JellyFish: Online Performance Tuning with Adaptive Configuration and Elastic Container in Hadoop Yarn

Xiaoan Ding, Yi Liu, Depei Qian
Sino-German Joint Software Institute
Beihang University
Beijing, China
dingxiaoan@buaa.edu.cn, yi.liu@buaa.edu.cn

***Abstract*—MapReduce is a popular computing framework for large-scale data processing. Practical experience shows that inappropriate configurations can result in poor performance of MapReduce jobs, however, it is challenging to pick out a suitable configuration in a short time. Also, current central resource scheduler may cause low resource utilization, and degrade the performance of the cluster. This paper proposes an online performance tuning system, JellyFish, to improve performance of MapReduce jobs and increase resource utilization in Hadoop YARN. JellyFish continually collects real-time statistics to optimize configuration and resource allocation dynamically during execution of a job. During performance tuning process, JellyFish firstly tunes configuration parameters by reducing the dimensionality of search space with a divide-and-conquer approach and using a model-based hill climbing algorithm to improve tuning efficiency; secondly, JellyFish re-schedules resources in nodes by using a novel elastic container that can expand and shrink dynamically according to resource usage, and a resource re-scheduling strategy to make full use of cluster resources. Experimental results show that JellyFish can improve performance of MapReduce jobs by an average of 24% for jobs run for the first time, and by an average of 65% for jobs run multiple times compared to default YARN.**

Keywords—Distributed Computing; Performance Tuning; MapReduce; YARN

I. INTRODUCTION

MapReduce provides an effective solution to parallel program development and massive data processing on distributed platforms [1]. As an open source implementation, Hadoop is widely used in building MapReduce-based applications on large clusters. Despite the popularity and usability of Hadoop, application developers and system users face a series of challenges to achieve good performance in their applications. It often requires specialized system knowledge and tuning skills to obtain appropriate configuration.

Researchers have shown that Hadoop configuration plays an important role in performance of MapReduce programs [5]. Appropriate configuration settings could reduce execution time of jobs by using cluster resources efficiently and avoiding unnecessary disk I/Os. Moreover, some parameters decide if a job can be successfully executed and should be treated carefully. However, it is difficult to obtain an optimized configuration because: (1) there exists hundreds of parameters in the system; (2) parameters are related each other and act cooperatively; (3) configuration is application and hardware dependent, that is, optimized configuration is specific to characteristics and input dataset of an application for specified cluster [5, 7, 8].

Previous configuration tuning works can be categorized into three groups: following best practices and MapReduce tuning guides [12-16], offline configuration tuning [4, 8, 17, 18], and online configuration tuning [19, 20]. Online tuning systems search appropriate configuration by dynamically assigning test configurations to running tasks in the job. However, there are multiple drawbacks in current online approach. Firstly, the searching strategies for finding optimal configuration take little consideration to characteristics of MapReduce; Secondly, they neglect efficient resource utilization in the whole system; Thirdly, after a desirable configuration is achieved, the job uses the same configuration afterwards. However, the configuration might not be suitable for latter tasks because of data skew [21]. Inappropriate configuration can cause a task being killed due to out of memory error.

While tuning configuration parameters improves task performance, using cluster resources efficiently can also achieve significant performance improvement [6]. Researches have shown that average resource utilization in real-world data centers is fairly low [10, 11]. Generally reasons for low resource utilization includes: (1) tasks request more resources than they actually need; (2) resource usage varies during task execution but the amount of resources allocated to a container is fixed; (3) rest resources in a node is not enough for new containers and remain idle.

To address aforementioned problems, this paper proposes *JellyFish*, an online performance tuning system for Hadoop YARN. JellyFish requires no modification to MapReduce program source code.

Main contributions of this paper include:

- We propose a novel *elastic container* that can expand and shrink dynamically according to resource usage of the container. To schedule resources in each node, a rescheduling strategy is also proposed. To the best of our knowledge, this is the first work to use an elastic container in the cluster.
- In searching desirable configuration, it uses a divide-

1521-9097/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPADS.2015.112

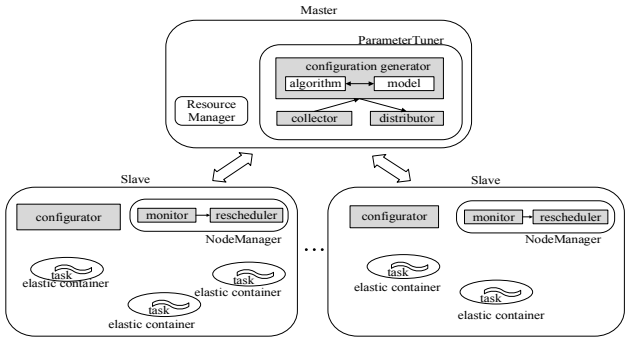


Fig. 1. The architecture of JellyFish.

and-conquer approach to reducing the dimensionality of searching space. A model-based hill climbing algorithm is also used to increase tuning efficiency.

- JellyFish firstly tunes configuration parameters to improve task performance, and secondly re-schedules idle resources to improve job performance and overall resource utilization in the cluster.

The rest of this paper is organized as follows. Section 2 gives the background and Section 3 describes the structure of our system. Section 4 and Section 5 present details of the system. Section 6 presents experimental results. Related works are discussed in Section 7 and section 8 concludes the paper.

II. SYSTEN OVERVIEW

JellyFish is a self-tuning system specialized in reducing execution time of *map/reduce* tasks, improving cluster resource utilization, thus enhancing the performance of MapReduce jobs. We implement JellyFish based on YARN, the next generation of Hadoop. Figure 1 shows the overall architecture of our system.

JellyFish improves performance from two perspectives: tuning configuration parameters and re-scheduling resources. The system is different from other performance tuning systems in two key designs: (1) The job-level configuration is changed from constant to dynamic on-the-fly task configuration. (2) The system uses novel *elastic container* and a resource re–scheduling strategy to employ idle resources on the node.

For configuration parameter tuning, JellyFish collects task statistics, generates and distributes configurations to newly started tasks with ParameterTuner. Correspondingly, each slave node has a monitor daemon collecting task metrics. The *generator* collects statistics of running tasks from the *collector*, searches suitable configuration values according to real-time statistics, and passes configurations to the *distributor*. The distributor maintains the best configuration at present and a list of test configurations.

For resource re-scheduling, each slave nodes has a *monitor* and a *resource rescheduler* that work together with *elastic containers*. The *monitor* traces all running containers continuously and delivers resource usage of individual container to *resource rescheduler* in real time.

Fig. 2 illustrates the tuning process in JellyFish. If a job has already reached to the desirable configuration from previous

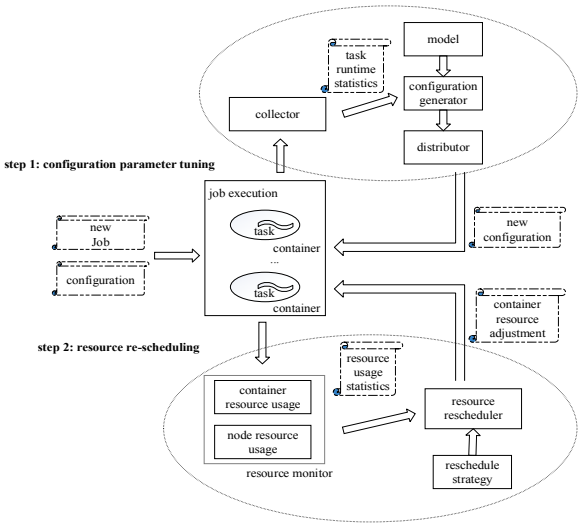


Fig. 2. The tuning process of JellyFish.

running tests, it will skip the first step. Otherwise, it starts with the default configuration in the first step.

During configuration parameter tuning process, each task runs with a specific configuration. The *collector* collects task runtime statistics and sends it to *configuration generator*. Newly generated configurations are placed in *distributor* and assign to new tasks in the next wave. The tuning process iterates until it satisfies the constraints in the algorithm.

Resource re-scheduling process starts only after a fixed configuration has been generated, since that dynamically changing resources interferes configuration tuning process. During resource re-scheduling, the *monitor* tracks runtime resource usage of each container and resource usage in the node. These statistics are used to decide whether to expand or shrink the capacity of each container based on current strategy.

III. CONFIGURATION PARAMETER TUNING

A. Selecting Tuning Parameters

Previous research shows that only a small set of parameters is critical to Hadoop performance [20]. So we follow the suggestion from previous studies and only selects the most important ones [8, 16]. The list of parameters we selected is shown in Table 1.

B. Searching Optimal Configurations

1) Dividing Parameters Search Space

Previous studies have shown that MapReduce parameters can be divided into two categories [8]. According to the impact on performance of MapReduce job, we divide these parameters into two groups and search separately: map-relevant parameters and reduce-relevant parameters. Low dimensionality of search space reduces search time dramatically.

1) Searching Algorithm

Searching optimal configuration in a high-dimensional search space is a black-box optimization problem. The current

TABLE I. CONFIGURATION PARAMETER IN JELLYFISH

parameter	default value	symbol
map phase tuning parameter		
mapreduce.task.io.sort.mb	100	p_{mapBuf}
mapreduce.map.sort.spill.percent	0.80	$p_{mapBufSpill}$
mapreduce.map.java.opts	200	$p_{mapJavaOpts}$
reduce phase tuning parameter		
mapreduce.reduce.input.buffer.percent	0.0	$p_{rudInPerc}$
mapreduce.reduce.shuffle.input.buffer. percent	0.70	$p_{shufInBufPerc}$
mapreduce.reduce.shuffle.merge.percent	0.66	$p_{shufMergePerc}$
mapreduce.reduce.shuffle.memory.limit. percent	0.25	$p_{perShufPerc}$
mapreduce.reduce.merge.inmem.threshold	1000	$p_{inmemMergeThre}$
mapreduce.reduce.shuffle.parallelcopies	5	$p_{paraCopy}$
mapreduce.reduce.java.opts	200	$p_{redJavaOpts}$
both map and reduce phase tuning parameter		
mapreduce.task.io.sort.factor	10	$p_{sortFactor}$

approach is to use heuristic search algorithms to find a near-optimal or best-possible solution, such as random genetic, simulated annealing, hill climbing, recursive search and Tabu search. Previous works have tried the first three algorithms in finding optimal configuration [4, 19, 20]. However, none of them combine MapReduce characteristics with search algorithm. To address these defects, we propose model-based hill climbing algorithm which has following characteristics:

- The model of configuration parameters are constructed based on YARN source code, and interactions among different function modules and the reciprocity of parameters values are taken into account.
- Metrics are collected to update the value ranges and value restrictions of parameters, which can increase convergence speed and search quality of the algorithm.
- The search algorithm is inspired by smart hill climbing algorithm [23]. We leverage Latin Hypercube Sampling (LHS) to avoid trapping in a local optimum.

Our Model-Based Hill Climbing Algorithm is shown in Algorithm 1, and is composed of two phases: global search phase and local search phase. The global search aims at covering the search space as broader as possible. The local search phase starts from current best configuration *C_curBest* and seeks the steepest direction by exploring *C_curBest*'s neighborhood.

2) Parameter Constraints

Our parameter model is similar to [9], but we put emphasis on how the selected parameters impact on the cost of a task. During the execution of a job, the system continually collect the size and record numbers of map output as *MapOutputByte* and *MapOutputRec* in each task and keep the greatest one in *MaxMapOutputByte* and *MaxMapOutputRec*. In LHS sampling in global search, we use the updated values to narrowing the scope of parameters with (1) (2) (3), where *mapNum* denotes the number of map tasks and *reduceNum* denotes the number of reduce tasks.

$$p_{sortFactor} > \left\lceil \frac{MaxMapOutputByte + MaxMapOutputRec * 16}{p_{mapBuf} * 2^{20} * p_{mapBufSpill}} \right\rceil \quad (1)$$

Algorithm 1 Model-based hill Climbing

```
1: Initialize sampling parameter m,global_threshold and local_threshold
2: global_search_time = 0, local_search_time =0,
   cost(C_curBest) = MAX_VALUE
3: while global_search_time < global_threshold do
4:         global_search_time++, local_search_time = 0
5:         samples[m] = conditional_LHS(m)
6:         ConfigList = satisfyConstraint(samples[m])
7:             if configList is empty
8:                 continue
9:             end if
10:        ConfigResult = assignToTasks(ConfigList)
11:        updateConstraintsAndRanges(ConfigResult)
12:        if cost(best(ConfigResult)) < cost(C_curBest)
13:            C_curBest = best(ConfigResult)
14:        else
15:            continue
16:        end if
17:        while local_search_time < local_threshold do
18:            local_search_time++
19:            NeighbourList = getNeighbours(C_curBest)
20:            ConfigList = satisfyConstraint(NeighbourList)
21:            if configList is empty
22:                break
23:            end if
24:            ConfigResult = assignToTasks(ConfigList)
25:            updateConstraintAndRanges(ConfigResult)
26:            if cost(best(ConfigResult)) < cost(C_curBest)
27:                C_curBest = best(ConfigResult)
28:            else
29:                continue
30:            end if
31:        end while
32: end while
```

$$p_{mapBuf} > \frac{MaxMapOutputByte + MaxMapOutputRec * 16}{2^{20}} \quad (2)$$

$$p_{redJavaOpts} > \frac{MaxMapOutputByte * mapNum}{reduceNum} \quad (3)$$

The constraints of parameters used in our algorithm is (4) (5) (6). Only the configurations satisfy these constraints will be assigned to tasks.

$$p_{redJavaOpts} * p_{shufInBufPerc} * p_{perShufPerc} > \frac{MaxMapOutputByte}{reduceNum} \quad (4)$$

$$p_{redJavaOpts} * p_{shufInBufPerc} * p_{shufMergePerc} > \frac{MaxMapOutputByte}{reduceNum} * mapNum \quad (5)$$

$$p_{mapBuf} * 2^{20} * p_{mapBufSpill} > MaxMapOutputByte * MaxMapOutputRec * 16 \quad (6)$$

3) Configuration Evaluation

Evaluation functions are defined to assess configurations, one is for map-relevant configuration, and the other is for reduce-relevant configuration. As our goal is to minimize execution time and reduce the number of spill records, these two factors are considered. Also, we add the ratio of spill records to eliminate impacts of data skew. The ratio of Java heap size is also included to avoid over-allocation of buffer. As shown in equation (7) (8), each item is given a weight value.

$$\text{mapTaskCost} = \frac{spillRecNum}{mapOutputRecNum} * w_{mapSpill} + \frac{1}{T_{mapTask}} * w_{mapTime} + \frac{p_{mapJavaOptis}}{mapMemory} * w_{mapHeap} \quad (7)$$

$$\text{reduceTaskCost} = \frac{spillRecNum}{reduceInputRecNum} * w_{reduceSpill} + \frac{1}{T_{reduceTask}} * w_{reduceTime} + \frac{p_{reduceJavaOptis}}{reduceMemory} * w_{reduceHeap} \quad (8)$$

C. Online-Tuning Approach

We deploy an online tuning approach in searching optimal configuration. Our work relies on the theory that all the map/reduce tasks have the same execution logic and similar input data in a job. These tasks run in multiple waves. We assign different configurations to the tasks and collect their running statistics. Then we evaluate these configurations with their statistics and get a suitable one for the job.

IV. Resource Re-scheduling

A. Elastic Container

Elastic container is proposed to support resource re-scheduling in our system. Unlike containers in YARN, the size of an elastic container can temporarily expand by taking idle resources from other containers or the node, and shrink by handing resources back to the node.

B. Rescheduling Strategies

Hadoop limits resource usage in each task to avoid system overload. While the amount of memory resources in a container is crucial, CPU resource is flexible because shortage of CPU resource would not incur task failure. Since our goal is improving the parallelism of a job and making full use of resources in the node, we arrange containers as much as possible in each node.

Firstly, we reset the amount of resources allocated to each container. In previous tuning step, we get the memory usage of a map/reduce task with the optimized configuration. We adjust memory allocation per container according to the maximum memory usage during task execution. For CPU resource in each container, it remains unchanged. Thus, from central resource scheduler perspective, a task requests fewer memory and more tasks are running in the system at the same time.

In resource re-scheduling, we assign idle resources to running tasks in the node. Idle resources refer to the portion of resources not being used at the moment. Using these resources would not incur severe resource contention. When *resource rescheduler* detects that a container needs more resources to maintain task execution or improve task performance, it assigns idle resources to this container temporarily. When borrowed resources have to return to the system, the container gives these resources back.

Equation (9) shows how we evaluate which container has the biggest chance to obtain extra resources. A positive value of *needs* means that a container is in a resource-critical situation and needs more resources, while a negative value means that a container has idle resources to be re-scheduled. A positive value of *unfairness* suggests that a container has lent resources from others, while a negative value suggests it has borrowed some resources to others.

$$scaling = needs + unfairness \tag{9}$$

Algorithm 2 shows our procedure in *resource rescheduler*. *ContainerList* is used to record all the containers in the node. The idle resource pool is designed to preserve idle resources in the node. We try to minimize the resources in the pool to improve the resource utilization of the node.

Algorithm 2 Resource Re-schedule

```
1: while (true)
2:     get all finished containers and remove from ContainerList
3:     give back resources to the idle resource pool
4:     get new start containers and add to ContainerList
5:     if (enough resource in idle resource pool)
6:         allocate resources to new containers
7:     else
8:         take all the resources from idle resource pool
9:         retrieve resource back from the containers with lowest
           scaling value
10:    end if
11:    monitor all the containers in ContainerList
12:    put idle resources to idle resource pool
13:    assign idle resources to containers
14: end while
```

CPU and memory resources are considered separately. While CPU resources are flexible, memory resources are fatal in task execution. Re-executing a task from beginning is time-consuming, and is more likely to fail again due to insufficient memory in the container. Our algorithm can avoid this out of memory error to certain extent. When we detect memory is overused in a container, our algorithm add more memory resources to it. Considering CPU resources, when we detect full CPU utilization in a container, our algorithm add idle CPU resources to it. If adding CPU resources has no performance improvement and CPU usage of the container is at a low level, it retrieves the CPU resource back to the idle resource pool.

We re-schedule resource in the node with non-interference to the central resource scheduler. From ResourceManager perspective, idle resources assigned to running containers are still available in the cluster.

V. Evaluation

A. Experiment Setup

We evaluate JellyFish on a 4-nodes cluster, with each node equipped with dual Intel six-core Xeon E5-2420 processors running at 2.2Ghz, 15 MB L3 cache, 7GB memory, one 320GB SAS disk, and a gigabit Ethernet card. We use seven benchmark applications from PUMA benchmark [2] as shown in Table 3.

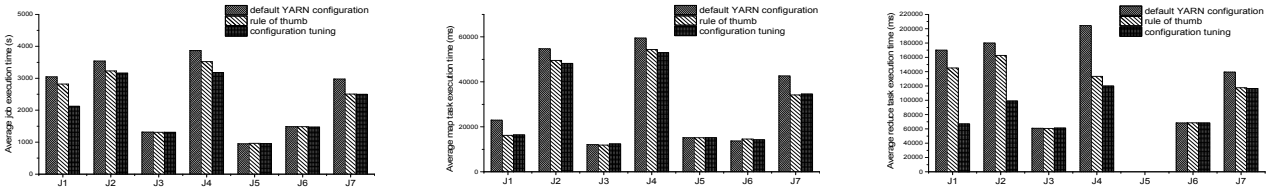
B. Effectiveness of Configuration Tuning Process

We firstly evaluate the effectiveness of configuration tuning process. Fig. 3(a) compares various applications completion time using default configuration, rule of thumbs [13], and configuration tuning process. Table 2 shows the performance improvement compared with default YARN configuration.

Fig. 3(b), (c) show average execution time of reduce tasks has more improvement than map tasks. The main reason is that we change the value of *input.buffer.percent*, which decides the maximum records a reduce task can retain in the buffer. The default value of this parameter is 0. Fig. 4 shows the reduction in spill records, which accounts for the reduction in execution time. Grep, Classification, and HistogramMovies has relatively fewer map output and shuffle records, thus fewer spill records. So our configuration tuning process has less effects on them.

TABLE II. CHARACTERISTIC OF BENCHMARK APPLICATIONS

Benchmark	Input Data	Input Size	# Maps	#Reduces	Label
TeraSort	TeraGen	50 GB	374	99	J1
WordCount	Wikipedia	50 GB	396	99	J2
Grep	Wikipedia	50 GB	396	99	J3
Inverted Index	Wikipedia	50 GB	396	99	J4
Classification	Movie ratings dataset	50 GB	422	0	J5
HistogramMovies	Movie ratings dataset	50 GB	422	99	J6
HistogramRatings	Movie ratings dataset	50 GB	422	99	J7

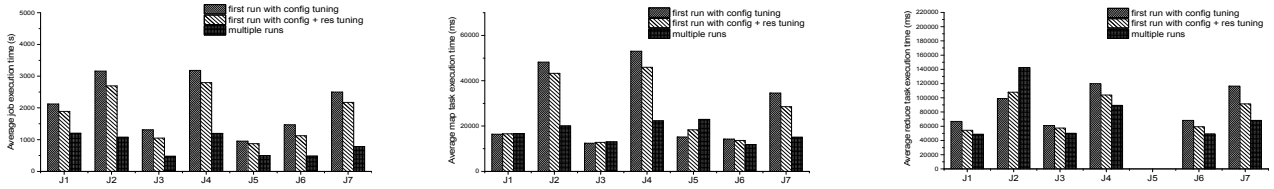


(a) Job execution time (b) Map task execution time (c) Reduce task execution time

Fig. 3. Effectiveness of Configuration Tuning Process

(a) Spill records of J1, J2, J4 (b) Spill records of J3, J5, J6, J7

Fig. 4. Spill records of a job with default YARN configuration, rule of thumbs, and configuration tuning process in JellyFish.



(a) Job execution time (b) Map task execution time (c) Reduce task execution time

Fig. 5. Effectiveness of Resource Re-scheduling Process.

(a) CPU utilization in the cluster. (b) Memory utilization in the cluster

Fig. 6. Improvement of resource utilization in the cluster

C. *Effectiveness of Resource Re-scheduling Process*

We also evaluate the effectiveness of our resource rescheduling process. We compare job executing time in three cases. The first case is that a job starts with the default configuration and runs with configuration tuning process. The second is that a job starts with default configuration and runs with both two tuning processes. The third is that a job begins with a suitable configuration obtained in previous job execution and runs with resource re-scheduling process.

Fig. 5 (a) compares various completion time in different cases and Table 2 shows the performance improvement by comparing execution time with YARN default configuration. Fig. 5 (b), (c) illustrate that JellyFish shortens task execution time with resource re-scheduling for most jobs. However, tasks in WordCount and Classification have longer execution time because both of them are CPU-intensive applications. They are in CPU-contention when adding more containers. By improving the parallelism in the cluster, they still have shorter job execution time. Fig. 6(a) illustrates that CPU utilization has been tripled. Figure 6(b) reveals that memory resource utilization also increases. TeraSort and HistogramRatings have

TABLE III. REDUCTION IN EXECUTION TIME OF JELLYFISH COMPARED WITH DEFAULT YARN CONFIGURATION

Application	first run with config tuning	first run with config + res tuning	multiple runs
J1	30%	38%	61%
J2	11%	24%	70%
J3	1%	20%	64%
J4	18%	28%	69%
J5	0%	8%	48%
J6	1%	24%	68%
J7	16%	27%	74%

less improvement because their memory usage is almost the same with resource allocation in default.

D. Impact of Job Size on the effective of JellyFish

To study the effectiveness of JellFish with different job size, we run Terasort with increasing input data sets ranging from 2GB to 100GB. Fig. 7 shows the result, where we can observe that JellyFish can reduce job execution time significantly in jobs run more than once. Jobs run for the first time in JellyFish have relatively longer execution time because it spends most of time in searching for the desirable configuration.

VI. RELATED WORK

MapReduce Configuration Parameters Tuning: MROnline [19] is an online performance system that enables task-level configuration tuning. Ant [20] is a self-adaptive task-level tuning system, it automatically finds the optimal settings for individual jobs running in a heterogeneous cluster. Wu et al. [4] train their system PPABS to form a set of equivalent classes of MapReduce application with CPU and memory usage and find appropriate configuration for each class. AROMA [17] collects application resource utilization and classifies them into different groups. Gunther [18] uses a genetic algorithm and take 20-40 runs to obtain a good parameter configuration. Herodotou et al. [5, 7-8] profiles concise statistics in the first run and use a what-if engine to calculate an optimal configuration.

Improve Resource Utilization In Hadoop: Guo et al. [6] propose a resource stealing strategy to enable running tasks to take resources reserved for idle slots and return them back. Polo et al. [24] present a resource-aware scheduling technique for MapReduce multi-job workloads. DynamicMR [25] use Dynamic Hadoop Slot Allocation by keeping the slot-based model.

VII. CONCLUSION

This paper proposes an online peformance tuning system, JellyFish, to improve performance of MapReduce jobs and increase resource utilization in the system. In our future work, we plan to extend JellyFish to support multiple resource re-scheduling strategy and multi-tenant Hadoop environment.

ACKNOWLEDGEMENT

This research was supported by the Natural Science Foundation of China under Grant No. 61133004, National Hi-tech R&D program of China (863 program) under grant No. 2012AA01A302.

REFERENCES

[1] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.

[2] Ahmad, Faraz, et al. "Puma: Purdue mapreduce benchmarks suite." (2012).

[3] A.S. Foundation. Apache Hadoop 2.6.0, 2014. http://hadoop.apache.org/docs/r2.6.0/.

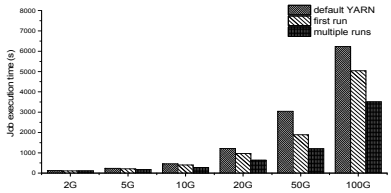


Fig. 7. Effectiveness of JellyFish affected by job size

[4] Wu, Dili, and Aniruddha Gokhale. "A self-tuning system based on application Profiling and Performance Analysis for optimizing Hadoop MapReduce cluster configuration." High Performance Computing (HiPC), 2013 20th International Conference on. IEEE, 2013.

[5] Babu, Shivnath. "Towards automatic optimization of MapReduce programs." Proceedings of the 1st ACM symposium on Cloud computing. ACM, 2010.

[6] Guo, Zhenhua, et al. "Improving resource utilization in mapreduce." Cluster Computing (CLUSTER), 2012 IEEE International Conference on. IEEE, 2012.

[7] Herodotou, Herodotos, Fei Dong, and Shivnath Babu. "No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics." Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, 2011.

[8] Herodotou, Herodotos, et al. "Starfish: A Self-tuning System for Big Data Analytics." CIDR. Vol. 11. 2011.

[9] Herodotou, Herodotos. "Hadoop performance models." arXiv preprint arXiv:1106.0940 (2011).

[10] Kavulya, Soila, et al. "An analysis of traces from a production mapreduce cluster." Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on. IEEE, 2010.

[11] Reiss, Charles, et al. "Towards understanding heterogeneous clouds at scale: Google trace analysis." Intel Science and Technology Center for Cloud Computing, Tech. Rep (2012): 84.

[12] Jiang, Dawei, et al. "The performance of mapreduce: An in-depth study." Proceedings of the VLDB Endowment 3.1-2 (2010): 472-483.

[13] Clodera. Optimizing MapReduce job performance,2012. http://www.slideshare.net/cloudera/mr-perf.

[14] Impetus. Advanced Hadoop Tuning and Optimization - Hadoop Consulting, 2009. http://www.slideshare.net/ImpetusInfo/ppt-on-advanced-hadoop-tuning-n-optimisation

[15] Impetus. Hadoop Performance Tuning, 2012. https://hadoop-toolkit.googlecode.com/files/White%20paperHadoopPerformanceTuning.pdf.

[16] White, Tom. Hadoop: The definitive guide. " O'Reilly Media, Inc.", 2012.

[17] Lama, Palden, and Xiaobo Zhou. "Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud." Proceedings of the 9th international conference on Autonomic computing. ACM, 2012.

[18] Liao, Guangdeng, Kushal Datta, and Theodore L. Willke. "Gunther: Search-based auto-tuning of mapreduce." Euro-Par 2013 Parallel Processing. Springer Berlin Heidelberg, 2013. 406-419.

[19] Li, Min, et al. "MRONLINE: MapReduce online performance tuning." Proceedings of the 23rd international symposium on High-performance parallel and distributed computing. ACM, 2014.

[20] Cheng, Dazhao, et al. "Improving MapReduce performance in heterogeneous environments with adaptive task tuning." Proceedings of the 15th International Middleware Conference. ACM, 2014.

[21] Jiang, Dawei, et al. "The performance of mapreduce: An in-depth study." Proceedings of the VLDB Endowment 3.1-2 (2010): 472-483.

[22] Vavilapalli, Vinod Kumar, et al. "Apache hadoop yarn: Yet another resource negotiator." Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013.

[23] Xi, Bowei, et al. "A smart hill-climbing algorithm for application server configuration." Proceedings of the 13th international conference on World Wide Web. ACM, 2004.

[24] Polo, Jose, et al. "Deadline-based MapReduce workload management." Network and Service Management, IEEE Transactions on 10.2 (2013): 231-244.

[25] ang, Shanjiang, Bu-Sung Lee, and Bingsheng He. "DynamicMR: A Dynamic Slot Allocation Optimization Framework for MapReduce Clusters." Cloud Computing, IEEE Transactions on 2.3 (2014): 333-34