mrOnline: MapReduce Online Performance Tuning

Min Li†, Liangzhao Zeng‡, Shicong Meng‡,
Jian Tan‡, Li Zhang‡, Ali R. Butt†, Nicholas Fuller‡
† Dept. of Computer Science, Virginia Tech; ‡IBM TJ Watson Research Center
{limin,butta}@cs.vt.edu, {lzeng,smeng,tanji,zhangli,nfuller}@us.ibm.com

ABSTRACT

MapReduce job parameter tuning is a daunting and time consuming task. The parameter configuration space is huge; there are more than 70 parameters that impact job performance. It is also difficult for users to determine suitable values for the parameters without first having a good understanding of the MapReduce application characteristics. Thus, it is a challenge to systematically explore the parameter space and select a near-optimal configuration. Extant offline tuning approaches are slow and inefficient as they entail multiple test runs and significant human effort.

To this end, we propose an online performance tuning system, mrOnline, that monitors a job's execution, tunes associated performance-tuning parameters based on collected statistics, and provides fine-grained control over parameter configuration. mrOnline allows each task to have a different configuration, instead of having to use the same configuration for all tasks. Moreover, we design a gray-box based smart hill climbing algorithm that can efficiently converge to a near-optimal configuration with high probability. To improve the search quality and increase convergence speed, we also incorporate a set of MapReduce-specific tuning rules in mrOnline. Our results using a real implementation on a representative 19-node cluster show that dynamic performance tuning can effectively improve MapReduce application performance by up to 30% compared to the default configuration used in YARN.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems —*distributed applications*; D.2.9 [**Software**]: Software Engineering —*software configuration management*

General Terms

Design, Algorithms, Performance, Evaluation.

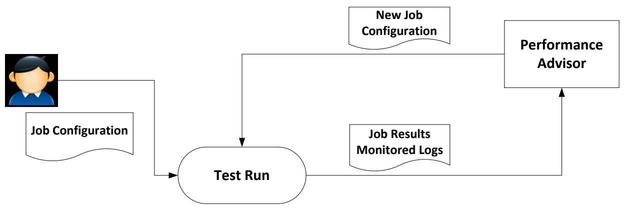

Figure 1: Current offline performance tuning approach used for MapReduce applications.

Keywords

Cloud computing; MapReduce; YARN; Online parameter tuning; Performance tuning; Dynamic parameter configuration; Hill climbing.

1. INTRODUCTION

The use of the MapReduce [3] large-scale data processing framework is growing exponentially as the amount of data created by web search engines, social networks, business web click streams, and academic and scientific research continue to increase at unprecedented rates. While the MapReduce framework enables users to scale up applications easily, writing good MapReduce applications requires specialized system-level skills and extra effort as users also have to provide application-specific job and system configuration parameters. These parameters are crucial and affect performance significantly. For example, consider the configuration parameter $io.sort.mb$ that controls the amount of buffer memory to use when sorting files. Setting $io.sort.mb$ to sub-optimal values can lead to unnecessary I/Os and consequently increase task running time. Moreover, different applications require different values for $io.sort.mb$ depending on the HDFS [8, 11] block size and the map task output size. Similarly, applications vary in demands, e.g., the MapReduce application Grep [9] requires smaller sort space than Terasort [10], as Grep usually outputs much less data than Terasort in the map phase. Recent research such as Starfish [15] shows that MapReduce application performance depends on the size and content of the data set, job characteristics, cluster hardware configuration, and more importantly configuration parameters. The importance of such performance tuning is further highlighted by the observation that a simple web search on the topic yields a long list of best practices and MapReduce tuning guides [1, 2, 16, 17, 31]. These documents show that multiple orders of application performance gains can be achieved when using tuned parameters compared to the default settings.

The challenge is that the MapReduce job parameter tuning is a daunting and time consuming task. This is mainly due to the fact that the parameter configuration space is huge; the number of performance related parameters in Hadoop [8] is more than 70. Moreover, it is difficult for a user to determine the optimal value for a parameter without first having a deep understanding of the application characteristics. The current approach to address this challenge is to use offline performance tuning. As shown in Figure 1, traditional offline tuning first selects a configuration based on default settings or a rough understanding of application characteristics. Next, several test runs of the application are done and profiled to collect data such as job performance counters, and system monitoring logs. The user then feeds the collected data to a performance advisor, or perform manual static analysis, to generate a new configuration. The whole process is then repeated multiple times until a desired performance goal is reached. The selected configuration is then employed for running the application on production clusters.

There are multiple drawbacks of the above traditional performance tuning for MapReduce. First, the process is time consuming as it requires many test runs and each run can only try a single configuration. This is further exacerbated when the application involves long running tasks. Second, the offline approach is not cost effective if the tuning is done for an application that will only run for few times or perhaps just once. Users would rather simply run their applications without tuning, leading to overall inefficient resource utilization. In addition, as shown by Starfish [15], the optimal configuration also depends on the data set and cluster hardware configuration. This implies that users would have to re-adjust the parameters each time they change the input data sets or run the applications on different clusters. Moreover, no one configuration is suitable for all tasks within a job. MapReduce jobs also commonly exhibit data skew [23] that requires different amounts of resources based on the different sizes of data being processed. Finally, the traditional offline tuning statically tunes the parameters once and use the configuration throughout the whole life cycle of a job. However, the job characteristics and cluster utilization are dynamic, and static tuning cannot adapt to such variations and thus cannot avoid performance-degrading cluster hot spots.

In this paper, we mitigate the aforementioned problems by designing an online performance tuning system, MRONLINE. MRONLINE improves single Hadoop job performance via online performance tuning, as well as expedites the performance tuning process by reducing the number of test runs by employing a finer grain online process that tests multiple configurations per run. MRONLINE provides the ability to tune multiple jobs' performance in a multi-tenant environment. Moreover, MRONLINE considers dynamic cluster utilization information to help MapReduce applications avoid hot spots. MRONLINE also does not require any modifications to user programs, which makes it user friendly and encourages quick adoption.

Specifically, this paper makes the following contributions:

- We design and implement a task-level dynamic configuration framework based on YARN [29], the second generation open source MapReduce implementation. MRONLINE enables different configurations for each map and reduce task, which is a key system-level improvement over extant approaches and offers huge optimization opportunities.

- We design a gray-box based smart hill climbing algorithm to systematically search the MapReduce parameter space, which relies on our task-level dynamic configuration framework. We support both aggressive and conservative tuning strategies.

- We propose tuning rules for key parameters, which improve search quality and reduce convergence iterations.

- We evaluate MRONLINE on YARN and present an experimental performance evaluation on a representative 19-node cluster. Our results demonstrate that compared to the default YARN settings, our approach achieves an efficiency improvement of 22% by dynamically tuning the applications. Moreover, for applications that run multiple times, MRONLINE can expedite the test runs and reduce job execution time by up to 30%. Our results show that MRONLINE offers an effective means to improve MapReduce application performance.

The rest of the paper is organized as follows. Section 2 presents an introduction of YARN, the classification of configuration parameters and identifies two use cases that we consider in MRONLINE. Section 3 discusses the system architecture of MRONLINE, followed by an explanation of the task-level dynamic configuration in Section 4. In Section 5, we detail the design of our gray-box based hill climbing algorithm to systematically search for optimal configuration parameters. Section 6 describes the tuning rules for various key job configuration parameters. Section 7 discusses our implementation details. Section 8 demonstrates the effectiveness of MRONLINE versus the default configuration through a series of experiments. Related works are discussed in Section 9, and finally Section 10 concludes the paper.

2. BACKGROUND

In the section, we first describe YARN that serves as an enabler for MRONLINE. Next, we present a classification of the considered configuration parameters, followed by an identification of two specific use cases that we have considered in MRONLINE.

2.1 YARN

MRONLINE is designed and implemented on YARN [29], the latest generation of the publicly available Hadoop platform. We choose YARN as it provides many advantages over prior versions of Hadoop. Hadoop is designed as a monolithic framework, which tightly couples the MapReduce programming model with distributed resource management. This leads to unnecessary/forced use and misuse of the MapReduce programming model when all what users want is to just leverage the large-scale compute resources provided by enterprises and research organizations. For instance, users have been observed to submit map-only applications to simply launch arbitrary processes (not necessarily MapReduce tasks) in Hadoop clusters [29], or submit applications that have map functions implemented as reduce tasks to circumvent limited map quotas [12]. Moreover, Hadoop employs a centralized scheduler for managing tasks of all jobs. This is becoming a performance bottleneck as the number of jobs submitted to a Hadoop cluster grows. Traditional Hadoop implementation also does not support changing the map or reduce slot configurations between different jobs, which precludes dynamically adapting to variations during a job's life-cycle, and consequently reduces system efficiency.

YARN has been designed to address the above limitations. It separates the computational programming models from resource management, and provides support for frameworks other than MapReduce such as Giraph [7, 27], Spark [19, 35], and Storm [28]. In this paper, we focus on tuning parameters of the MapReduce programming model running on top of YARN. However, YARN can be exploited to extend our approach to support performance tuning of other frameworks as well. Another useful feature of YARN is that it delegates application related scheduling to per *application masters* that can employ application-specific resource scheduling, thus providing for higher scalability. For this purpose, YARN manages cluster-wide resources through the use of a new key concept, "container." A container is a resource scheduling unit that encapsulates the number of CPUs, required memory, interconnect bandwidth, etc. for scheduling. Different MapReduce applications can request different-sized containers from YARN as per their needs. For example, an application master is responsible for specifying the number of needed containers, the size of each container, and the mapping between the containers and tasks. MRONLINE leverages the containers to design a task-level configuration framework.

2.2 Parameter Classification

We focus on parameters that impact application performance and are amenable to dynamic configuration. The optimal values of these parameters depend on the application characteristics, the size and the contents of the associated input data and the cluster setup. We classify the considered parameters into three categories based on when the modified value of a parameter can take effect.

The first category includes parameters that are difficult to change after the application has started. The number of mappers, the number of reducers, and slow start ($mapreduce.job.reduce.slowstart.completedmaps$) are three key parameters that fall into this category. Slow start specifies the number of mappers that should be completed before any reducers are launched. Starting the reduce tasks early can help overlap the map tasks execution with the shuffle phase and improve performance. However, starting the reduce tasks too early creates contention for the cluster resources that are also needed by the mappers. The optimal value for the parameter is application specific.

The second category consists of parameters that cannot be changed on the fly for already running tasks, but impact the tasks that will be launched after changes have been made. Examples of such parameters include $io.sort.mb$, the number of virtual cores in a container, the size of memory in a container, and parameters specifying reduce buffer size. Choosing the right values for this category can reduce I/Os and improve the cluster utilization.

The third category consists of parameters that can be changed on the fly and become effective immediately. Parameters such as $mapred.inmemmerge.threshold$ and $io.sort.spill.percent$ fall into this category. These two parameters control the threshold of when to spill out data from memory onto disks. MRONLINE can even try multiple values within a task for parameters in this category, thus speeding up the tuning process.

MRONLINE currently supports tuning of parameters in the second and third categories. Tuning of parameters in the first category can be done using simulation tools, such as MRPerf [30], and remains a focus of our on-going research.

2.3 Use Cases for MRONLINE

We considered two use cases for designing MRONLINE. The first use case is to expedite test runs by trying multiple task configurations in a single test run. This enables us to reduce the tuning time significantly. The second use case is to improve the performance of applications that are executed only once. MRONLINE employs different strategies for each of these two use cases.

1. Expedited Test Runs Use Case: In this use case, we aggressively and systematically search for different parameter configurations to find optimal values. We first design and implement a task-level configuration framework that enables testing of different parameter configurations in a single test run. We then design a gray-box based smart hill climbing algorithm to find the optimal configuration. The quality of the generated solution depends on the number of tasks executed in a single test run. If too few tasks are executed, the configuration quality can be improved by multiple test runs. We also increase the algorithm convergence speed by monitoring and modeling the runtime statistics into the algorithm.

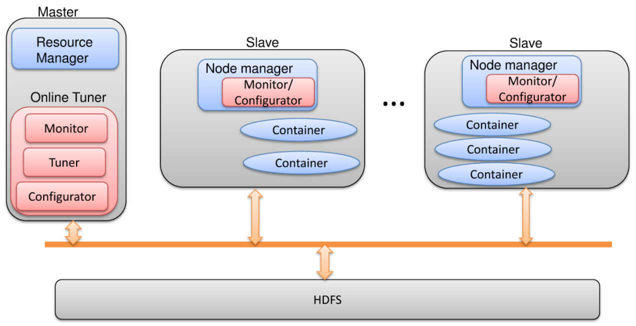

Figure 2: MRONLINE system architecture.

2. Fast Single Run Use Case: In this use case, we improve the application performance in a single run. Here, we conservatively tune the configurations mostly based on the observed runtime statistics. For example, if we observe that too many spills are being written to the disk, we increase the size of the sort buffer. Our main goal in this case is to improve performance of the current task, instead of searching for a desirable configuration that can be used for later runs. This is particularly useful for jobs that run few times or only once. For this use case, a performance boost can be achieved by simply co-executing MRONLINE with target applications.

3. SYSTEM ARCHITECTURE

The overall architecture of MRONLINE is shown in Figure 2. MRONLINE is based on YARN that, unlike the centralized job tracker of earlier versions of Hadoop, has a resource manager that manages cluster resources and the execution cycle of distributed application-specific masters, and tracks node liveness. To support dynamic configuration, MRONLINE modifies the YARN resource manager to support allocation of different-sized containers for different applications. The Node manager, akin to the task tracker of Hadoop, runs on each cluster node and is responsible for managing the containers running locally on the node. However, YARN delegates the task tracking functions to per application components. MRONLINE implements its sub-components within each node manager to leverage existing features such as resource monitoring.

MRONLINE consists of a centralized master component, *online tuner*, which is a daemon that can run on the same machine as the resource manager of YARN or on a dedicated machine. Online tuner controls a number of distributed slave components that run within the node managers on the slave nodes of the YARN cluster.

Online tuner is composed of three components: a *monitor*, a *tuner* and a *dynamic configurator*. The monitor works together with the per-node slave monitors to periodically monitor application statistics. Specifically, the slave monitors gather statistics about the tasks running on the node, as well as the node statistics, and send the information to the centralized monitor. The centralized monitor then aggregates, analyzes and passes the information to the tuner.

The tuner implements the tuning strategies and algorithms, which decide what parameters should be changed and what new values should be assigned. When needed, the tuner generates new configurations for each application and task. Finally, the dy-

API	Description
`List<String> getConfigurableJobParameters(JobID jid)`	Returns the set of configurable parameters for the job with job ID `jid` and associated tasks that are currently running or will run in the future.
`List<String> getConfigurableTaskParameters(JobID jid, TaskID tid)`	Returns the set of configurable parameters for the tasks with job and task IDs `jid` and `tid`, respectively.
`int setJobParameters(JobID jid, Map<String, String> kv)`	Sets the parameters for a job with ID `jid`.
`int setTaskParameters(JobID jid, TaskID tid, Map<String, String> kv)`	Sets the parameters for a task with job and task IDs `jid` and `tid`, respectively.
`int setTaskParameters(JobID jid, Map<String, String> kv)`	Sets the parameters for all the tasks associated with a job with ID `jid`.

Table 1: Key APIs provided by the dynamic configurator of MRONLINE.

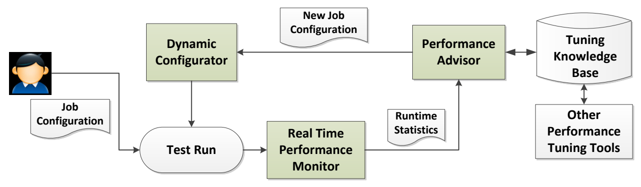

Figure 3: The tuning process used by MRONLINE.

namic configurator takes the newly chosen configurations and distributes the lists of new parameters to the slave configurator components. The slave configurators are responsible for activating the new changes for tasks that are running on their associated nodes.

Figure 3 illustrates the tuning process used by MRONLINE. After storing input data sets in HDFS, a user launches the application using a default configuration or a configuration based on rough understanding of application characteristics. The real-time performance monitor then starts to track runtime statistics including per task information such as task progress rate, CPU and memory utilization, the number of spilled records, I/O utilization and per node resource utilization. This information is periodically sent to the performance advisor that is implemented in the tuner component shown in Figure 2. The performance advisor analyzes and determines new configurations and sends them to the dynamic configurator that then changes the configuration for each task accordingly. The tuning process iterates until a desirable configuration is generated. MRONLINE supports both aggressive and conservative tuning strategies (Section 2). The performance tuning advisor can also be extended to communicate with other performance tuning tools, and the tuning rules can also be stored in a tuning knowledge base to be used across application runs.

4. TASK-LEVEL DYNAMIC CONFIGURATION

We extend YARN to support task-level dynamic configuration. In contrast to traditional YARN applications that use one configuration for executing all of the tasks, MRONLINE enables different configurations for different tasks.

Upon receiving a list of tasks and associated configuration mappings, the dynamic configurator writes per-task configuration files to the working directory of the corresponding application. When the tasks are assigned to containers, the slave configurator on the nodes retrieves the changed configuration files associated with the tasks. The launched tasks then read the changed configuration files, which in turn changes the configuration accordingly. Thus, each task can have a different configuration that can also vary.

Current implementation of YARN supports one fixed container size for all map tasks or all reduce tasks. We extend the resource scheduler to support requests that require different-sized containers. Specifically, we use a hash map data structure to keep track of the different-sized containers requested and corresponding operations such as assignment and release of the containers.

The key APIs supported by the dynamic configurator of MRONLINE are shown in Table 1. The functions $getConfigurableJobParameters$ and $getConfigurableTaskParameters$ return a set of configurable parameters for a specified job or task. The other three functions set the job or task configuration parameters to specified values. The APIs also allow other tuning algorithms, and not just ours, to easily tune the job parameters if needed.

5. GRAY-BOX BASED HILL CLIMBING ALGORITHM

In this section, we present the design of our tuner that systematically searches through the configuration space and finds a desirable configuration given a specific application, data set size and cluster configuration. To this end, we introduce a gray-box based hill climbing algorithm to tune the job parameter configurations for YARN applications. Our approach is inspired by the smart hill climbing algorithm [33] that was developed to provide black-box optimization for configuring web application servers. Our approach has three desirable properties: 1) it provides probabilistic guarantees on the closeness of a determined configuration to the optimal configuration; 2) it effectively tolerates noise in evaluated cost from factors such as resource contention; and 3) it adopts the weighted latin hypercube sampling (LHS) technique that helps improve the sampling quality and increases the convergence speed. Applying LHS in our approach allows us to partition the probability distributions of each parameter value into equal probability intervals, and sample a value from each interval, which leads to higher quality sampling. Moreover, we also consider the system-level monitoring information and apply it to the algorithm to speed up the search process. Thus, our algorithm offers a gray-box based method.

Algorithm 1 shows the details of tuning we have devised in MRONLINE. Our algorithm has two phases, a global search phase and a local search phase. The global search phase aims to find promising local areas to explore through efficient LHS sampling. The local search phase moves or narrows the search neighborhood area determined by any cost improvements compared to the current best configuration, until the size of the neighborhood is smaller than a predefined threshold. The local search phase also uses LHS to search the neighborhood and picks the best configuration.

Algorithm 1 Gray-Box based Hill Climbing.

```
1:  Initialize LHS parameters $k$, $m$, $n$, the threshold of neighborhood size
    $N_t$, the shrink factor $f$ and the threshold of global search $g$.
2:  $local\_search = 1, global\_search = 1$
3:  config[m] = LHS_sampling(m)
4:  $C_{cur}$=best(config[m])
5:  $N_{c_{cur}}$=adjust_neighbor($C_{cur}$)
6:  While $global\_search < g$ do
7:      if $local\_search == 1$ then
8:          while $N_{C_{cur}} > N_t$ do
9:              config[n]=LHS_sampling(n)
10:             $C_{candi}$=best(config[n])
11:             if($cost(C_{candi}) < cost(C_{cur})$)
12:                 $C_{cur}$=$C_{candi}$
13:                 $N_{c_{cur}}$=adjust_neighbor($C_{cur}$)
14:             else
15:                 $N_{c_{cur}}$=shrink_neighbor($C_{cur}$)
16:             endif
17:         endwhile
18:         $local\_search = 0$
19:     endif
20:     config[$m$]=LHS_sampling($m$)
21:     $C_{candi}$=best(config[$m$])
22:     if($cost(C_{candi}) < cost(C_{cur})$)
23:         $C_{cur} = C_{candi}$
24:         $N_{c_{cur}}$=adjust_neighbor($C_{cur}$)
25:         $local\_search = 1$
26:     else
27:         $global\_search$++
28:     endif
29: endwhile
```

The initial value for parameters such as the number of sampled configurations in the global search phase m (set to 24 in our tests), the number of sampled configurations in the local search phase n (16), the threshold of neighborhood size N_t (0.1), and the shrink factor f (0.75) that controls the ratio of the current neighborhood size to the shrunken neighborhood size, are set based on experimentation with the factors along with the theoretical analysis provided by the smart hill climbing algorithm [33]. The LHS interval, k, indicates the granularity of each parameter interval, and is set to 24 in our evaluation.

After the initialization, we enter the global search phase that uses LHS to generate m test configurations. Next, we configure the first m tasks to each use one of the generated configurations. As the tasks execute, the application performance is periodically monitored and the observations are used to estimate the cost of each configuration. We choose the configuration C_{cur} that has lowest estimated cost as the current search point and set the neighborhood size based on C_{cur}. Next, we switch to the local search phase, where we iteratively apply LHS sampling with n sampled configurations on the updated neighborhood with the center point C_{cur}. The dynamic configurator uses the newly calculated configurations to configure the newly launched tasks dynamically and the monitor component then gathers the execution statistics of the launched tasks. A candidate configuration C_{candi} is chosen based on the updated minimum estimated cost. The algorithm compares the estimated costs of a candidate configuration C_{candi} and the current best configuration C_{cur}. If the candidate configuration is better than the current configuration, it implies that there is a high possibility that a better configuration from the neighborhood with center configuration C_{candi} can be obtained. Otherwise, the algorithm shrinks the neighborhood size with the same center point C_{cur} with shrink factor f. The local search phase is then terminated, after the local search finds a local point with a neighborhood size smaller than a predefined threshold N_t. This implies that the algorithm finds a local optimal point.

After the local search phase, the algorithm enters the global search phase again to find a promising configuration space to analyze. If a point that is better than the current local optimal configuration is found, the system enters the local search phase to refine the search, otherwise the algorithm terminates after a specified number of iterations g (set to 5 in our tests).

There are several challenges we have to address to incorporate the gray-box hill climbing algorithm into MRONLINE.

Mapping Sampled Configurations to Tasks. Our monitor keeps track of the launched tasks and their associated configurations, as well as queued tasks both with and without assigned configurations. Given the fact that the tasks are independent from each other in YARN, when the configurations are generated, our tuning system randomly chooses a task from the queued tasks list and assigns one of the configurations to the task. The configuration is then further adjusted based on the task-related information.

Estimating Cost of Executed Tasks. Equation 1 shows how we estimate the cost of each task. We consider four factors: CPU utilization, memory utilization, ratio of the number of spill records to the number of map output or combiner output records, and ratio of the current task execution time to the maximum task execution time of all the tasks in the job. The goal of this formula is to reduce the task execution time and the number of spill records of all the tasks, while keeping the memory and CPU fully utilized. A caveat to avoid is that over-utilizing resources can create contention between tasks, thus increasing task execution time.

$$y = (1.0 - u_{mem}) + (1.0 - u_{cpu}) + num_{spill}/num_{mapoutput} + T_{task}/T_{maxtask}. \quad (1)$$

Utilizing Tuning Rules to Reduce the Number of Convergence Iterations and Resizing the Neighborhood Size. We consider the statistics collected from the monitor for enhancing search quality, which we detail in Section 6.

Moreover, the dependencies between the parameters are also considered in the algorithm. For example, the memory size of mappers should always be greater than the size of $io.sort.mb$. The parameter $mapreduce.reduce.shuffle.input.buffer.percent$ should always be greater than the parameter $mapreduce.reduce.shuffle.merge.percent$. Since a job with a small number of map tasks can restrict MRONLINE to try out all the parameters listed in Table 2, considering these tuning rules helps us converge to a suitable configuration quickly. In the evaluation section, we quantify how the tuning effectiveness is impacted by the length of a job.

6. TUNING RULES

In this section, we present the guidelines that we incorporate into our gray-box based algorithm (Section 5) for tuning MapReduce setups for the two target use cases identified in Section 2.3. We focus on the CPU and memory related parameters shown in Table 2. Other parameters are tuned using the hill climbing algorithm without using the additional tuning rules.

The tuning rules are aimed at improving the cluster utilization by adjusting containers to meet the task requirements and alleviate over- or under- utilization, as well as to reduce extra I/O traffic by carefully tuning the memory buffer options. The current implementation of MRONLINE provides per-task configuration, and

Configuration Parameters	Default Value
Memory Tuning	
$mapreduce.map.memory.mb$	1 GB
$mapreduce.reduce.memory.mb$	1 GB
$mapreduce.task.io.sort.mb$	100
$mapreduce.map.sort.spill.percent$	0.8
$mapreduce.reduce.shuffle.input.buffer.percent$	0.7
$mapreduce.reduce.shuffle.merge.percent$	0.66
$mapreduce.reduce.shuffle.memory.limit.percent$	0.25
$mapreduce.reduce.merge.inmem.threshold$	1000
$mapreduce.reduce.input.buffer.percent$	0.0
CPU Tuning	
$mapreduce.map.cpu.vcores$	1
$mapreduce.reduce.cpu.vcores$	1
$mapreduce.task.io.sort.factor$	10
$mapreduce.reduce.shuffle.parallelcopies$	5

Table 2: The key configuration parameters in MRONLINE.

application-wide auto-configuration, e.g., selection of the number of mappers and reducers, remains the focus of our future work.

6.1 Tuning Guidelines for the Considered Use Cases

Expedited Test Runs Use Case: The goal in this case is to reduce the number of test runs and find a near-optimal configuration for YARN applications. To this end, we allow MRONLINE to temporarily yield worse performance than the default configuration as the algorithm searches through the configuration space comprehensively and monitors the changes and their impact. We adopt an aggressive strategy that tries out as many cases as possible using a wave pattern for invoking parameter changes. We first update several tasks with new configurations at once, run and collect data about the tasks, and then adjust the parameter settings in the next wave based on the collected statistics from the previous wave. Moreover, MRONLINE controls the YARN application execution flow by holding off the launching of new tasks until the tasks in the previous wave are finished. This strategy slows down the test run execution, but allows the gray-box search algorithm to find a near-optimal configuration with high confidence.

Fast Single Run Use Case: In this case, we aim to improve performance in a single job run. Here, we adopt a conservative approach. We start the job with default values in the first wave and tune the parameters based on the collected information in the next. Moreover, MRONLINE does not interrupt the application task scheduling sequence, thus minimizing any negative impact of the gray-box algorithm on performance. The slave configurator running on each participating node uses the updated configuration file if available. If the configuration file is not present, the task is launched with the default configuration.

6.2 Memory Tuning

The first part of Table 2 shows parameters that decide the memory allocation of map and reduce tasks and the memory allocation for the sub-phases within these tasks. These parameters should be selected carefully. If the memory is set to too big of a value, it will waste memory resources that can be allocated to other containers, thus reducing overall cluster utilization. In contrast, if the memory is set to too small a value, it will incur resource contention leading to extra disk operations (even out of memory errors), thus degrading performance. The optimal values depend on the input data size, the map/reduce function, and the output data size.

To tune the memory allocation of map and reduce tasks, we adjust the parameters $mapreduce.map.memory.mb$ and $mapreduce.reduce.memory.mb$. For aggressive tuning, we obey the hill climbing algorithm using LHS sampling to try memory options within predefined memory ranges. After we obtain the task execution time and the memory utilization of map or reduce tasks that ran in the previous wave, we adjust memory bounds to help our hill climbing algorithm to narrow down the search space of these two parameters. This is done as follows. If we observe the memory utilization to be beyond 90%, it may cause over-utilization, so we increase the memory lower bounds to the 80^{th} percentile of sampled memory values. We also decrease the memory upper bounds to 80^{th} percentile of sampled memory values if we detect memory under-utilization (50% of memory utilization). When the tasks suffer from data skew and exhibit heterogeneous behavior, MRONLINE keeps track of the 80^{th} percentile value, and adjusts the bounds accordingly. For conservative tuning, we try different memory values only when they have high probabilities to yield better results. For the first wave, we conservatively use default values and collect statistics. We then estimate the memory size needed by the map or reduce tasks using this information. If the memory is underutilized, our hill climbing algorithm tries the lower value with a higher probability; otherwise, it tries the higher value with a higher probability.

The next finer grain level of memory parameter tuning includes three key parameters: $mapreduce.task.io.sort.mb$ in the map phase, and $mapReduce.reduce.shuffle.input.buffer.percent$ and $mapReduce.reduce.input.buffer.percent$ in the reduce phase. These affect performance in that they control the number of spill records written to disks. If enough memory is allocated both in the map and reduce phases, the number of spill records will be minimized. The parameter $mapreduce.tasks.io.sort.mb$ should not exceed the memory size of map tasks.

Ideally, the number of spill records in the map phase should equal the number of map output records. The number of spill records in the reduce phase should equal zero. Otherwise, the number of spill records is $3\times$ the number of map output records in the worst case. However, allocating more memory than needed would cause memory contention between the buffers and application logic, which negatively impacts job performance. The optimal memory buffer sizes depend on job and cluster characteristics.

The approach used for tuning the parameter $mapreduce.task.io.sort.mb$ is to configure the buffer size based on map output size by continuously monitoring the number of spill records and the size of map outputs. For conservative tuning, the value is set as the default value in the beginning. As the first few map tasks are started, the buffer size is set to the estimated map output size. If the ratio of increased number of spill records to increased map output records is greater than one, we increase the lower bound to 80^{th} percentile of the sampled values, since the current parameter value is not big enough to hold the map or combine outputs. If the ratio is one, MRONLINE decreases the upper bound to 80^{th} percentile of the sampled values. The rule is similar for aggressive tuning, except that before any application statistics are obtained, multiple configurations as determined by the hill climbing algorithm are first tried.

The parameter $mapreduce.map.sort.spill.percent$ decides when to spill data out to disks. It enables pipelining between map functions and disk writes. When the parameter $io.sort.mb$ is big enough, the value of the parameter

$mapreduce.map.sort.spill.percent$ should be set to a high value to ensure that disk writes are not triggered. Thus, for both aggressive and conservative tuning, we set the value to 0.99. If spilling extra records is unavoidable, we reset the parameter to its default value.

For tuning buffers in the reduce phase, we calculate the buffer sizes based on the estimated reduce input sizes. Specifically, the input size of each reducer is estimated by monitoring the number of spill records in the reduce phase and the sum of the size of partitions generated by each map output for the corresponding reducer.

The parameter $MapReduce.reduce.input.buffer.percent$ decides when to write the merged reduce output to disk. For example, when the reduce function requires only a small amount of memory, the parameter $mapreduce.shuffle.input.buffer.percent$ is set equal to the shuffle buffer to avoid any spills written to disks. Specifically, we use the memory utilization statistics from node managers to determine the memory usage of reducers.

The parameter $mapreduce.reduce.shuffle.merge.percent$ controls the trigger for memory to disk merge pipelining shuffle and memory-disk merge. It cannot exceed the reduce buffer size. For conservative tuning, the value is initially set as the default value. When the shuffle buffer is big enough to accommodate all the reduce input, the value can be set equal to the shuffle buffer to avoid additional disk I/Os. Otherwise, for safety, the value is set to $(mapreduce.reduce.shuffle.input.buffer.percent - 0.04)$ that has the same value difference with the parameter $mapreduce.reduce.shuffle.input.buffer.percent$ as in the default YARN configuration. Finally, we set the parameter $mapreduce.reduce.merge.inmem.threshold$ to 0, which makes the merge trigger only based on memory consumption.

6.3 CPU Tuning

Table 2 also lists the key parameters we consider for CPU tuning. YARN supports allocation of different number of CPUs to map and reduce tasks. The parameter $yarn.nodemanager.resource.cpu\text{-}vcores$ manages the number of CPU virtual cores that can be allocated for containers running in each slave node. If the value is 32, then on a 8-core machine, each virtual core has $1/4$ share of a physical core. Given that the number of physical cores per machine is fixed, a larger value yields smaller share per virtual core. This parameter is not suitable for dynamic tuning.

The parameters $mapreduce.map.cpu.vcores$ and $mapreduce.reduce.cpu.vcores$ directly control the CPU allocation of map and reduce tasks. The basic tuning rule is to allocate enough CPU resources to map and reduce tasks without sacrificing the cluster utilization. For conservative tuning, we start with the default value of 1, and collect container utilization information from the node manager. If full CPU utilization is observed, we increase the allocation by 1. If the task execution time is reduced and CPU under-utilization is not observed, we continue to increase the virtual core allocation.

The parameter $mapreduce.reduce.shuffle.parallelcopies$ determines the concurrent transfers executed by reduce tasks during shuffle. The desirable value depends on the amount of shuffled data. A higher amount leads to a higher number of parallel shuffles. For conservative tuning, starting from the default value, we increase the parameter in increments of 10 until the task execution time is not improved any further.

The parameter $mapreduce.task.io.sort.factor$ controls the concurrency of disk to disk merge with a default value of 10. The optimal value depends on the amount of data to be merged. For conservative tuning, we increase the value by 20 until the task execution time stops showing improvement.

The above discussion introduced all of the guidelines that we have incorporated into MRONLINE for parameter tuning. The provided APIs of the dynamic configurator are flexible, and can be used easily to incorporate additional tuning logic for more parameters as necessary.

7. IMPLEMENTATION

We have implemented MRONLINE on top of Hadoop-2.1.0-beta [8]. The online tuner is implemented as a daemon that extends the AbstractService class within YARN and includes the three components of Section 3 running in dedicated threads. The AbstractService class maintains service state and a list of service state change listeners. Once the service state has been changed, the service state change listeners are informed. The online tuner is implemented by extending CompositeService class within YARN. The CompositeService class consists of a list of AbstractService instances. It has a shutdown hook that allows the child services within the composite service to be shut down gracefully when the Java Virtual Machine (running the YARN instance) is shut down. Leveraging this feature allows us to gracefully shut down the online tuner and its child components as needed.

The monitor periodically gets a job counter for each submitted and running job from YARN through the JobClient interface. It then sends the job identifier and associated job counters to the tuner. The monitor also retrieves the task-level counter and cluster-level information such as the CPU, memory, network I/O, disk I/O from each slave node. The tuner takes the input from the monitor and determines the parameters that have to be changed and the values that should be assigned to these parameters. This is done by using the gray-box hill climbing algorithm and tuning rules described earlier.

After the tuner generates the list of parameters to be changed, it sends the information to the dynamic configurator. The dynamic configurator updates are then communicated to the working directory of corresponding jobs in HDFS through the JobClient interface. The slave configuration thread—that we have implemented in the node manager of YARN—then periodically checks whether per task configuration files are updated, picks up the values and changes the parameters accordingly.

System Overhead: The test runs using aggressive tuning can potentially have longer execution times than compared to the test runs using the default configuration. However, MRONLINE is much more effective than other offline tuning techniques in that we finish the test run in one trial instead of $20 - 40$ trials reported by works such as Gunther [25]. For the fast single run use case, we employ conservative tuning, which does not interfere with the application execution flow and thus have minimal overhead. The design of our monitor is also non-intrusive, since we leverage the JobClient APIs within YARN, which functions periodically in the standard setting as well. The dynamic configurator updates the parameter values, which consumes few resources. Thus, we note that the overall overhead of our system is negligible compared to default YARN.

8. EVALUATION

In this section, we show the effectiveness of our approach on a 19 node cluster. We note that the size of our testbed is in line with those considered by recent related works [15, 25]. We first show the performance improvement achieved for the studied MapReduce applications using the aggressive tuning strategy of MRONLINE. Next, we show that MRONLINE can generate desirable configurations that yield better application performance using conserva-

Benchmark	Input Data	Input Size	Shuffle Size	Output Size	#Map, #Reduce	Job Type
Bigram	Wikipedia	90.5 GB	80.8 GB	27.6 GB	676, 200	Shuffle
Inverted index	Wikipedia	90.5 GB	38 GB	10.3 GB	676, 200	Map
Wordcount	Wikipedia	90.5 GB	30.3 GB	8.6 GB	676, 200	Map
Text search	Wikipedia	90.5 GB	2.3 GB	469 MB	676, 200	Compute
Bigram	Freebase	100.8 GB	84.8 GB	77.8 GB	752, 200	Shuffle
Inverted index	Freebase	100.8 GB	21 GB	11 GB	752, 200	Compute
Wordcount	Freebase	100.8 GB	16.7 GB	9.4 GB	752, 200	Map
Text search	Freebase	100.8 GB	906 MB	229 MB	752, 200	Compute
Terasort	synthetic	100 GB	100 GB	100 GB	752, 200	Shuffle
BBP	N/A	0	252 KB	0	100, 1	Compute

Table 3: The benchmarks used in our tests and their characteristics.

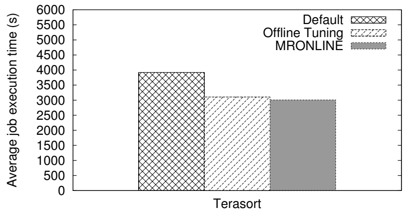

Figure 4: Job execution times under MRONLINE, offline tuning, and the default YARN configuration for Terasort for the expedited test runs use case.

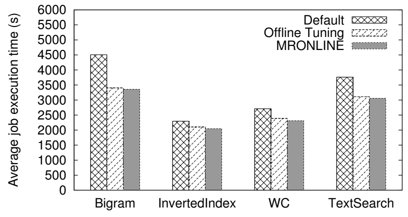

Figure 5: Job execution times under MRONLINE, offline tuning, and the default YARN configuration using the Wikipedia data set for the expedited test runs use case.

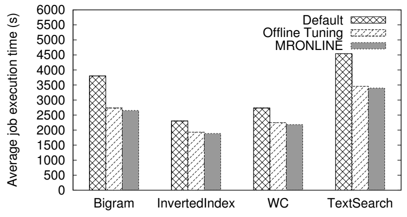

Figure 6: Job execution times under MRONLINE, offline tuning, and the default YARN configuration using the Freebase data set for the expedited test runs use case.

tive performance tuning as well. Next, we present results to illustrate the impact of job size on the effectiveness of MRONLINE. Finally, we show that MRONLINE can also improve application performance in a multi-tenant environment.

8.1 Methodology

Each node on our 19-node test cluster has two Intel Quad-core Xeon E5462 2.80 GHz CPUs, 12 MB L2 cache, 8 GB memory, a 320 GB Seagate ST3320820AS_P SATA disk, and a 1 $Gbps$ network card. One node works as the master and the rest of the 18 nodes work as slaves. The nodes are arranged in two racks with nine and ten nodes, respectively.

For the expedited test runs use case, we compare MRONLINE against both the default YARN configuration and a well-regarded offline tuning guide made available by an enterprise cloud provider [2]. For the fast single run use case and multi-tenant tests, we only compare MRONLINE against the default YARN configuration as the offline tuning approach is not capable of tuning and detecting runtime resource utilization hot spots etc. Moreover, since MRONLINE currently does not support the tuning of parameters such as the number of mappers and reducers and application-wide parameters, we use the same values of these parameters for both the offline tuning and MRONLINE. The values used for the default YARN configuration are the ones specified by the Hadoop Wiki [8], with the following changes. We use a block size of 128 MB. The number of virtual cores available for container allocation is 28 (4 for data nodes and node manager daemons), and the memory available for container allocation is 6 GB (2 GB for data nodes and node manager daemons).

Table 3 lists the representative MapReduce applications that we have used in our evaluation. Terasort, word count (WC), text search (Grep), and BBP that is a compute-intensive program that uses Bailey-Borwein-Plouffe to compute exact digits of PI, are available with the Hadoop distribution and serve as standard benchmarks. In addition, we also consider two more applications, $bigram$ and $inverted\ index$. Bigram [26] counts all unique sets of two consecutive words in a set of documents. Inverted index [26] generates word to document indexing from a list of documents. We classify the applications into three categories: Map intensive, Shuffle intensive and Compute intensive. Map intensive means that the map phase accounts for the largest part of the execution time (spent mostly doing I/Os). Shuffle intensive jobs spend the largest part of time in the shuffle phase, while compute intensive jobs spend the largest amount of time in the map phase doing computation.

We use two data sets to drive bigram, inverted index, word count and text search. Wikipedia [32] data set has the original size of 45 GB. We concatenate two copies of this data set together to produce a larger data set of 90 GB. Note that this does not change the workload characteristics of the data set. Freebase [18] is an open source 100.8 GB data set released by Google. It is a knowledge graph database for structuring human knowledge, which is used to support the collaborative web based data oriented applications. Finally, the data sets used by Terasort range from 2 GB to 100 GB in size and are generated synthetically using Teragen.

To account for variance in the results due to events such as network, disk I/O congestion, and hardware and file system errors, we repeat each experiment four times. In the following, we report the average results from the multiple runs.

8.2 Performance Improvement for the Expedited Test Runs Use Case

In this experiment, we evaluate the effectiveness of MRONLINE for the expedited test runs use case that employs aggressive tuning. We first run MRONLINE with each of the studied application to generate the best parameter configuration applicable. We then use the configurations to run applications and compare against applications running with the default configuration and with configurations produced using the offline tuning guide [2].

Figure 4 shows the average execution time for Terasort that experiences a 23$\%$ improvement in execution time under MRONLINE compared to the default configuration. Figure 5 and Figure 6 show the execution time for the four applications using the Wikipedia and

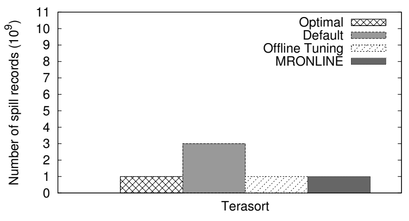

Figure 7: The number of spill records under MRONLINE, offline tuning, and the default YARN configuration for Terasort for the expedited test runs use case.

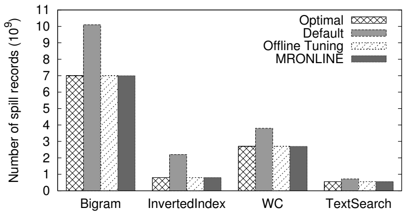

Figure 8: The number of spill records under MRONLINE, offline tuning, and the default YARN configuration using the Wikipedia data set for the expedited test runs use case.

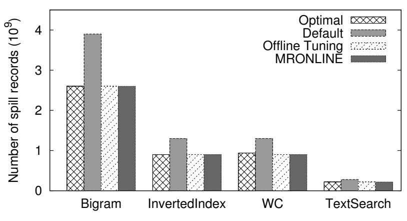

Figure 9: The number of spill records under MRONLINE, offline tuning, and the default YARN configuration using the Freebase data set for the expedited test runs use case.

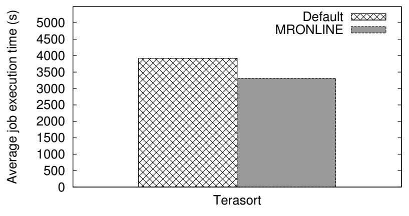

Figure 10: Job execution times under MRONLINE and the default YARN configuration using Terasort for the fast single run use case.

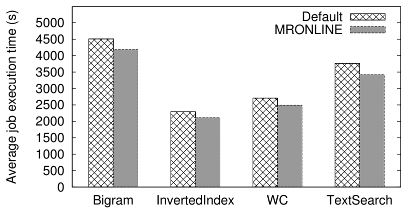

Figure 11: Job execution times under MRONLINE and the default YARN configuration using the Wikipedia data set for the fast single run use case.

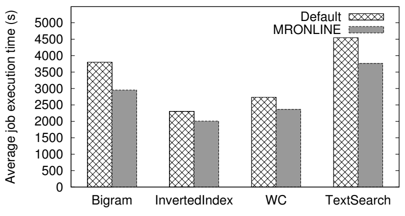

Figure 12: Job execution times under MRONLINE and the default YARN configuration using the Freebase data set for the fast single run use case.

Freebase data sets, respectively. We observe that for the Wikipedia data set MRONLINE reduces the job execution time by 25%, 11%, 14% and 19% for bigram, inverted index, word count, and text search, respectively. Similarly, the performance enhancement for the Freebase data set under MRONLINE is 30%, 18%, 20%, 25% compared to the default configuration for bigram, inverted index, word count, and text search, respectively. MRONLINE improves the performance mainly due to three factors: 1) it effectively reduces the number of spill records written and read from disks; 2) it increases the resource utilization by tuning the container size for mappers and reducers; and 3) it detects near-optimal values for other performance related parameters. We observe here that compared to the offline tuning guide, MRONLINE yields similar performance. However, MRONLINE is able to finish the gray-box based hill climbing algorithm within a single test run as there are around $600 - 800$ mappers and 200 reducers in these applications. In contrast, the offline guide took us much higher number of runs to determine a suitable configuration to use.

To further understand the effectiveness of MRONLINE, we studied how MRONLINE reduces the number of spill records. Figures 7, 8, and 9 show the number of spill records generated by the map tasks under MRONLINE, the default configuration, and configurations obtained through the offline tuning guide for the studied applications and data sets. In the figures, $Optimal$ refers to the number of records generated by a combiner in the map phase or generated by the map function if there is no combiner, and represents the number of spill records that an optimal configuration would produce. We can see that the numbers of spill records are effectively reduced to optimal for all applications by both MRONLINE and offline tuning. However, MRONLINE also minimizes the number of test runs compared to the offline tuning approach.

8.3 Performance Improvement for the Fast Single Run Use Case

In our next experiment, we compare the job execution time under MRONLINE using the conservative tuning strategy against the default YARN configuration on the Wikipedia, Freebase and the synthetic data sets. Conservative tuning is beneficial to applications that run once, since the goal is to improve performance and not necessarily find the best configurations. We run the applications under MRONLINE and measure the job execution time. The results are shown in Figures 10, 11 and 12. We observe that MRONLINE is able to improve the performance for all the studied applications and data sets from 8% (for word count using the Wikipedia data set) to up to 22% (for bigram using the Freebase data set). The significant reduction in execution time under MRONLINE is achieved because MRONLINE improves the cluster utilization by adjusting the container size, alleviates the I/O contention by reducing the spill records and searches for the optimal values for other performance related parameters.

This experiment demonstrates that MRONLINE can effectively reduce job execution time for applications that run once or for a few times. For such applications, the users do not need to worry about tuning application parameters before running the jobs and can achieve a speedup automatically by using MRONLINE.

8.4 The Impact of Job Size on the Effectiveness of Parameter Tuning

In our next experiment, we study how the effectiveness of parameter tuning using MRONLINE is affected by job size. To this end, we run Terasort with increasing input data sets ranging from $2\ GB$ to $100\ GB$. The number of reducers is set to about $1/4$ that of the number of mappers. For example, we have 4 reducers and 16 mappers for a job with a size of $2\ GB$, 12 reducers and 46 mappers for another job with a size of $6\ GB$. We execute MRONLINE for a single run of each job to generate an optimal configuration using aggressive tuning. We then use this configuration to run the job again and compare against the default YARN configuration. Figure 13 shows the results, where we can observe that MRONLINE reduces job execution time marginally for jobs with sizes smaller than $10\ GB$. The reason for this is that, for these jobs, MRONLINE does not have the sufficient number of mappers or reducers

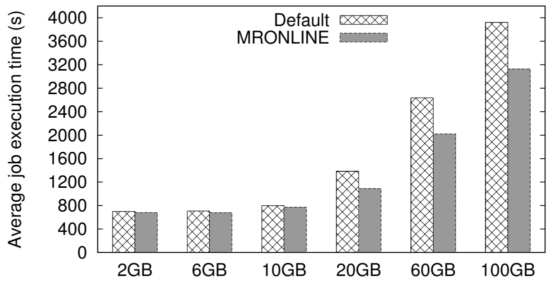

Figure 13: **Job execution time under** MRONLINE **and the default YARN configuration using Terasort with different data set sizes.**

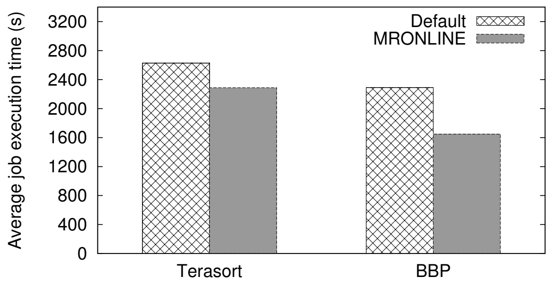

Figure 14: **Job execution time for Terasort and BBP.**

to search through the configuration space. Here, jobs finished before MRONLINE can find a good configuration. For jobs that use data sets greater than 20 GB, MRONLINE becomes effective and reduces the job execution time by 21%, 23%, and 20% for job sizes of 20 GB, 60 GB, and 100 GB, respectively. Once MRONLINE has determined a suitable configuration, increasing the data set size does not further improve performance under MRONLINE. This is because the additional mappers or reducers are unnecessary for MRONLINE as it is able to explore the design space thoroughly using the number available under the 20 GB case.

8.5 MRONLINE Tuning Efficiency in a Multi-tenant Environment

In our next experiment, we demonstrate that MRONLINE is particularly useful in a multi-tenant environment. For this test, we simultaneously run two MapReduce applications, Terasort and BBP, on our cluster using the fair share scheduling algorithm. We configure Terasort with an input data set size of 60 GB with 448 mappers and 200 reducers. BBP is configured to compute 0.5×10^6 digits of PI. We execute MRONLINE with aggressive tuning and produce desirable configurations for the two applications. We compare application characteristics under configurations generated using MRONLINE to that of the default YARN configuration.

Figure 14 shows the job execution time for Terasort and BBP. We observe that MRONLINE reduces the job execution time by 13% and 28% for Terasort and BBP, respectively. To further understand the performance impact of MRONLINE, we examined the memory and CPU utilization of Terasort and BBP under the two approaches shown in Figure 15 and Figure 16, respectively. In the figures, Terasort-m represents the average utilization of all mappers, while Terasort-r represents the same for all reducers. Similarly, BBP-m and BBP-r show the average utilization for all mappers and reducers, respectively, for BBP. We observe that under the default configuration the memory utilization of both the applications is below 50%. In contrast, MRONLINE improves the memory utilization of the two applications to above 80%, for both map and reduce tasks. For CPU utilization, we see that, with the exception of BBP-m, the utilization is below 25% for all cases under the default configuration. MRONLINE improves the CPU utilization by assigning fewer CPUs to Terasort and BBP-r. Note that the CPU utilization of BBP-m is around 99%. MRONLINE identifies this as CPU over-utilization, and allocates more CPU cores to BBP. Moreover, we note that the number of spill records for Terasort is reduced from 1.8×10^9 under the default configuration to 0.6×10^9 under MRONLINE. Reducing the number of spill records is beneficial, especially when disk I/O is one of the key performance bottleneck.

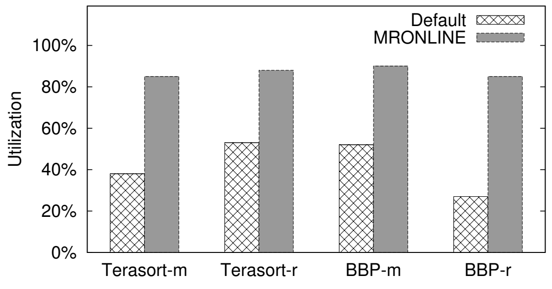

Figure 15: **Memory utilization for Terasort and BBP.**

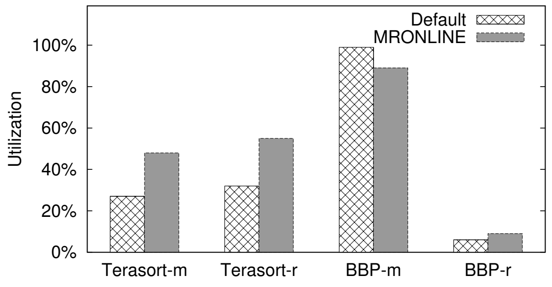

Figure 16: **CPU utilization for Terasort and BBP.**

This experiment shows that MRONLINE can effectively increase the memory utilization and CPU utilization for Terasort and BBP and thus reduce the job execution time. In other words, in this multi-tenant experiment where CPU is a bottleneck for BBP, MRONLINE successfully identifies idle CPUs and reassign a portion of them to BBP. Thus, we have demonstrated that MRONLINE can mitigate hot spots in the cluster and improve overall system utilization.

9. RELATED WORK

MapReduce Configuration Parameter Tuning: There are several works [13–15] that have focused on MapReduce job configuration tuning in recent years. Herodotos et al. [13–15] proposed a cost based optimization technique to help users identify good job configurations for MapReduce applications. The system consists of a profiler to get concise statistics including data flow information and cost estimation, a what-if engine to reason about the impact of parameter configuration settings, and a cost based optimizer to find good configurations through invocations of the what-if engine. The effectiveness of this approach depends on the accuracy of the what-if engine that uses a mix of simulation and model based estimation. MRONLINE is different from this work in that MRONLINE finds desirable configuration parameters through real test runs on real systems. Additionally, we use task level dynamic configuration to avoid multiple what-if iterations, and unlike such prior approaches are also able to adjust to dynamic cluster runtime status, e.g., network congestion or I/O congestion.

Gunther [25] is another offline tuning method that uses a genetic algorithm to identify good parameter configurations, tries one configuration per test run, and can take $20 - 40$ test runs. In contrast, MROnline can perform the tuning in a single job run. Moreover, we use a gray-box based approach that effectively exploits MapReduce runtime statistics, while Gunther is a black-box approach. In addition, we identify two specific use cases where MROnline is helpful; aggressive tuning aims to reduce the number of test runs, while conservative tuning can help improve the performance of jobs that only run once. In contrast, Gunther cannot help in either case.

AROMA [24] aims to automate the resource allocation and job configuration for heterogeneous clouds to satisfy SLAs while minimizing cost. AROMA uses a two-phase machine learning and optimization framework based on support vector machine based performance models. The offline phase classifies executed jobs using k-mediod clustering algorithm using CPU, network, and disk utilization patterns, while the online phase captures the resource utilization signature of tested applications. Finally, AROMA finds near optimal resource allocation and configuration parameters based on a pattern matching optimization method. Compared to MROnline, AROMA does not support dynamic configuration. Moreover, AROMA has to collect application resource utilization signatures before finding a near optimal configuration. This is not suitable for jobs that run once.

Parameter tuning guides [1, 2, 16, 17, 31] are also proposed by industry and vendors to help MapReduce non-experts to set desirable values for their applications. However, these tuning guides are based on heuristics. The burden is still on the end users to try out multiple parameter combinations, which is time consuming and cumbersome as discussed in Section 1.

MapReduce Performance Tuning: Performance tuning of the MapReduce framework itself [4, 20–22] has also gained a lot of attention from industry and the research community. MANIMAL [20] focuses on the efficiency of query processing of MapReduce framework and utilizes static program analysis techniques on user-defined functions (UDFs) to detect standard query optimization opportunities. To bridge the performance gap of MapReduce and parallel DBMS, Hadoop++ [4] tries to inject optimizations into UDFs, which makes query processing pipeline explicit and present it as a DB style physical query execution plan. This work has a different focus than MROnline. SUDO [36] analyzes UDFs to identify beneficial functional properties to optimize data shuffling for MapReduce frameworks by utilizing program analysis techniques. PerfXplain [22] provides a tool for non-expert users to tune MapReduce performance. This tool auto-generates an explanation for the queries comparing two jobs, which can help identify the reasons why inefficient or unexpected behavior happens. However, this work does not provide clear guidelines of what job configuration parameters should be used. Jiang et al. [21] provides a performance study of MapReduce, pinpointing factors that impact MapReduce performance including I/O, indexing, record decoding, grouping schemes and block level scheduling in database context. Although these works share with MROnline the goal of improving MapReduce application performance, these systems differ from MROnline because of different optimization aspects and different targeted environments. Moreover, to the best of our knowledge, MROnline is unique in its focus on YARN-based systems.

Simulation based performance tuning [6, 30] techniques have also been explored. Our own previous work, MRPerf [30], utilizes a simulation methodology to capture various factors that impact Hadoop performance. Similarly, Mumak [6] is designed as a MapReduce simulator for researchers to prototype features and predict their behavior and performance. These projects do not tune configuration parameters as such and only provide means to estimate application performance on given configurations, and thus are complementary to MROnline.

Parameter Tuning in Other Areas: A number of search techniques are proposed to find good configuration with high probability [33, 34] in other research areas as well. Recursive random search [34] is a black-box optimization approach that employs a heuristic search algorithm for tuning network parameter configurations. Smart hill climbing, designed for server parameter tuning, is another black-box optimization approach that is designed to improve the recursive random search algorithm. Smart hill climbing adopts a weighted LHS technique to improve the random sampling on the first phase. Moreover, the algorithm learns from past and searches the space using steepest descent direction and improves the search efficiency. The tuning algorithm of MROnline is inspired by the smart hill climbing algorithm. However, MROnline is unique in its focus on MapReduce, which is a different targeted problem than that addressed by prior works.

iTuned [5] concentrates on tuning database configuration parameters by adaptive sampling and uses an executor to support online experiments through a cycle-stealing paradigm. This approach is not suitable for MapReduce systems. JustRunIt [37] is an experiment based management system for virtualized data centers. It shares with MROnline the goal of tuning parameters using actual experiments that are cheaper, simpler and more accurate than performance models or simulations. However, the approach is not simply applicable to MapReduce.

10. CONCLUSION

MapReduce job parameter configuration significantly impacts application performance, yet extant implementations place the burden of tuning the parameters on application programmers. This is not ideal, especially because the application developers may not have the system-level expertise and information needed to select the best configuration. Consequently, the system is utilized inefficiently which leads to degraded application performance. In this paper, we present the design of MROnline, a tool that enables task-level dynamic configuration tuning to improve performance of MapReduce applications. MROnline expedites the test runs by trying out multiple configurations within a single test run. Given the large MapReduce parameter space, finding a near-optimal configuration in an efficient manner is challenging. To this end, we designed a gray-box based hill climbing algorithm to systematically search through the space and find a desirable configuration. To speedup the convergence iteration of our algorithm, we leverage MapReduce runtime statistics and consider design tuning rules for some of the key parameters. We have implemented MROnline on the YARN framework, and our evaluation shows that on a 19-node cluster and across a suite of six representative applications, MROnline achieves an average performance improvement of up to 30% compared to the typically used default YARN configurations.

We have focused on key parameters that affect task execution time. In our future work, we plan to investigate tuning of parameters, such as the number of mappers and reducers, which affect the overall application execution time.

11. ACKNOWLEDGMENT

This work was sponsored in part by the NSF under Grants CNS-1016793 and CCF-0746832. Min Li was also supported through an IBM PhD Fellowship. We thank the reviewers for their feedback and appreciate their support for our work. We also thank Minkyong Kim and Krishnaraj K Ravindranathan for the valuable discussions that helped us improve our design.

12. REFERENCES

[1] Cloudera. 7 tips for improving MapReduce performance, 2009. http://blog.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/.

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

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. USENIX OSDI*, 2004.

[4] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, 2010.

[5] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[6] A. S. Foundation. Mumak: MapReduce simulator, 2009. https://issues.apache.org/jira/browse/MAPREDUCE-728.

[7] A. S. Foundation. Apache Giraph, 2013. http://giraph.apache.org/.

[8] A. S. Foundation. Hadoop-2.1.0-Beta, 2013. http://www.trieuvan.com/apache/hadoop/common/hadoop-2.1.0-beta/.

[9] A. S. Foundation. Grep example, 2014. http://wiki.apache.org/hadoop/Grep.

[10] A. S. Foundation. Terasort example, 2014. https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html.

[11] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[12] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proc. USENIX NSDI*, 2011.

[13] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[14] H. Herodotou, F. Dong, and S. Babu. Mapreduce programming and cost-based optimization? Crossing this chasm with starfish. *Proceedings of the VLDB Endowment*, 4(12):1446–1449, 2011.

[15] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Proc. Conference on Innovative Data System Research*, 2011.

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

[17] Impetus. Hadoop performance tuning, 2012. https://hadoop-toolkit.googlecode.com/files/White paper-HadoopPerformanceTuning.pdf.

[18] G. Inc. Freebase data dumps, 2013. https://developers.google.com/freebase/data.

[19] A. Incubator. Spark: Lightning-fast cluster computing, 2013. http://spark.incubator.apache.org/.

[20] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, 2011.

[21] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *Proceedings of the VLDB Endowment*, 3(1-2):472–483, 2010.

[22] N. Khoussainova, M. Balazinska, and D. Suciu. Perfxplain: debugging mapreduce job performance. *Proceedings of the VLDB Endowment*, 5(7):598–609, 2012.

[23] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2012.

[24] P. Lama and X. Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proc. ACM International Conference on Autonomic Computing*, 2012.

[25] G. Liao, K. Datta, and T. L. Willke. Gunther: Search-based auto-tuning of mapreduce. In *Proc. Springer Euro-Par*, 2013.

[26] J. Lin and C. Dyer. Cloud9: A hadoop toolkit for working with big data, 2010. http://lintool.github.io/Cloud9/index.html.

[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2010.

[28] Twitter. Storm: Distributed and fault-tolerant realtime computation, 2013. http://storm-project.net/.

[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. In *Proc. ACM Symposium on Cloud Computing*, 2013.

[30] G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *Proc. IEEE MASCOTS*, 2009.

[31] T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2012.

[32] Wikipedia. Wikipedia data dumps, 2014. http://dumps.wikimedia.org/enwiki/latest/.

[33] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *Proc. ACM International Conference on World Wide Web*, 2004.

[34] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):196–205, 2003.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[36] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Proc. USENIX NSDI*, 2012.

[37] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. JustRunIt: Experiment-based management of virtualized data centers. In *Proc. USENIX ATC*, 2009.