Self-Balancing Job Parallelism and Throughput in Hadoop

Bo Zhang[1], Filip Křikava[2(✉)], Romain Rouvoy[1], and Lionel Seinturier[1]

[1] University of Lille/Inria, Villeneuve-d'ascq, France
{bo.zhang,romain.rouvoy,lionel.seinturier}@inria.fr
[2] Czech Technical University, Prague, Czech Republic
krikava@gmail.com

Abstract. In Hadoop cluster, the performance and the resource consumption of MapReduce jobs do not only depend on the characteristics of these applications and workloads, but also on the appropriate setting of Hadoop configuration parameters. However, when the job workloads are not known *a priori* or they evolve over time, a static configuration may quickly lead to a waste of computing resources and consequently to a performance degradation. In this paper, we therefore propose an on-line approach that dynamically reconfigures Hadoop at runtime. Concretely, we focus on balancing the job parallelism and throughput by adjusting Hadoop capacity scheduler memory configuration. Our evaluation shows that the approach outperforms vanilla Hadoop deployments by up to 40 % and the best statically profiled configurations by up to 13 %.

1 Introduction

Along the years, Hadoop has emerged as the *de facto* standard for big data processing and the MapReduce paradigm has been applied to large diversity of applications and workloads. In this context, the performance and the resource consumption of Hadoop jobs do not only depend on the characteristics of applications and workloads, but also on an appropriately configured Hadoop environment. Next to the infrastructure-level configuration (*e.g.* the number of nodes in a cluster), the Hadoop performance is affected by job- and system-level parameter settings. Optimizing the job-level parameters to accelerate the execution of Hadoop jobs has been a subject to a lot of research work [2,9,13–16].

Beyond job-level configuration, Hadoop also includes a large set of system-level parameters. In particular, YARN (*Yet Another Resource Negotiator*), the resource manager introduced in the new generation of Hadoop (version 2.0) defines a number of parameters that control how the applications (*e.g.* MapReduce jobs) are scheduled in a cluster which influence jobs performance. Among YARN parameters, the MARP (*Maximum Application Master Resource in Percent*: yarn.scheduler.capacity.maximum-am-resource-percent) property directly affects the level of MapReduce job parallelism and associated throughput. This property balances the number of concurrently executing MapReduce jobs versus

© IFIP International Federation for Information Processing 2016
Published by Springer International Publishing Switzerland 2016. All Rights Reserved
M. Jelasity and E. Kalyvianaki (Eds.): DAIS 2016, LNCS 9687, pp. 129–143, 2016.
DOI: 10.1007/978-3-319-39577-7_11

the number of the corresponding map/reduce tasks. An inappropriate MARP configuration will therefore either reduce the number of jobs running in parallel resulting in idle jobs, or reduce the number of map/reduce tasks and thus delay the completion of jobs. However, finding an appropriate MARP value is far from trivial. On the one hand, the diversity of MapReduce applications and workloads suggests that a simple, *one-size-fits-all* application-oblivious configuration, will not be broadly effective—*i.e.* one MARP value that works well for one MapReduce application/workflow combination might not work for another [22]. On the other hand, YARN configuration is static and as such it cannot reflect any changes in workload dynamics. The only possibility is to do a *best-effort* configuration based on either experience or a static profiling in the case the jobs and workloads are known as *a priori*. However, (1) this might not be always possible, (2) it requires additional work, and (3) any unpredictable workload changes (*e.g.* a load peak due to node failures) will cause performance degradation.

In this paper, we therefore focus on dynamic MARP configuration. The main contributions are the following:

(1) an analysis of the effects of the MARP parameter on the MapReduce job parallelism and throughput, and
(2) a feedback control loop that self-balances MapReduce job parallelism and throughput.

Our evaluation shows that our approach systematically achieves better performance than static configurations. Concretely, we outperform the default Hadoop configuration by up to 40 % and up to 13 % for the *best-effort* statically profiled configurations, yet without any need for prior knowledge of the application or the workload shape, nor any need for any learning phase.

The rest of this paper is organized as follows. In Sect. 2, we introduce the architecture of YARN. The motivation of our research is placed in Sect. 3. Section 4 illustrates the memory and performance issues usually faced by Hadoop clusters. Section 5 describes the methodology we adopt and Sect. 6 evaluates our solution using various Hadoop benchmarks. We discuss related work in Sect. 7 before concluding in Sect. 8.

2 Overview of YARN

YARN is a cluster-level computing resource manager responsible for resource allocations and overall jobs orchestration. It provides a generic framework for developing distributed applications that goes beyond the MapReduce programming model. It consists of two main components (cf. Fig. 1): a per-cluster ResourceManager acting as a global computing resource arbiter and a per-node NodeManager responsible for managing node-level resources and reporting their usage to the ResourceManager.

Figure 1 depicts the architecture of YARN. The ResourceManager contains a scheduler that allocates resources for the running applications, like the JobTracker in previous version of Hadoop. However, ResourceManager does not

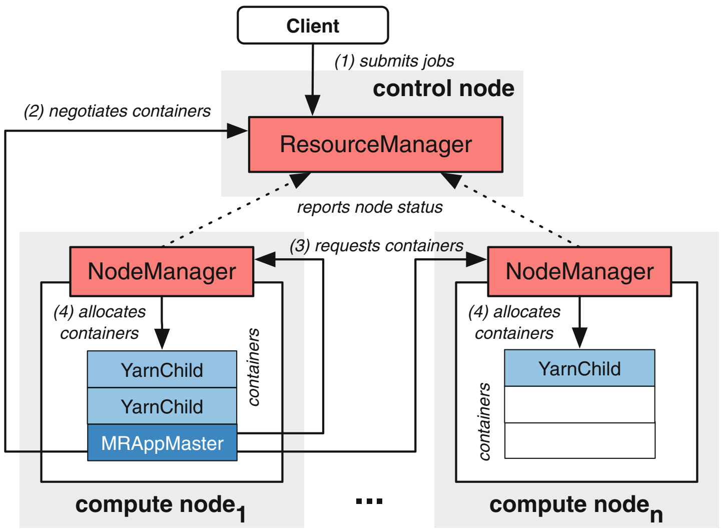


Fig. 1. High-level YARN architecture.

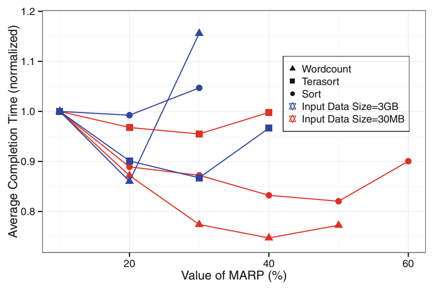
do any application monitoring or status tracking. This responsibility is left for the per-job instance of *Application Master* (AM). AM is an application-specific process that negotiates resources from the ResourceManager and collaborates with the NodeManager(s) to execute and monitor its individual tasks. The scheduling is based on the application resource requirements and it is realized using an abstract notion of *containers*. Essentially, each computing node is partitioned into a number of containers which are fixed-size resource blocks running AMs and their corresponding tasks.

3 Motivation

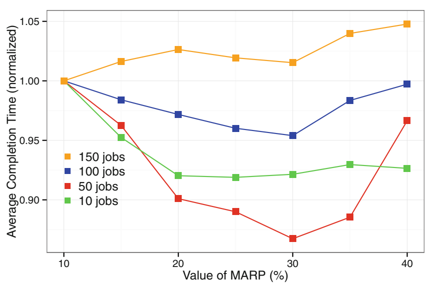
To understand the limitation of static configuration, we first study how the number of tasks to be processed and the MARP affects the overall completion time of Hadoop jobs. All experiments were performed using an Hadoop cluster made of 11 physical hosts[1] (1 control node and 10 compute nodes) deployed on the Grid5000 infrastructure. We use Hadoop 2.6.0.

Figure 2a reports on the completion time of the three applications provided by the HiBench benchmark suite [11]: Wordcount, Terasort, and Sort. For each of the input workloads—*i.e.* 30MB and 3GB—we observe the impact of the MARP parameter on the mean completion time of 100 jobs. To guarantee the comparisons visible, the values are normalized according to the absolute completion time of the vanilla Hadoop configuration—*i.e.* MARP = 0.1. The absolute completion time can be found at the paper web companion page: https://spirals-team.github.io/had-loop/DAIS2016.html

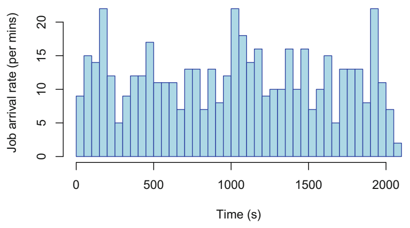
[1] 2 Intel Xeon L5420 CPUs with 4 cores, 15GB RAM, 298GB HDD.



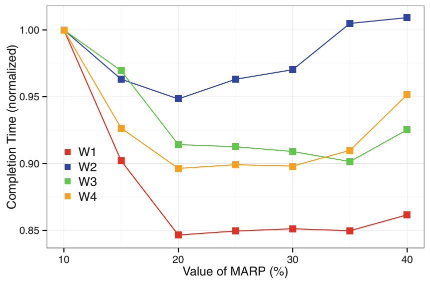
(a) Effects of different MARP configurations, job type and job size on mean completion time of 100 jobs.



(b) Effects of different MARP configurations and load peak stress on mean completion time.



(c) A job distributions generated by SWIM used for W1.



(d) Effects of different MARP configurations and different SWIM generated workloads on overall completion time.

Fig. 2. Effects of MARP and an example of job distributions in SWIM.

As expected, the vanilla configuration does not provide the best configuration for any of the workloads. Furthermore, one can observe that the best performance is not achieved by a single value of MARP, but rather tends to depend on the type and the size of the job. In particular, increasing the value of MARP—thus allowing more jobs running in parallel—tends to benefit the smaller Hadoop jobs, while large jobs complete faster when more resources is dedicated to the YarnChild containers which are responsible for processing requests.

Next, we stress the Hadoop cluster by running a different number of jobs in parallel in order to observe the impact of a load peak on the job mean completion time. Figure 2b shows the performance when running Terasort with 3GB workload under various stress conditions. Compared to Fig. 2a, one can observe that by increasing the number of concurrently running jobs, the optimal value of MARP differs from the previous experiment. Therefore, while a MapReduce job can be profiled for a *best-effort* MARP configuration in a specific Hadoop cluster, any unpredictable changes in the workload dynamics will lead to a performance degradation.

Finally, we consider heterogeneous workloads. Concretely, we use SWIM (*Statistical Workload Injector for Mapreduce*) [3] to generate 4 realistic MapReduce workload. SWIM contains several large workloads (thousands of jobs), with complex data, arrival, and computation patterns that were synthesized from historical traces from Facebook 600-nodes Hadoop cluster. The proportion of job sizes in each input workloads has been scaled down to fit our cluster size using a Zipfian distribution (see http://xlinux.nist.gov/dads/HTML/zipfian.html). (cf. Fig. 2c).

As previously observed for homogeneous workloads, Fig. 2d demonstrates that not a single MARP value fits all the workloads and the best configuration can only be set by having a deep understanding of the Hadoop jobs and their dynamics.

Synthesis. These preliminary experiments demonstrate that the MARP configuration clearly impacts Hadoop performances. They show that the default value is not optimal. While one can profile the different applications to identify the *best-effort* static configuration, we have shown that any unforeseen change in the workload dynamics can degrade the overall performance. We therefore advocate for a self-adaptive approach that continuously adjusts the MARP configuration based on the current state of the Hadoop cluster. In next section, we will analyse How MARP affects the system performance of the Hadoop cluster.

4 Memory Consumption Analysis

In this section, we focus on memory consumption (YARN can manage CPU and memory, but in this paper, we only consider memory) and analyze the causes of the performance bottlenecks.

In an Hadoop cluster, the memory can be divided into four parts: M_{system}, M_{AM}, M_{YC}, and M_{idle}. M_{system} is the memory consumed by the system components—*i.e.* ResourceManager, NodeManager in YARN and NameNode, DataNode in HDFS. M_{system} is constant in a Hadoop cluster.

The other three parts represents the memory held by NodeManager(s) as a result of processing MapReduce jobs:

M_{AM} is the memory allocated to all the MRAppMaster containers across all compute nodes. This is controlled by the MARP configuration—*i.e.* $M^*_{AM} = M_{compute} \times \text{MARP}$. During the processing of jobs, $M_{AM} \leqslant M^*_{AM}$.

M_{YC} is the memory used by all the YarnChilds to process map/reduce tasks across all the concurrently running jobs on all the computing nodes. This part directly impacts the job processing rate. A larger M_{YC} means that the more map/reduce tasks can be launched in parallel and the faster ongoing jobs are completed.

M_{idle} is the unused memory across all the computing nodes. High M_{idle} value together with pending jobs is a symptom of a waste of resources.

Their relationship with the overall computing memory of a Hadoop cluster, $M_{compute}$, can be expressed as follows: $M_{compute} = M_{AM} + M_{YC} + M_{Idle}$. Upon starting an Hadoop cluster, $M_{compute}$ is fixed (unless new computing nodes are enlisted or existing discharged from the cluster).

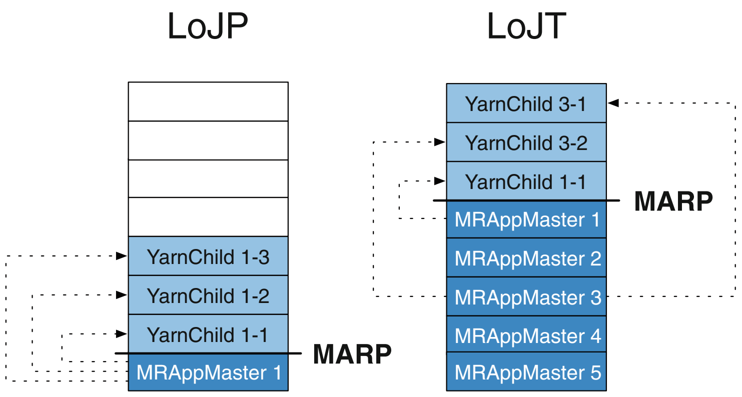


Fig. 3. LoJP and LoJT in Hadoop.

4.1 Loss of Jobs Parallelism

The maximum number of concurrently running jobs, N_{max}, in an Hadoop cluster is $N_{max} = \frac{M^*_{AM}}{M_{container}}$ where $M_{container}$ is the NodeManager container size (by default it is 1GB). The smaller the MARP value is, the smaller N_{max} will be and the less jobs will be able to run in parallel.

In the case the number of running jobs equals to N_{max}, all available application master containers are exhausted and ResourceManager cannot schedule any more jobs. Therefore, $M_{compute} = M^*_{AM}+M_{YC}+M_{idle}$. Where M_{idle} will emerge with a low N_{max}. When the number of running jobs reaches N_{max}, $M_{AM} = M^*_{AM}$ and no more pending jobs can be run even though $M^*_{AM} + M_{YC} < M_{compute}$. Therefore, we can observe that the lower $M^*_{AM} + M_{YC}$ is, the higher M_{idle} is. This indicates a memory / container waste that in turn degrades performances. We call this situation the *Loss of Jobs Parallelism* (LoJP). Figure 3 illustrates such a situation. An Hadoop cluster with 8 containers has the MARP value set too low, allowing only one job to be executed at a time. Any pending job have to wait until the current job has finished, despite the fact that some containers are unused.

4.2 Loss of Job Throughput

As shown in the previous section, small N_{max} limits the jobs parallelism within an Hadoop cluster. However, large N_{max} may also impact the job performance. By increasing N_{max} (or M^*_{AM}) in order to absorb M_{idle}, $M_{compute}$ can be rewritten as follow: $M_{compute} = M_{AM} + M_{YC}$.

In this case, when an Hadoop cluster processes a large number of concurrent jobs, M_{AM} becomes a major part of $M_{compute}$ and thus it limits M_{YC}. MRAppMaster is a job-level controller and it does not participate in any map/reduce task processing. Therefore, a limited M_{YC} decreases significantly the processing throughput of an Hadoop cluster. This symptom is identified as a *Loss of Job*

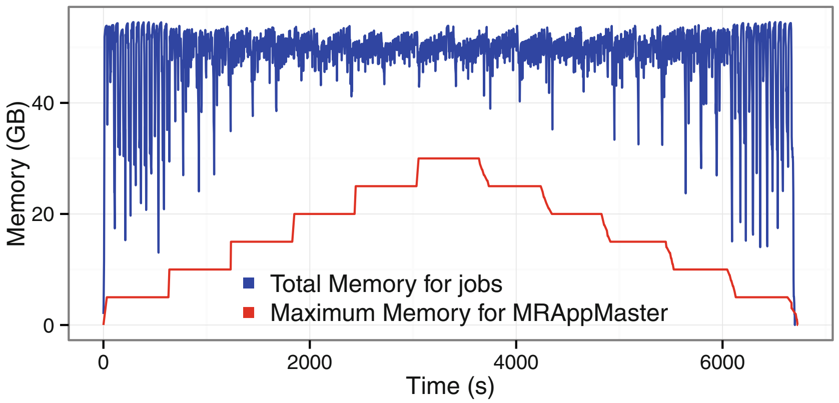


Fig. 4. Amplitude of memory drops depending on the MARP value.

Throughput (LoJT) and is also illustrated in Fig. 3. In this case, we have set the MARP too high, which allows many jobs to run in parallel, yet the actual processing capacity is limited by the low number of available container for running YarnChild.

4.3 Large Drops of Memory Utilization

Depending on the size of the jobs and the memory used in YarnChild containers, the dynamic allocation of resources can result in abruptly large drops of memory utilization (cf. Fig. 4). This is especially true when the tasks are rather fast to complete.

These memory drops usually appear at the end of concurrently running jobs. When a job comes to the end, all its corresponding M_{YC} will be quickly released. But its MRAppMaster is still running to organize data, and to report results to users. Due to the running MRAppMaster, idle jobs cannot get the permission to access memory for processing. Meanwhile, if other concurrently running jobs do not have enough unscheduled map/reduce tasks to consume these M_{idle} (released M_{YC}), the memory utilization will drop. A higher MARP value means more concurrently running jobs, which probably have more unscheduled map/reduce tasks to avoid the memory drops, and vice versa.

The memory drops cause temporarily high M_{idle}, and therefore reduce the average memory utilization—*i.e.* this phenomenon also contributes to performance degradation. Moreover, the frequent and large memory drops can also disturb the users to accurately detect the state of the Hadoop cluster.

5 Memory Consumption Balancing

Based on the previous section, we propose a self-adaptive approach for dynamically adjusting the MARP configuration based on the current state of the cluster.

5.1 Maximizing Jobs Parallelism

The symptom of LoJP—*i.e.* small N_{max}, large M_{idle} leading to decrease the memory utilization— can be detected from the ResourceManager component and fixed by increasing the MARP parameter. However, it should not consequently cause LoJT (cf. Section 4.2). We therefore propose a greedy algorithm to gradually increase the MARP parameter (cf. Algorithm 1). It is a simple heuristics that periodically increments MARP by a floating step (*inc*) until a given threshold (T_{LoJP}) is reached—*i.e.* the overall memory consumption $M_U = M_{AM} + M_{YC}$ falls below the threshold $M_U < T_{LoJP}$. Both, the current M_U and MARP values can be observed from ResourceManager. Once the increment becomes effective, ResourceManager will continue to schedule any pending jobs until the N_{max} limit is reached. A short delay between the increment steps (*delay*) is therefore required to let the cluster settle and observe the effects of the increment.

Algorithm 1. Fixing LoJP by incrementing MARP.

```
procedure LoJP($T_{LoJP}$, inc, delay)
    $M_U \leftarrow$ actual memory utilization
    if $M_U < T_{LoJP}$ then
        $MARP \leftarrow$ current MARP value
        $MARP \leftarrow MARP +$ inc
        RELOAD($MARP$)
        SLEEP(delay)
```

5.2 Maximizing the Job Throughput

The LoJT symptom is more difficult to detect since, at the first glance, the Hadoop cluster appears to fully utilize its resource. However, this situation can be also a result of the cluster saturation with too many jobs running in parallel. It therefore requires to better balance the resources allocated to M_{AM} and M_{YC}. Algorithm 2 applies another greedy heuristics to gradually reduce the amount of memory allocated to MRAppMaster by a floating step (*dec*) until we detect that the overall memory utilization (M_U) falls below the maximum memory utilization threshold T_{LoJT}.

To avoid an oscillation between the two strategies, we combine them in a double-threshold (T_{LoJP}, T_{LoJT}, where $T_{LoJP} < T_{LoJT}$) heuristic algorithm that ensures that they work in synergy (cf. Algorithm 3). When memory usage is higher than 0.9, it is enough to prove that LoJP disappears. Meanwhile, an stably over-high memory usage (*e.g.* 0.95) is probably caused by LoJT. The increment and decrement steps are not fixed. Instead, they are computed in each loop iteration based on the difference between the memory utilization and the target threshold. This allows the system to automatically achieve the translation between rapid and fine-gained tuning—*i.e.* if the M_U is near a threshold, the square root will be small, while shall the memory utilization be far from a threshold, the increment or decrement will be large.

Algorithm 2. Fixing LoJT by decrementing MARP.

```
procedure LoJT($T_{LoJT}$, dec, delay)
    $M_U \leftarrow$ actual memory utilization
    if $M_U > T_{LoJT}$ then
        $MARP \leftarrow$ current MARP value
        $MARP \leftarrow MARP -$ dec
        reload($MARP$)
        sleep(delay)
```

Algorithm 3. Balancing LoJP and LoJT.

```
procedure Balance(delay)
    $M_{compute} \leftarrow$ overall maximum memory
    $T_{LoJP} \leftarrow 0.9 \times M_{compute}$
    $T_{LoJT} \leftarrow 0.95 \times M_{compute}$
    loop
        $M_U \leftarrow$ actual memory utilization
        if $M_U < T_{LoJP}$ then
            LoJP($T_{LoJP}$, $\frac{\sqrt{T_{LoJP} - M_U}}{M_{compute}}$, delay)
        else if $M_U > T_{LoJT}$ then
            LoJT($T_{LoJT}$, $\frac{\sqrt{M_U - T_{LoJT}}}{M_{compute}}$, delay)
```

5.3 Handling Drops of Memory Utilization

Drops of memory utilization are caused by the completion of map/reduce tasks that release large blocks of memory. Such memory fluctuation can result in MARP oscillations when the Algorithms 1, 2 and 3 will be constantly scaling up and down the MARP value. To prevent this, we use a Kalman filter to smooth the input—*i.e.* the memory utilization. It helps to stabilize the value and eliminate the noise induced by the memory fluctuation [18]. Concretely, we apply a 1D filter defined as: $M(t + \delta_t) = A \cdot M(t) + N(t)$. where M refers to the state variable—*i.e.* the memory usage—A is a transition matrix and N the noise introduced by the monitoring process.

6 Evaluation

In this section, we evaluate the capability of our self-balancing approach to address the problem of MapReduce job parallelism and throughput. We start with an quick overview of the implementation of the self-balancing algorithm followed by a series of experiments. The evaluation has been done using a cluster of 11 physical hosts deployed on the Grid5000 infrastructure, the same as we used in Sect. 3. We use Hadoop 2.6.0. Additional configuration details and experiment raw values are also available at the paper web companion page.

6.1 Implementation Details

The implemation is based on the feedback control loop that implements the balancing algorithm introduced in the previous section. It follows the classical MAPE (*Monitor-Analyze-Plan-Execute*) decomposition [12].

The control loop is implemented in Java and runs on the control node alongside with YARN. The memory information are collected using the ResourceManager services. The MARP value is accessed via YARN configuration and is changed by the YARN ResourceManager admin client (`yarn rmadmin` command). For the Kalman filter, we used the jkalman library[2]. It can smooth the memory utilization to avoid unnecessary MARP adjustments. The completion time of one map task is about 10 seconds. It is a reasonable value for *delay* to ensure the capture of memory fluctuation.

6.2 Job Completion Time

We start the evaluation by running the same set of MapReduce benchmark as we did at the beginning in Sect. 3—*i.e.* Wordcount, Terasort and Sort from the HiBench benchmark suite, each with two datasets (30MB and 3GB). Figure 5 shows the mean job completion time of 100 jobs using, the vanilla Hadoop 2.6.0 configuration (MARP = 10 %), the *best-effort* statically profiled configuration where the values were obtained from our initial experiments (cf. Fig. 2a), and finally our self-balancing approach (dyn). The values were normalized to the vanilla configuration.

For each of the considered applications and workloads, our self-balancing approach outperforms both other configurations. Often the difference between the statically profiled configuration and our dynamic one is small. This is because the *best-effort* MARP value already provides a highly optimal configuration so the applications cannot execute much faster. The important thing to realize is

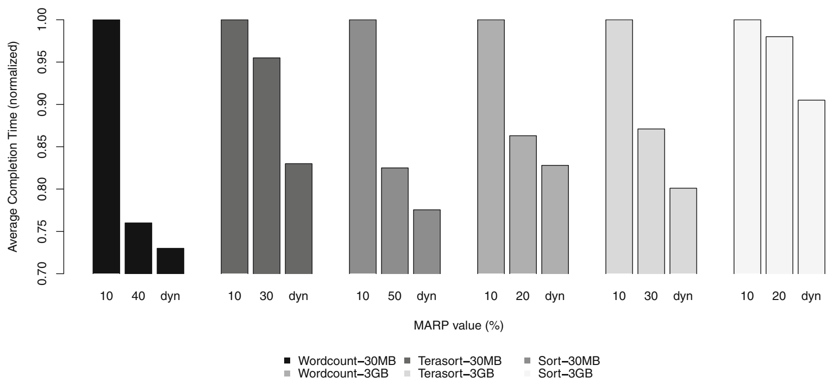
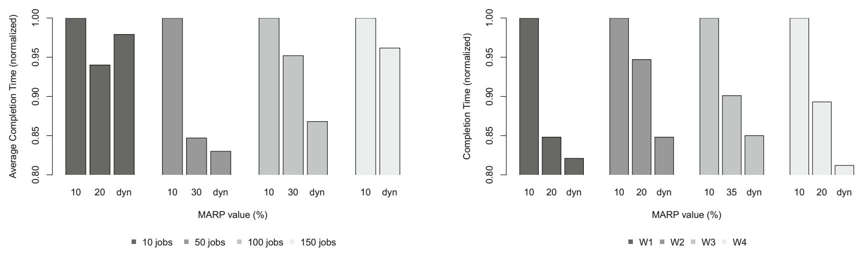


Fig. 5. Performance comparisons of 3 HiBench applications and 2 datasets.

[2] http://sourceforge.net/projects/jkalman.



(a) Performance comparisons of Terasort configured with 3GB under 4 workloads.

(b) Performance comparisons of 4 SWIM workloads.

Fig. 6. Performance comparisons

that our approach adapts to any application and does not require any profiling effort. It continuously finds a MARP configuration under which the application executes at least as fast as under the *best-effort* configuration.

Next, we evaluate how the approach performs under different workload sizes. Figure 6 shows the completion time of the Terasort with 3 GB input data size benchmark under varying number of concurrently running jobs—*i.e.* 10, 50, 100 and 150. In this case, the self-balancing algorithm outperforms the other configurations in all but the first case of a small number of jobs. The reason is that our solution always starts with the default MARP configuration which is 10 % and converges towards the optimal value (20 % in this case) along the execution. However, the overall completion time of the 10 jobs is too short and the jobs finish before our algorithm converges. Furthermore, the dynamic MARP values are also available at the paper web companion page.

Finally, we evaluate our approach with 4 time-varying workloads generated by SWIM. We use the same workloads as we presented in Sect. 3. The job size distribution varies across the different workloads: each job has only one reduce task and a varying number of map tasks chosen randomly from a given map size set. The actual configuration of the 4 workloads is given in Table 1. Each map task manipulates (reads or writes) one HDFS block; in our case 64MB. The complete input size of the workload is shown in the last column.

Figure 7 compares the per-job completion time distributions for static and dynamic MARP values. For each workloads, one can observe that, compared

Table 1. Configuration of SWIM workloads.

	#Jobs	#Maps	Map size set	Total input size
W1	500	10460	$\{5, 10, 40, 400\}$	335 GB
W2	500	25605	$\{5, 10, 50, 100, 300, 400\}$	819 GB
W3	1000	5331	$\{1, 2, \ldots, 35\}$	342 GB
W4	500	15651	$\{26, 27, \ldots, 50\}$	500 GB

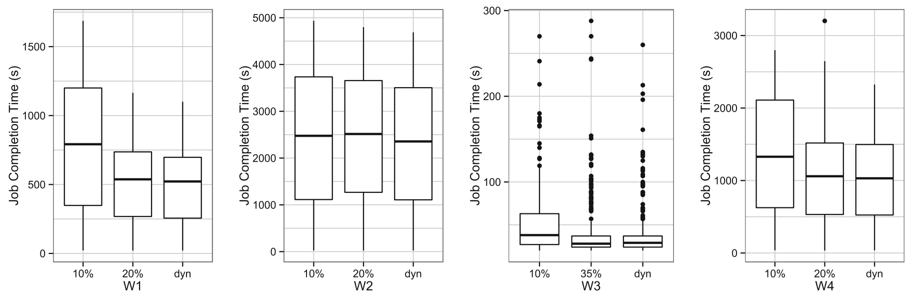


Fig. 7. The comparison of per-job completion time distribution observed for static and dynamic configuration parameters.

to the vanilla configuration, our approach can significantly reduce the job completion times (*e.g.* up to 40 % in W1). It also systematically delivers a better performance than the *best-effort* configurations.

The job-level accelerations can be accumulated and lead to the improvement of workloads-level performance. The overall completion times of the four SWIM workloads is further shown in Fig. 6. Similarly, our approach outperforms all the other configurations.

7 Related Work

Recently, the performance optimization for MapReduce systems has become a main concern in the domain of Big Data. This has resulted in a number of different approaches that aim to improve Hadoop performances.

Auto-Configuration in Hadoop. AROMA [14] is an automatic system that can allocate resources from a heterogeneous cloud and configure Hadoop parameters for the new nodes to achieve the service-quality goals while minimizing incurred cost. But, the VMs in the Cloud require to be provisioned and installed with the required Hadoop system *a priori.* Changlong *et al.* [15] also propose a self-configuration tool named AACT to maintain the performance of an Hadoop cluster. However, the adjustment of configurations for parallel requests are likely to conflict each others. The purpose of Starfish [9] is to enable Hadoop users and applications to get good performance automatically throughout the data lifecycle in analytics. Starfish measures the resource consumption of MapReduce jobs like CPU cycles and I/O throughput of HDFS to estimate average map execution time. However, the prediction may largely differ from the runtime situation. In concurrent case, due to its complex analytic steps, the over-head will also increase significantly. Gunther [16] is a search-based approach for Hadoop MapReduce optimization. It introduces an evolutionary genetic algorithm to identify parameter setting, resulting in near-optimal job performance. But, due to the complexity of the genetic algorithm, identifying an optimal configuration requires Gunther to repeat computing, thus causing performance to degrade.

Many other researches focusing on dynamic configuration like [19,20,23] also exist. Authors design self-adaptive models to optimize system performance, but their compatibility needs to be reconsidered for YARN.

Scalability at Runtime. Ghit *et al.* [5] have investigated a multi-allocation policies design, FAWKES, which can balance the distribution of hosts among several private clusters. In this case, FAWKES is focused on the dynamic redistribution of compute nodes between several clusters while the sum of compute nodes is fixed. However, due to the strict isolation between users, the clusters need to frequently grow or shrink to balance the scales, thereby penalizing each cluster. Chen *et al.* [2] propose a resource-time-cost model, which can display the relationship among execution time, input data, available system resource and the complexity of Reduce function for an ad-hoc MapReduce job. This model is a combination of the white-box [8] and machine-learning approaches. Its main purpose is to identify the relationship between the amount of resources and the job characteristics. Hadoop clusters can benefit from this research to optimize resource provisioning while minimizing the monetary cost. Finally, Berekmeri *et al.* [1] introduce a proportional-integral controller to dynamically enlist and discharge existing compute nodes from live Hadoop cluster in order to meet a given target service-level objectives.

Other Optimization Approaches. Some other studies look beyond Hadoop configuration optimization and scalability to library extensions and runtime improvements. FMEM [24] is a *Fine-grained Memory Estimator for MapReduce jobs* to help both users and the framework to analyze, predict and optimize memory usage. iShuffle [6] decouples shuffle-phase from reduce tasks and converts it into a platform service. It can proactively push map output data to nodes via a novel *shuffle-on-write* operation and flexibly schedule reduce tasks considering workload balance to reduce MapReduce job completion time. Seokyong *et al.* [10] propose an approach to eliminate fruitless data items as early as possible to save I/O throughput and network bandwidth, thus accelerating the MapReduce data processing. Benjamin *et al.* [7] deal with a geo-distributed MapReduce system by a two-pronged approach, which provide high-level insights and corresponding *cross-phase* optimization techniques, to minimize the impact of data geo-localization. Manimal [13] performs static analysis of Hadoop programs and deploys optimizations, including B-tree indexing, to avoid reads of unneeded data. Panacea [17] is a domain-specific compiler which performs source-to-source transformations for jobs to reduce the synchronization overhead of iterative jobs. Twister [4] introduces a new in-memory MapReduce library to improve the performance of iterative jobs. Some researches like [21,25] propose new MapReduce task scheduler to improve resource utilization while observing job completion time goals.

Since our contribution works on the YARN level, we believe that it complements these approaches.

8 Conclusion

Optimizing the performance of Hadoop clusters has become a key concern for big data processing. In YARN, inappropriate memory usage may lead to significant performance degradation. In this paper, we propose a self-adaptation approach based on a closed feedback control loop that automatically balances the memory utilization between YARN MapReduce processes. We have shown that it outperforms the default Hadoop configuration as well as the *best-effort* statically profiled ones. While in this paper we focus on MapReduce, our approach works on YARN level and therefore we plan to look for other applications based on YARN. For the further work, CPU management of YARN will be considered as a new part of this research. Furthermore, we look forward to explore the potential of this research on multi-queues basis, and also focus on HDFS I/O throughput to complement our approach with a support for I/O intensive jobs.

Acknowledgments. This work is partially supported by the Datalyse project www.datalyse.fr. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

1. Berekmeri, M., Serrano, D., Bouchenak, S., Marchand, N., Robu, B.: A control approach for performance of big data systems. In: IFAC World Congress (2014)
2. Chen, K., Powers, J., Guo, S., Tian, F.: CRESP: towards optimal resource provisioning for MapReduce computing in public clouds. IEEE Trans. Parallel Distrib. Syst. **25**, 1403–1412 (2014)
3. Chen, Y., Ganapathi, A., Griffith, R., Katz, R.H.: The case for evaluating MapReduce performance using workload suites. In: IEEE/ACM MASCOTS (2011)
4. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.H., Qiu, J., Fox, G.: Twister: a runtime for iterative MapReduce. In: HPDC (2010)
5. Ghit, B., Yigitbasi, N., Iosup, A., Epema, D.H.J.: Balanced resource allocations across multiple dynamic MapReduce clusters. In: ACM SIGMETRICS (2014)
6. Guo, Y., Rao, J., Zhou, X.: iShuffle: Improving hadoop performance with shuffle-on-write. In: Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013) (2013)
7. Heintz, B., Chandra, A., Sitaraman, R., Weissman, J.: End-to-end optimization for geo-distributed MapReduce. IEEE Trans. Cloud Comput. **PP**(99), 1–14 (2014)
8. Herodotou, H., Babu, S.: Profiling, what-if analysis, and cost-based optimization of MapReduce programs. PVLDB **4**(11), 1111–1122 (2011)
9. Herodotou, H., Lim, H., Luo, G., Borisov, N.: Starfish: a self-tuning system for big data analytics. In: Conference on Innovative Data Systems Research (2011)
10. Hong, S., Ravindra, P., Anyanwu, K.: Adaptive information passing for early state pruning in MapReduce data processing workflows. In: Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013) (2013)

11. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The HiBench benchmark suite: characterization of the MapReduce-based data analysis. In: Proceedings of the 26th International Conference on Data Engineering (ICDE)
12. IBM: An Architectural Blueprint for Autonomic Computing, 4 edition. Technical report, IBM (2006)
13. Jahani, E., Cafarella, M.J., Ré, C.: Automatic optimization for MapReduce programs. Proc. VLDB Endow. **4**, 385–396 (2011)
14. Lama, P., Zhou, X.: AROMA: automated resource allocation and configuration of mapreduce environment in the cloud. In: ICAC (2012)
15. Li, C., Zhuang, H., Lu, K., Sun, M., Zhou, J., Dai, D., Zhou, X.: An Adaptive auto-configuration tool for hadoop. In: ICECCS (2014)
16. Liao, G., Datta, K., Willke, T.L.: Gunther: search-based auto-tuning of MapReduce. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 406–419. Springer, Heidelberg (2013)
17. Liu, J., Ravi, N., Chakradhar, S., Kandemir, M.: Panacea: towards holistic optimization of MapReduce applications. In: CGO (2012)
18. Nzekwa, R., Rouvoy, R., Seinturier, L.: A flexible context stabilization approach for self-adaptive application. In: Proceedings of the 8th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom). IEEE (2010)
19. Padala, P., Hou, K., Shin, K.G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A.: Automated control of multiple virtualized resources. In: Proceedings of the 2009 EuroSys (2009)
20. Padala, P., Shin, K.G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., Salem, K.: Adaptive control of virtualized resources in utility computing environments. In: Proceedings of the 2007 EuroSys (2007)
21. Polo, J., Becerra, Y., Carrera, D., Torres, J., Ayguade, E., Steinder, M.: Adaptive MapReduce scheduling in shared environments. In:14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 61–70 (2014)
22. Ren, K., Gibson, G., Kwon, Y., Balazinska, M., Howe, B.: Hadoop's adolescence: a comparative workloads analysis from three research clusters. In: SC Companion on High Performance Computing, Networking Storage and Analysis (2012)
23. Wang, Y., Wang, X., Chen, M., Zhu, X.: Power-efficient response time guarantees for virtualized enterprise servers. In: Real-Time Systems Symposium (2008)
24. Xu, L., Liu, J., Wei, J.: FMEM: a fine-grained memory estimator for MapReduce jobs. In: Proceedings of the 10th International Conference on Autonomic Computing (2013)
25. Zhang, W., Rajasekaran, S., Wood, T., Zhu, M.: MIMP: deadline and interference aware scheduling of hadoop virtual machines. In: IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, May 2014