES is as effective as it can be because it moves (almost) all resources that it can move to the cluster with the highest load (c-2 in scenario 1 and c-1 in scenario 2), almost always leaving the other clusters at their minimum shares.

## 8. RELATED WORK

In this section, we summarize the related work from four aspects: resource sharing mechanisms for multicluster environments, malleability of parallel applications, cluster sizing problems, and fair-sharing provisioning policies.

**Resource Sharing.** To simplify cluster programming, a diverse array of specific frameworks for big data processing has been developed. Having multiple such frameworks within the same cluster or datacenter infrastructure requires a high-level resource management layer. Closest to our work are Yarn [29], Mesos [19], and Koala [23], all having the same design goal of sharing resources between multiple cluster computing frameworks. Yarn considers the resource requests received from the applications, the resource availability, and the internal scheduling policies to dynamically allocate resources to different frameworks. Heavy users or large jobs are prevented from monopolizing the cluster through system partitioning which can be employed with the Capacity [4] and Fair schedulers [33]. Mesos takes a different approach to resource allocation by delegating the scheduling decisions to the frameworks, rather than to individual applications. To reduce the overhead of the distributed scheduling, Mesos allows frameworks to specify through filters which are the resources they will never accept. Koala is a resource manager which co-allocates processors, possibly from multiple clusters, to various HPC applications and to isolated MapReduce [12] frameworks. When resources are available, each framework may receive additional resources from Koala, but it is their decision to accept or reject them.

FAWKES is fundamentally different from Yarn and Mesos. While the latter target near-optimal data locality, FAWKES achieves performance isolation and balanced resource allocations by relaxing the strict data locality assumptions through a fast and reliable grow and shrink mechanism. Furthermore, instead of the request and offer-based approaches, FAWKES controls the scheduling by observing the jobs submitted within each framework. Although FAWKES mechanism can be incorporated into Koala, we are currently maintaining and using it as a standalone research prototype.

**Malleable Applications.** To improve resource utilization, jobs which can be executed on a variable number of processors have emerged. When the number of processors assigned to a job can be increased or decreased by the scheduler at runtime, the job is called malleable [6], [11]. There are two ways to enable job malleability in parallel applications, either by creating a large number of threads equal to the cluster capacity, coupled with a multiplexing mechanism, or by inserting application specific code at synchronization points to repartition the data when the allocation changes. The poor performance of the former approach and the additional coding effort of the latter have limited the popularity of exploiting the job malleability for (tightly coupled) parallel applications. However, certain parallel applications based on the master-slave programming model, in which processors are required to execute relatively small and independent units of computations from a central scheduler (e.g., MapReduce [9]), can use malleability relatively easy. Moreover, MapReduce applications can be accelerated by using Amazon's spot instances, despite their variable and unknown lifetime [8].

Novel in this paper, instead of dynamically changing the allocations of single jobs [6], we exploit malleability of sets of MapReduce jobs, by growing and shrinking the framework itself. To do so, we propose a data-oriented mechanism in order to gracefully remove nodes from MapReduce frameworks at runtime. Towards this goal, we relax the traditional data locality constraints and we provision the MapReduce frameworks with temporary nodes that retrieve their input data for the tasks they run from core nodes.

**Cluster Sizing.** Cluster sizing problems are notoriously difficult because of the large parameter space related to the number of resources, the type of resources, and the job configuration. GreenHadoop [15] is mainly powered by solar energy source and uses the electrical grid only as backup. The scheduler minimizes the consumption of electrical energy by allowing MapReduce jobs to use more resources when green energy is available and less resources during peaks of electrical energy costs. Elastizer [18] provides an offline automated technique to solve cluster sizing problems using job profiles and simulations to search through the parameter space. Datacenters may benefit from a dynamic right-sizing mechanism that limits the number of active servers during periods of low load. Towards this end, optimization-based models and corresponding online algorithms for capacity provisioning in power proportional datacenters have been proposed and analyzed [22], [30].

Although we do not optimize for energy consumption, we investigate a similar problem of dynamic right-sizing a MapReduce cluster, but in a different setting. FAWKES attempts to find the fair share of each MapReduce cluster relative to the service levels of other concurrent clusters.

**Fair Allocations.** Fair-sharing algorithms have been explored in networking and operating systems domains for decades (see [33] and references therein). Datacenter schedulers like Fair [33] and Quincy [21] provide fairness for a single resource type by maximizing the minimum allocation (*max-min fairness*) received by each user in the system. To provide fairness in more general settings in which jobs may have heterogeneous requirements and hard placement constraints, the max-min fairness model has been extended to support multiple resource types [13] and to satisfy users constraints [14]. Pisces [27] is a datacenter scheduler that isolates the performance of multiple users of a shared key-value storage and provides max-min fairness. Pisces employs weighted fair-sharing and combines complementary mechanisms (partition placement, weight allocations, replica selection, and weighted fair queuing) which operate on per-application requests. A general framework that enables weighted proportional allocations for user differentiation is analyzed from a theoretical perspective in [24].

Unlike the former schedulers, FAWKES operates at the framework level, maintains a global view of the system by observing the jobs during their lifetime, and assigns to each framework a dynamically changing weight. In this paper, we propose three elements to differentiate MapReduce frameworks at runtime, viz. based on demand, on usage, and on performance.

## 9. CONCLUSION

Isolating the performance of multiple time-varying MapReduce workloads is an attractive yet challenging target for many organizations with large-scale data processing infrastructures. Towards this end, we have presented FAWKES, a mechanism for balancing the allocations of multiple MapReduce instances such that they experience similar service levels. FAWKES is based on the MR-cluster, a new abstraction for deploying MapReduce instances on physical resources which assumes the usual data locality constraints for a set of core nodes, but relaxes these constraints for nodes outside the core. For the fair-sharing problem, FAWKES employs weighted proportional allocations. The specific provisioning policies assign dynamic weights to different MR-clusters that take into account their dynamic load conditions.

In this paper, we take an experimental approach to provisioning multiple MR-clusters in a datacenter or multicluster system. With our micro-experiments we find that a relaxed data locality model has a limited impact on the application performance. Furthermore, our macro-experiments show that FAWKES delivers good performance and balanced resource allocations, even in unfavorable conditions of highly imbalanced workloads.

## 10. ACKNOWLEDGMENT

We thank Kees Verstoep of VU Amsterdam for his help with configuring and fixing software issues in the DAS-4 system. This research was supported by the Dutch national program COMMIT and STW/NWO Veni grant 11881.

## 11. REFERENCES

[1] Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce.

[2] Apache Hadoop. http://hadoop.apache.org.

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

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

[5] B. Avi-Itzhak and H. Levy. On Measuring Fairness in Queues. *Advances in Applied Probability*, 36(3), 2004.

[6] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema. Scheduling Malleable Applications in Multicluster Systems. *IEEE Cluster*, 2007.

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

[8] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz. See Spot Run: Using Spot Instances for MapReduce Workflows. *HotCloud*, 2010.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. *Comm. of the ACM*, 51(1), 2008.

[10] C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, and A. Streit. On Advantages of Grid Computing for Parallel Job Scheduling. *CCGrid*, 2002.

[11] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. *JSSPP*, 1997.

[12] B. Ghit, N. Yigitbasi, and D. Epema. Resource Management for Dynamic MapReduce Clusters in Multicluster Systems. *MTAGS*, 2012.

[13] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. *NSDI*, 2011.

[14] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-Min Fair Sharing for Datacenter Jobs with Constraints. *Eurosys*, 2013.

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

[16] A. G. Greenberg and N. Madras. How Fair Is Fair Queuing. *JACM*, 39(3), 1992.

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

[18] H. Herodotou, F. Dong, and S. Babu. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-Intensive Analytics. *SOCC*, 2011.

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

[20] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The Hibench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. *ICDEW*, 2010.

[21] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. *SIGOPS*, 2009.

[22] M. Lin, A. Wierman, L. Andrew, and E. Thereska. Dynamic Right-Sizing for Power-Proportional Data Centers. *INFOCOM*, 2011.

[23] H. Mohamed and D. Epema. Koala: A Co-allocating Grid Scheduler. *Concurrency and Computation: Practice and Experience*, 20(16), 2008.

[24] T. Nguyen and M. Vojnovic. Weighted Proportional Allocation. *SIGMETRICS*, 2011.

[25] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. *SIGMOD*, 2008.

[26] D. Raz, H. Levy, and B. Avi-Itzhak. A Resource-Allocation Queueing Fairness Measure. *SIGMETRICS/PERFORMANCE*, 2004.

[27] D. Shue, M. J. Freedman, and A. Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. *OSDI*, 2012.

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

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

[30] K. Wang, M. Lin, F. Ciucu, A. Wierman, and C. Lin. Characterizing the Impact of the Workload on the Value of Dynamic Resizing in Data Centers. *SIGMETRICS*, 2012.

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

[32] M. Wojciechowski, M. Capotă, J. Pouwelse, and A. Iosup. BTWorld: Towards Observing the Global BitTorrent File-Sharing Network. *HPDC*, 2010.

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