Improving MapReduce Performance in Heterogeneous Environments with Adaptive Task Tuning

Dazhao Cheng, Jia Rao, Yanfei Guo, Xiaobo Zhou
Department of Computer Science
University of Colorado, Colorado Springs, USA
{dcheng,jrao,yguo,xzhou}@uccs.edu

Abstract

The deployment of MapReduce in datacenters and clouds present several challenges in achieving good job performance. Compared to in-house dedicated clusters, datacenters and clouds often exhibit significant hardware and performance heterogeneity due to continuous server replacement and multi-tenant interferences. As most Mapreduce implementations assume homogeneous clusters, heterogeneity can cause significant load imbalance in task execution, leading to poor performance and low cluster utilizations. Despite existing optimizations on task scheduling and load balancing, MapReduce still performs poorly on heterogeneous clusters.

In this paper, we find that the homogeneous configuration of tasks on heterogeneous nodes can be an important source of load imbalance and thus cause poor performance. Tasks should be customized with different settings to match the capabilities of heterogeneous nodes. To this end, we propose an adaptive task tuning approach, *Ant*, that automatically finds the optimal settings for individual tasks running on different nodes. Ant works best for large jobs with multiple rounds of map task execution. It first configures tasks with randomly selected configurations and gradually improves tasks settings by reproducing the settings from best performing tasks and discarding poor performing configurations. To accelerate task tuning and avoid trapping in local optimum, Ant uses genetic functions during task configuration. Experimental results on a heterogeneous cluster and a virtual cluster with varying hardware capabilities show that Ant improves the average job completion time by 23%, 11%, and 16% compared to stock Hadoop, customized Hadoop with industry recommendations, and a profiling-based configuration approach, respectively.

Categories and Subject Descriptors

D.4.8 [**Performance**]: Modeling and prediction; C.1.3 [**Other Architecture Styles**]: Heterogeneous (hybrid) systems

General Terms

Performance, Design

Keywords

Improving MapReduce Performance, Adaptive Task Tuning, Heterogeneous Clusters

1. INTRODUCTION

MapReduce, a parallel and distributed programming model on clusters of commodity hardware, has emerged as the de facto standard for processing a large set of unstructured data. Since big data analytics requires distributed computing at scale, usually involving hundreds to thousands of machines, access to such facilities becomes a significant barrier to practising big data processing in small business. Deploying MapReduce in datacenters or cloud platforms, offers a more cost-effective model to implement big data analytics. However, the heterogeneity in datacenters and clouds present significant challenges in achieving good MapReduce performance [2, 30].

Hardware heterogeneity occurs because servers are gradually upgraded and replaced in datacenters. Interferences from multiple tenants sharing the same cloud platform can also cause heterogeneous performance even on homogeneous hardware. The difference in processing capabilities on MapReduce nodes breaks the assumption of homogeneous clusters in MapReduce design and can result in load imbalance, which may cause poor performance and low cluster utilization. To improve MapReduce performance in heterogeneous environments, work has been done to make task scheduling [30] and load balancing [2] heterogeneity aware. Despite these optimizations, most MapReduce implementations such as Hadoop still perform poorly in heterogeneous environments. For the ease of management, MapReduce implementations use the same configuration for tasks. Existing research [13, 14, 21] has shown that MapReduce configurations should be set according to cluster size and hardware configurations. Thus, running tasks with homogeneous configurations on heterogeneous nodes inevitably leads to sub-optimal performance.

In this work, we propose a task tuning approach that allows tasks to have different configurations, each optimized for the actual hardware capabilities, on heterogeneous nodes. We address the following challenges in automatic MapReduce task tuning. First, determining the optimal task configuration is a tedious and error-prone process. A large number of performance-critical parameter can have complex interplays on task execution. Previous studies [12, 14, 17, 29] have shown that it is difficult to construct models to connect parameter settings with MapReduce performance. Second, there is no one-size-fit-all model for different MapReduce jobs and even different configurations are needed for different execution phases or input sizes. In a cloud environment, task configurations should also be changed in response to the changes in interferences. Finally, most MapReduce implementations use fixed task configurations that are set during job initializations [27]. Adaptive task tuning requires new mechanisms for on-the-fly task reconfiguration.

We present *Ant*, an adaptive self-tuning approach for task configuration in heterogeneous environments. Ant monitors task execution in large MapReduce jobs, which comprise multiple waves of tasks and optimizes task configurations as job execution progresses. It clusters worker nodes (either physical or virtual nodes) into groups according to their hardware configurations or the estimated capability based on interference statistics. For each node group, Ant launches tasks with different configurations and considers the ones that complete sooner as good settings. To accelerate tuning speed and avoid trapping in local optimum, Ant uses genetic functions *crossover* and *mutation* to generate task configurations for the next wave from the two best performing tasks in a group. We implement Ant in Hadoop, the popular open source implementation of MapReduce, and perform comprehensive evaluations with representative MapReduce benchmark applications. Experimental results on a physical cluster with three types of machines and a virtual cluster in our university cloud show that Ant improves the average job completion time by 23%, 11%, and 16% compared to stock Hadoop, customized Hadoop with industry recommendations (i.e., heuristic), and a profiling-based configuration approach, respectively. Our results also show that although Ant is not quite effective for small jobs with only a few waves of tasks, it can significantly improve the performance of large jobs. Experiments with Microsoft's MapReduce workload, which consist of 10% large jobs, demonstrate that Ant is able to reduce the overall workload completion time by 12.5% and 8% compared to heuristic- and profiling-based approaches.

The rest of this paper is organized as follows. Section 2 gives motivations on improvement of MapReduce configuration framework. Section 3 describes the design of Ant. Section 4.4 presents the details of the proposed self-tuning algorithm. Section 5 gives Ant implementation details. Section 6 presents the experimental results and analysis. Section 7 reviews related work. Section 8 concludes the paper.

2. MOTIVATIONS

In this section, we first introduce the default Hadoop configuration framework and then provide motivating examples to demonstrate that static task configurations do not optimize performance across different jobs and platforms.

2.1 Background

MapReduce is a distributed parallel programming model originally designed for processing a large volume of data in a homogeneous environment. Based on the default Hadoop framework, a large number of parameters need to be set before a job can run in the cluster. These parameters control the behaviors of jobs during execution, including their memory allocation, level of concurrency, I/O optimization, and the usage of network bandwidth. As shown in Figure 1,

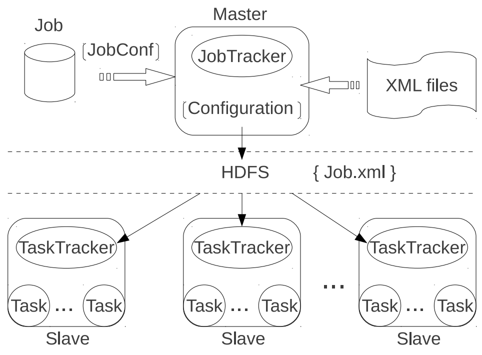


Figure 1: The Hadoop framework.

slave nodes load configurations from the master node where the parameters are configured manually. By design, tasks belonging to the same job share the same configuration.

In Hadoop, there are more than 190 configuration parameters, which determine the settings of the Hadoop cluster, describe a MapReduce job to the Hadoop framework, and optimize task execution [27]. Cluster-level parameters specify the organization of a Hadoop cluster and some long-term static settings. Changes to such parameters require rebooting the cluster to take effect. Job-level parameters determine the overall execution settings, such as input format, number of map/reduce tasks, and failure handling. These parameters are relatively easier to tune and have uniform effect on all tasks even in a heterogeneous environment. Task-level parameters control the fine-grained task execution on individual nodes and can possibly be changed independently and on-the-fly at runtime. Parameter tuning at the task level opens up opportunities for improving performance in heterogeneous environments and is our focus in this work.

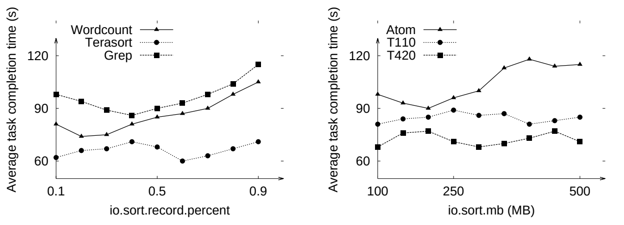
Hadoop installations pre-set the configuration parameters to default values assuming a reasonably sized cluster and typical MapReduce jobs. These parameters should be specifically tuned for a target cluster and individual jobs to achieve the best performance. However, there is very limited information on how the optimal settings can be determined. There exist rule of thumb recommendations from industry leaders (e.g., Cloudera [7] and MapR [20]) as well as academic studies [14, 17]. These approaches can not be universally applied to a wide range of applications or heterogeneous environments. In this work, we develop an online self-tuning approach for task-level configuration. Next, we provide motivating examples to show the necessity of configuration tuning for heterogeneous workloads and hardware platforms.

2.2 Motivating Examples

we created a heterogeneous Hadoop cluster composed of three types of machines listed in Table 1. Three MapReduce applications from the PUMA benchmark [1], i.e., *WordCount*, *Terasort* and *Grep*, each with 300 GB input data, were run on the cluster. We configured each slave node with four map slots and two reduce slots, and HDFS block size was set to 256MB. The heap size `mapred.child.java.opts` was set to 1 GB and other parameters were set to the default values. We measured the map task completion time in two different scenarios – heterogeneous workload on homo-

Table 1: Multiple machine types in the cluster.

Machine model	CPU	Memory	Disk
Supermicro Atom	4*2.0GHz	8 GB	1 TB
PowerEdge T110	8*3.2GHz	16 GB	1 TB
PowerEdge T420	24*1.9GHz	32 GB	1 TB



(a) Heterogeneous workloads. (b) Heterogeneous platforms.

Figure 2: The optimal task configuration changes with workloads and platforms.

geneous hardware and homogeneous workload on heterogeneous hardware. We show that the heterogeneity either in the workload or hardware makes the determination of the optimal task configuration difficult.

Figure 2(a) shows the average map completion times of the three heterogeneous benchmarks on a homogeneous cluster only consisting of the T110 machines. The completion times changed as we altered the values of parameter `io.sort.record.percent`. The figure shows that *wordcount*, *terasort*, and *grep* achieved their minimum completion times when the parameter was set to 0.4, 0.2, and 0.6, respectively. Figure 2(b) shows the performance of *wordcount* on machines with different hardware configurations. Map completion times varied as we changed the value of parameter `io.sort.mb`. The figure suggests that uniform task configurations do not lead to the optimal performance in a heterogeneous environment. For example, map tasks achieved the best performance on the Atom machine when the parameter was set to 125 while the optimal completion time on the T420 machine was due to the parameter being set to 275.

[**Summary**] We have shown that the performance of Hadoop applications can be substantially improved by tuning task-level parameters for heterogeneous workloads and platforms. However, parameter optimization is an error-prone process involving complex interplays among the job, the Hadoop framework and the architecture of the cluster. Furthermore, manual tuning still remains difficult due to the large parameter search space. As many MapReduce jobs are recurring or have multiple waves of task execution, it is possible to learn the best task configurations based on the feedback of previous runs. These observations motivated us to develop a task self-tuning approach to automatically configure parameters for various Hadoop jobs and platforms in an online manner.

3. ANT DESIGN AND ASSUMPTIONS

3.1 Architecture

Ant is a self-tuning approach for multi-wave MapReduce applications, in which job executions consist of several rounds of map and reduce tasks. Unlike traditional MapReduce implementations, Ant centers on two key designs: (1) tasks belonging to the same job run with different configurations matching the capabilities of the hosting machines; (2) the configurations of individual tasks dynamically change to search for the optimal settings. Ant first spawns tasks with random configurations and executes them in parallel. Upon task completion, Ant collects task runtimes and adaptively adjusts task settings according to the best-performing tasks. After several rounds of tuning, task configurations on different nodes converge to the optimal settings. Since task tuning starts with random settings and improves with job execution, ant does not require any priori knowledge of MapReduce jobs and is model independent. Figure 3 shows the architecture of Ant.

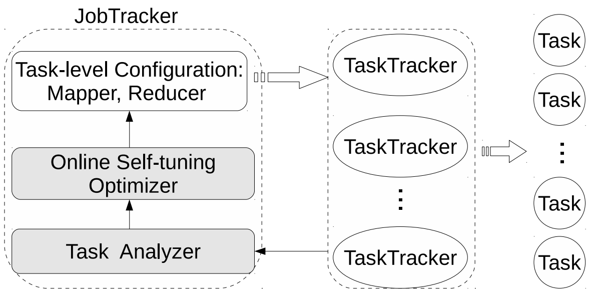


Figure 3: The architecture of Ant.

- **Self-tuning optimizer** uses a genetic algorithm (GA)-based approach to generate task configurations based on the feedback reported by the *task analyzer*. Settings that are top-ranked by the task analyzer are used to re-produce the optimized configurations.
- **Task analyzer** uses a fitness (utility) function to evaluate the performance of individual tasks due to different configurations. The fitness function takes into account task completion time as well as other performance critical execution statistics.

Ant operates as follows. When a job is submitted to the `JobTracker`, the configuration optimizer generates a set of parameters randomly in a reasonable range to initialize the task-level configuration. Then the `JobTracker` sends the randomly initialized tasks to their respective `TaskTrackers`. The steps of task tuning correspond to the multiple waves of tasks execution. Upon completing a wave, the task analyzer residing in the `JobTracker` recommends good configurations to the configuration optimizer for the next wave of execution. This process is repeated until the job completes.

3.2 Assumptions

Our findings that uniform task configurations lead to sub-optimal performance in a heterogeneous environment motivated the design of Ant, a self-tuning approach that allows differentiated task settings in the same job. The effectiveness of Ant relies on two assumptions – substantial performance improvement can be achieved via task configurations and the MapReduce jobs are long running ones (e.g., with multiple waves) which allow for task reconfiguration and performance optimization. There are two levels of heterogeneity that can affect task performance, i.e., task-level data skewness and machine-level varying capabilities. Although due

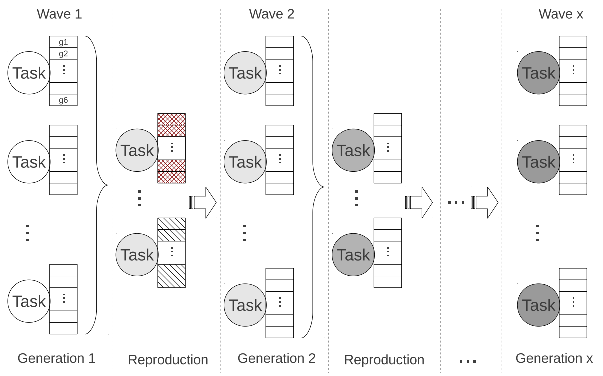


Figure 4: Task self-tuning process in Ant.

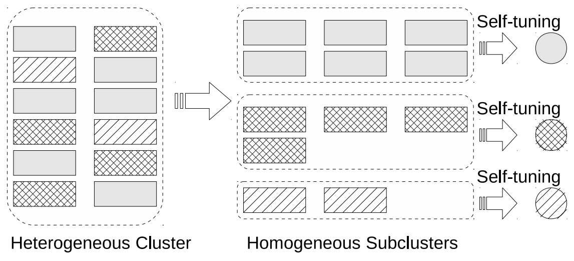


Figure 5: Ant on a heterogeneous cluster.

to data skew some tasks inherently take longer to finish, Ant assumes that the majority of tasks have uniform completion time with identical configurations. Ant focuses on improving performance for average tasks by matching task configurations to the actual hardware capabilities. To address hardware heterogeneity, Ant groups nodes with similar hardware configurations or capabilities together and compares parallel executing tasks to determine the optimal configurations for the node group. However, task skew and varying hardware capabilities due to interferences in multi-tenant clouds can possibly impede getting good task configurations. We will discuss how Ant address these issues in Section 6.8.

4. ADAPTIVE SELF-TUNING

Ant identifies good configurations by comparing the performance of parallel executing tasks on the nodes with similar processing capabilities in a self-tuning manner. Due to the multi-wave task execution in many MapReduce jobs, Ant is able to continuously improve performance by adapting task configurations. In this section, we first describe how Ant forms self-tuning task groups in which different configurations can be compared. We then discuss the set of parameters Ant optimizes and the utility function Ant uses to evaluate the goodness of parameters. Finally, we present the design of a genetic algorithm-based self-tuning approach and a strategy to accelerate the tuning speed.

4.1 Forming Self-tuning Groups

We begin with describing Ant's workflow in a homogeneous cluster and discuss how to form homogeneous subclusters in a heterogeneous cluster and a virtual cluster whose capability is dependent on the interference of co-located workloads.

Homogeneous cluster. In a homogeneous cluster, all nodes have the same processing capability. Thus, Ant considers the whole Hadoop cluster as a self-tuning group. Each node in the Hadoop cluster is configured with a predefined number of map and reduce slots. If the number of tasks (e.g., mappers) exceeds the available slots in the cluster (e.g., map slots), execution proceeds in multiple waves. Figure 4 shows the multi-wave task self-tuning process in a homogeneous cluster. Ant starts multiple tasks with different configurations concurrently in the self-tuning group and reproduces new configurations based on the feedback of completed tasks. We frame the tuning process as an evolution of configurations, in which each wave of execution refers to one configuration generation. The reproduction of generations is directed by a genetic algorithm which ensures that the good configurations in prior generations are preserved in new generations.

Heterogeneous cluster. In a heterogeneous cluster, as shown in Figure 5, Ant divides nodes into a number of homogeneous subclusters based on their hardware configurations. Hardware information can be collected by the `JobTracker` on the master node using the heartbeat connection. Ant treats each subcluster as an homogeneous cluster and independently applies the self-tuning algorithm to them. The outcomes of the self-tuning process are significantly improved task-level configurations, one for each subcluster. Since each subcluster has different processing capability, the optimized task configurations can be quite different across subclusters.

Virtual cluster. When running Hadoop in a virtual cluster, finding nodes with the same hardware capability is more challenging. Node configured with the same virtual hardware could have varying capacity due to interferences from co-located users [23]. Thus, Ant estimates the actual capabilities of virtual nodes based on low-level resource utilizations of virtual resources. Previous study found that MapReduce jobs are mostly bottlenecked by the slow processing of a large amount of data [5]. Excessive I/O rate and a lack of CPU allocation are signs of slow processing. Ant characterizes a virtual node based on two measured performance statistics: I/O rate and CPU steal time. Both statistics can be measured at the `TaskTracker` of individual nodes. Ant monitors the number of data bytes written to disk during the execution of a task. Since there is little data reuse in MapReduce jobs, the volume of writes is a good indicator of I/O access rate and memory demand. The CPU steal time is the amount of time that the virtual node is ready to run but failed to obtain CPU cycles because the hypervisor is serving other users. It reflects the actual CPU time allocated to the virtual node and can effectively calibrate the configuration of virtual hardware according to the experienced interferences. Ant uses the k-means [26] clustering algorithm to classify virtual nodes into configuration groups. In this work, we only classify virtual nodes at the beginning of self-tuning process. It is possible that background interference could change and the performance of virtual nodes will change over time. Thus, a re-clustering of the virtual nodes may be needed to form new tuning groups. Ant can be easily extended to adapt to varying interferences by performing re-clustering if significant changes in virtual node performance is detected.

4.2 Task-level Parameters

Table 2: Task-level parameters and search space.

Task-level parameters	Search space	Symbol
io.sort.factor	{1, 300}	g_1
io.sort.mb	{100, 500}	g_2
io.sort.record.percent	{0.05, 0.8}	g_3
io.sort.spill.percent	{0.1, 0.9}	g_4
io.file.buffer.size	{4K, 64K}	g_5
mapred.child.java.opts	{200, 500}	g_6

Parameter search space. Task-level parameters control the behavior of task execution, which is critical to the Hadoop. Previous studies have shown that a small set of parameters are critical to Hadoop performance. Thus, as shown in Table 2, we choose task-level parameters which have significant performance impact as the candidates for tuning. We further shrink the initial searching space of these parameters to a reasonable range in order to accelerate the search speed. This simple approach allows us to cut the search time down from a few hours to a few minutes.

Parameter sensitivity. Ant is designed to tune multiple task-level parameters jointly. Figure 6 shows that tuning different parameters can lead to different performance improvements. The improvement of job completion time varies from 10% to 37% when the six parameters are tuned separately. However, tuning parameters independently to their respective optimal values does not lead to the optimal overall performance. For instance, parameter `io.sort.mb` determines the total amount of buffer memory to use while sorting files and parameter `io.sort.factor` specify the number of streams to merge while sorting such files. These two parameters are highly correlated and should be tuned together to improve job performance.

4.3 Evaluating Task Configurations

To compare the performance of different task configurations, Ant requires a quantitative metric to rank configurations. As the goal of task tuning is to minimize job execution time, task completion time (TCT) is an intuitive metric to evaluate performance. However, TCT itself is not a reliable metric to evaluate task configurations. A longer task completion time does not necessarily indicate a worse configuration as some tasks are inherently longer to complete. For example, due to data skew, tasks that have expensive records in their input files can take more than five times longer to complete. Thus, we combine TCT with another performance metric to construct a utility function (or a fitness function in genetic algorithms). We found that most task mis-configurations are related to task memory allocations and incur excessive data spill operations. If either of the `kvbuffer` or the metadata buffer fills up, a map task spills intermediate data to local disks. The spills could lead to three times more I/O operations [7, 11]. Thus, Ant is designed to simultaneously minimize task completion time and the number of spills.

We define the fitness function of a configuration candidate (C_i) as: $f(C_i) = \frac{1}{TCT^2(C_i)\times(\#spills)}$, where TCT is the task completion time and *#spills* is the number of spill operations. Since majority of tasks have little or no data skew, we give more weight to TCT in the formulation of the fitness function. Task configurations with high fitness values will be favored in the tuning process. Note that the fitness function does not address the issue of data skew due to non-uniform record distributions in task inputs. We believe that configurations optimized for a task with inherently more data can be even harmful to normal tasks as allocating more memory to normal tasks incurs resource waste.

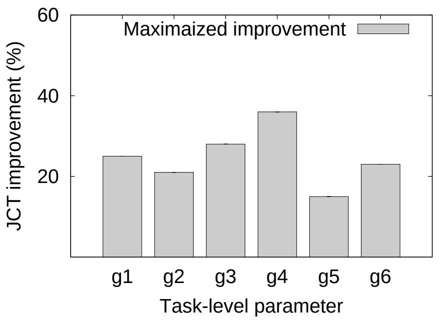


Figure 6: The sensitivity of task-level parameters.

Algorithm 1 Ant task self-tuning algorithm.

```
1: /*Evaluate the fitness of each completed task*/
2: $f(C_1), \cdots, f(C_i), \cdots, f(C_M)$
3: repeat
4:     if Any slot is available then
5:         Select two configuration candidates as parents
6:         Do Crossover and Mutation operation
7:         Use the obtained new generation $C_{new}$ to assign
   task to the available slot
8:     end if
9: until The running job completed
```

4.4 Task Self-tuning

Ant deploys an online self-tuning approach based on genetic algorithm to search the optimal task-level configurations. We consider MapReduce jobs composed of multiple waves of map tasks. The performance of individual task T is determined by its parameter set C. A set candidate C_i consisting of a number of selected parameters (refer to genes in GA), denoted as $C_i = [g_1, g_2, \cdots, g_n]$, represents a task configuration set, where n is the number of parameters. Each element g represents a task-level parameter as shown in Table 2.

Reproduction process. Ant begins with an initial configuration of randomly generated candidates for the task assignment. After that, it evolves individual task configuration to breed good solutions during each interval by using the genetic reproduction operations. As shown in Algorithm 1, Ant first evaluates the fitness of all completed tasks in the last control interval. Note that M represents the total number of the completed tasks in the last control interval. When there is any available slot in the cluster, it selects two configuration candidates as the evolving parents. Ant generates the new generation configuration candidates by using the proposed genetic reproduction operations. Finally, it assigns the task with the new generated configuration set to the available slot.

There are many variations of the reproduction algorithm obtained by altering the *selection*, *crossover*, and *mutation*

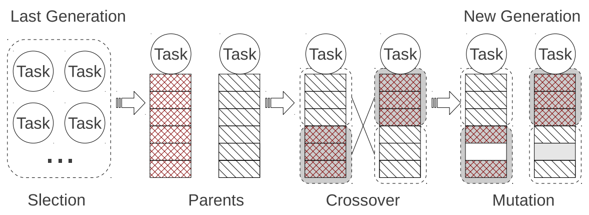


Figure 7: Reproduction operations.

operators as shown in Figure 7. The selection determines which two parents (task configuration sets) in the last generation will have offsprings in the next generation. The crossover operator determines how genes are exchanged between the parents to create those offsprings. The mutation allows for random alteration of genes. While the selection and crossover operators tend to increase the quality of the task execution in the new generation and force convergence, mutation tends to bring in divergence.

Parents selection. A popular selection approach is the Roulette Wheel (RW) mechanism. In this method, if fitness $f(C_i)$ is the fitness of completed task performance in the candidate population, its probability of being selected is $P_i = \frac{f(C_i)}{\sum_{i=1}^{M} f(C_i)}$, where M is the number of tasks completed in the previous interval. This allows candidates with good fitness values to have a higher probability of being selected as parents. The selection module ensures reproduction of more highly fit candidates compared to the number of less fit candidates.

Crossover. A crossover function is used to cut the sequence of elements from two chosen candidates (parents) and swap them to produce two new candidates (children). As crossover operation is crucial to the success of Ant and it is also problem dependent, an exclusive crossover operation is employed for each individual. We implement relative fitness crossover [8] instead of absolute fitness crossover operation, because it moderates the selection pressure and controls the rate of convergence. Crossover operation is exercised on configuration candidates with a probability, known as crossover probability (P_c).

Mutation. The mutation function aims to avoid trapping in the local optimum by randomly mutating an element with a given probability. Instead of performing gene-by-gene mutation at each generation, a random number r is generated for each individual. If r is larger than the mutation probability (P_m), the particular individual undergoes the mutation process. Otherwise, the mutation operation involves replacing a randomly chosen parameter with a new value generated randomly in its search space. This process prevents premature convergence of the population and helps Ant sample the entire solution space.

4.5 Aggressive Selection

The popular Roulette Wheel selection mechanism has a higher probability of selecting good candidates to be parents than bad ones. However, this approach still results in too many task evaluations, which in turn reduces the speed of convergence. Therefore, our selection procedure is more aggressive and deterministically selects good candidates to be parents. We use the following two strategies to accelerate the task-level parameter tuning.

Algorithm 2 Aggressive selection algorithm.

```
1: /*Select the best configuration candidate $C_{best}$*/
2: $best = \arg\max_i [f(C_i)], i \in \{1, \cdots, M\}$
3: /*Select another configuration set from the candidates
   with fitness scores that exceed the mean by WR*/
4: $Avg_f = \frac{1}{M} \sum_{i=1}^{M} f(C_i)$
5: if $f(C_i) > Avg_f$ then
6:     Select $C_{WR} = f(C_i)$ with possibility $P_i$
7: end if
8: Use $C_{best}$ and $C_{WR}$ as parents
```

Elitist strategy: We found that good candidates are more likely to produce good offsprings. In this work, an elitist strategy is developed similar to the proposed GAs in study [8]. Elitism provides a means for reducing genetic drift by ensuring that the best candidate is allowed to copy their attributes to the next generation. Since elitism can increase the selection pressure by preventing the loss of low salience genes of candidates due to deficient selection pressure, it improves the performance with regard to optimality and convergence. However, the elitism rate should be adjusted suitably and accurately because high selection pressure may lead to premature convergence. The best candidate with highest fitness value in the previous generation will be preserved as one of the parents in the next generation.

Inferior strategy: We also found that it is unlikely for two low fitness candidates to produce an offspring with high fitness. This is due to the fact that bad performance is often caused by a few key parameters and these bad settings continue to be inherited in real clusters. For instance, for an application that is both CPU and shuffle-intensive in a cluster with excessive I/O bandwidth and limited CPU resources, enabling compression of map outputs would stress the CPU and degrade application performance, regardless of others. The selection method should eliminate this configuration quickly. In order to quickly eliminate poor candidates, we calculate the mean fitness of the completed tasks for each generation and only select parents with fitness scores that exceed the mean.

Aggressive selection algorithm. Based on the above two aggressive selection strategies, the parents selection of the self-tuning reproduction is operated by an integrated selection algorithm. As shown in Algorithm 2, Ant firstly selects the best configuration candidate with the highest fitness in the last interval as one of the reproduction parents. Then it selects another configuration set from the candidates with fitness scores that exceed the mean by applying the Roulette Wheel approach. Finally, Ant generates the new generation configuration candidates by using the two selected candidates (i.e., C_{best} and C_{WR}) as the reproduction parents.

Furthermore, the aggressive selection strategies also reduce the impact of task skews during the tuning process. Long tasks due to data skews may add noises in the proposed GA-based task tuning. Taking the advantages of the aggressive selection, only the best configurations are possibly used to generate new configurations. It is unlikely that the tasks with skews would be selected as reproduction candidates. Thus, Ant would find the best configurations for the majority of tasks.

Table 3: The characteristics of benchmark applications used in our experiments.

Category	Type	Label	Input size (GB)	Input data	# Maps	# Reduces
Wordcount	CPU intensive	J1/J2/J3	100/300/900	Wikipedia	400/1200/3600	14/ 14/ 14
Grep	I/O intensive	J4/J5/J6	100/300/900	Wikipedia	400/1200/3600	14/ 14/ 14
Terasort	I/O intensive	J7/J8/J9	100/300/900	TeraGen	400/1200/3600	14/ 14/ 14

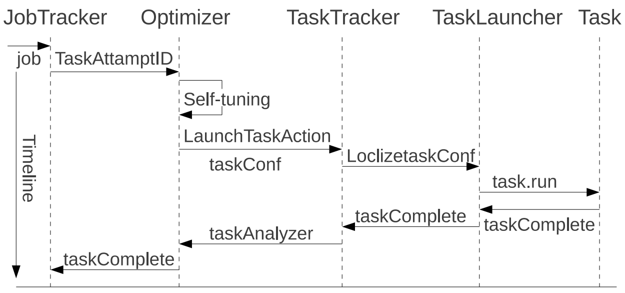


Figure 8: Ant work flow.

5. IMPLEMENTATION

Hadoop modification: We implemented Ant by modifying classes `JobTracker`, `TaskTracker` and `LaunchTaskAction` based on Hadoop version 1.0.3. We added a new interface `taskConf`, which is used to specify the configuration file of individual tasks while assigning them to slave nodes. Each set of task-level parameter set is tagged with its corresponding `AttemptTaskID`. Additionally, we added another new interface `Optimizer` to implement the GA optimization. During job execution, we created a method `taskAnalyzer` to collect the status of each completed task by using `TaskCounter` and `TaskReport`.

Ant execution process: At slave nodes, once a `TaskTracker` gets task execution commands from the `TaskScheduler` by calling `LaunchTaskAction`, it requires task executors to accept a `launchTask()` action from a local TaskTracker. As shown in Figure 8, Ant uses the `launchTask()` RPC connection to pass on the task-level configuration file description (i.e., `taskConf`), which is originally supported by the Hadoop. Ant creates a directory in the local file system to store the per-task configuration data for map/reduce tasks at `TaskTracker`. The directory is under the task working directory, and is tagged with `AttemptTaskID` which is obtained from `JobTracker`. Therefore, tasks can load their specified configuration items by accessing their task local file systems while initializing individual tasks by `Localizetask()`. Then after task localization, it kicks off a process to start a `MapTask` or `ReduceTask` thread to execute user-defined map and reduce functions.

Algorithm implementation: We implemented the self-tuning algorithm to generate the configuration sets for the new generation tasks in each control interval (i.e., 5 minutes). The selection of the control interval is a trade-off between the parameter searching speed and average task execution time. If the interval is too long, it will take more time to find good configurations. If the interval is too short, the task with new configurations may not complete and no performance feedback can be collected. Thus, we choose a control interval of 5 minutes which is approximately 2 times of the average task execution time. The mutated value of a parameter is randomly chosen from its search space. Since our aggressive selection algorithm prunes poor regions, we can use an atypically high mutation rate (e.g., $p_m = 0.2$) without impacting convergence. The value of p_m is empirically determined. A cut point is randomly chosen in each parent candidate configuration and all parameters beyond that point are swapped between the two parents to produce two children. We empirically set the crossover probability p_c to be 0.7.

6. EVALUATION

6.1 Experiment Setup

We evaluate Ant on a physical cluster composed of 1 Atom, 3 T110, 3 T420 (as shown in Table 1), 1 T320 (12-core CPUs, 24 GB RAM and 1 T hard disk), and 1 T620 (24-core CPUs, 16 GB RAM and 1 T hard disk). The master node is hosted on one T420 machine in the cluster. The servers are connected with Gigabit Ethernet. Each slave node is configured with four map slots and two reduce slots. The block size of HDFS is set to 256M due to the large input data size in the experiment. Ant is modified based on the version 1.0.3 of Hadoop implementation. FIFO scheduler is used in our experiments. We evaluate Ant using three MapReduce applications from the PUMA benchmark [1] with different input sizes as shown in Table 3, which are widely used in evaluation of MapReduce performance by previous works [6].

We compare the performance of Ant with two other main competitors in practical use: Starfish [14], a job profiling based configuration approach from Duke university, and Rules-of-Thumb [1] (Heuristic), another representative heuristic configuration approach from industry leader Cloudera [7]. For reference, we normalize the Job Completion Time (JCT) achieved by various approaches to the JCT achieved by the Hadoop stock parameter settings. Unless otherwise specified, we use the stock configuration setting of Hadoop implementation for the other items that are not listed in the Table 4. Note that both Heuristic and Starfish always maintain identical configuration files for job executions as described in Section 2. For fairness, the cluster-level and job-level parameters for all approaches (including the baseline stock configuration) in the experiments are set to suggested values by the rules of thumb from Cloudera [7]. For example, we roughly set the value of `mapred.reduce.tasks` (the number of reduce tasks of the job) to 0.9 times the total number of reduce slots in the cluster. We will discuss the performance impact of such job-level parameters in Section 6.8.

[1]Cloudera recommends a set of configuration items based on its industry experience, e.g.,`io.sort.record.percent` is recommended to set as $\frac{16}{16+avg-record-size}$, which is based on the average size of map output records. More rules are available in [7].

Table 4: Task-level parameter settings by current representative industry and academic approaches.

Approach	Rules-of-Thumb (Heuristic) from Cloudera						Starfish from Duke University					
Parameter	g_1	g_2	g_3	g_4	g_5	g_6	g_1	g_2	g_3	g_4	g_5	g_6
Wordcount	10	100mb	0.35	0.8	4k	-Xmx200m	10	50mb	0.15	0.8	4k	-Xmx200m
Grep	10	200mb	0.05	0.6	16k	-Xmx300m	100	200mb	0.1	0.66	16k	-Xmx300m
Terasort	10	150mb	0.15	0.7	32k	-Xmx250m	10	200mb	0.15	0.7	32k	-Xmx350m

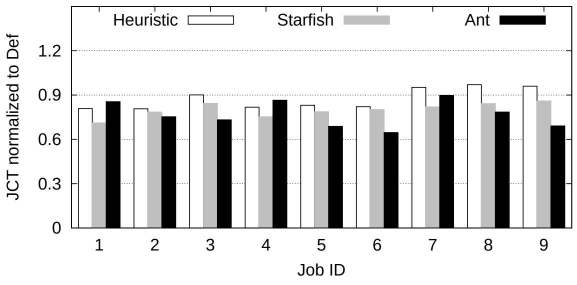


Figure 9: Job completion times on physical cluster.

Table 5: The number of spilling (J3, J6 and J9).

Approach	Wordcount	Grep	Terasort	Improve
Stock	2.4×10^4	4.7×10^4	1.7×10^5	baseline
Heuristic	1.9×10^4	3.8×10^4	1.4×10^5	18%
Starfish	1.75×10^4	3.6×10^4	1.25×10^5	23%
Ant	1.4×10^4	2.9×10^4	0.95×10^5	42%

Our evaluation firstly demonstrates the performance improvement achieved by Ant and analyzes the improvement from the task execution perspective. Then we show the effectiveness of Ant under a synthetic workload from Microsoft and the effectiveness of the aggressive selection strategy. Furthermore, we investigate the performance of Ant on the virtual cluster and how fast Ant can find a good configuration. Finally, we discuss the impact of job-level parameters, control overhead and multi-tenant environment.

6.2 Effectiveness of Ant

Reducing job completion time. Figure 9 compares various job completion times achieved by Heuristic, Starfish and Ant, respectively. The results demonstrate that all of these configuration approaches improve the job completion times more or less when compared with the performance achieved by Hadoop stock parameter setting (Stock). Figure 9 shows that Ant improves the average job completion time by 31%, 20% and 14% compared with Stock, Heuristic and Starfish on the physical cluster, respectively. This is due to the fact that Stock, Heuristic and Starfish all rely on a unified and static task-level parameter setting. Such unified configuration is apparently inefficient in the heterogeneous cluster. The results also reveal that Starfish is more effective than Heuristic on the physical cluster since Starfish benefits from its job profiling capability. The learning process of Starfish is more accurate than the experience based tuning of Heuristic to capture the characteristic of individual jobs.

Impact of job size. Figure 9 shows that Ant slightly reduces the completion time of small jobs compared with stock Hadoop, e.g., J1, J4 and J7. In contrast, Ant is more effective for large jobs, e.g., J3, J6 and J9. This is due to the fact that small jobs usually have short execution times and the self-tuning process can not immediately find the optimal configuration solutions. Thus, such small jobs are not favored by Ant.

Impact of workload type. Figure 9 reveals that Ant reduces the job completion time of I/O intensive workloads, i.e., *Grep* and *Terasort*, 10% more than that of CPU intensive workloads, i.e., *Wordcount*. This is due to the fact that Ant focuses on the task-level I/O operation parameter tuning and accordingly it affects more for I/O intensive workloads.

Overall, Ant achieves consistently better performance than Heuristic and Starfish do on the physical Hadoop cluster. This is due to its capability of adaptively tuning task-level parameters while considering various workload preferences and heterogeneous platforms.

6.3 Improvement on Task Execution

Reducing task execution time. Figure 10 shows the improvement comparisons in terms of the average task completion time by Ant, Heuristic and Starfish on the physical cluster respectively. It demonstrates that map task completion time improvements are much more than the reduce task improvements for all of the three configuration approaches. Ant outperforms Heuristic and Starfish significantly for map task performance improvement while obtaining similar reduce task performance improvement. This is due to the fact that Ant focuses on optimizing the intermediate data merging operation of multi-wave map tasks. Most production jobs have only one reduce wave since the number of reduce tasks in a job is typically set to be 0.9 times the number of available reduce slots. Hence, map tasks usually occur in multiple waves, while reduce tasks tend to complete in one to two waves for most real-world workloads. Accordingly, Ant focuses on improving the execution time of map tasks, which aims to take advantage of the multi-wave behaviors of map tasks. Understanding the wave behavior of tasks, such as the number of waves and the size of waves, would aid task configuration to improve cluster utilization.

Reducing number of data spills. Ant is designed to optimize task-level performance, particularly I/O operations during the map phase. The data spilling operations play an important role in the map phase and usually consume most execution time of map task processing [18]. Ant aims to reduce the map output data spilling times by tuning the buffer related parameters during execution, e.g., `io.sort.mb` and `io.sort.spill.percent`. To evaluate the effectiveness, we measure the data spilling times of each map task by analysis of the spilling logs in the experiments. Table 5 shows

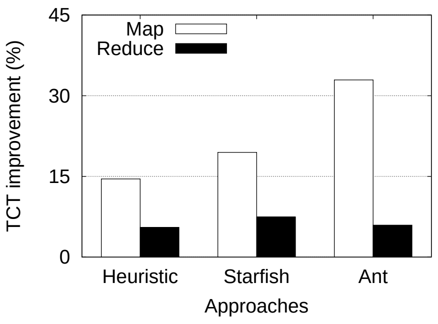


Figure 10: Task execution comparison.

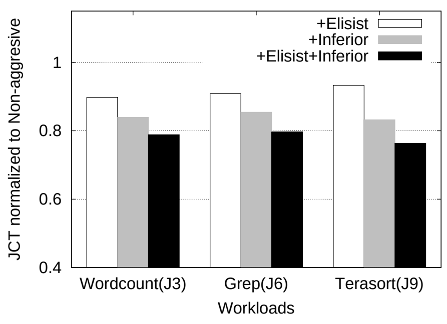


Figure 11: Effectiveness of aggressive selection in terms of job completion time.

Table 6: Performance improvement of Microsoft workload due to different approaches.

Input size	1-100GB	0.1-1TB	1-10TB	Total
Job fraction	40%	20%	10%	70%
$JCT_{Heuristic}$	2.1 h	6.3 h	11.7 h	19.1 h
$JCT_{Starfish}$	2 h	5.9 h	11.3 h	18.2 h
JCT_{Ant}	2.3 h	5.3 h	9.1 h	16.7 h

that Ant reduces the average data spilling times 42% while Heuristic and Starfish only reduce 18% and 23%, compared with the stock Hadoop configuration. This is due to the fact that Ant optimizes the intermediate data spilling, which satisfies individual needs of different workloads and platforms in the heterogeneous cluster.

6.4 Ant under Microsoft Workload

To better understand the effectiveness of Ant in a production environment, we analyzed 174,000 jobs submitted to a production analytics cluster in Microsoft datacenter in a single month in 2011 [3]. We use a synthetic workload, "MicroSoft-Derived (MSD)", which models Microsoft's production workload. It is a scaled-down version of the workload studied in [3] since our cluster is significantly smaller. MSD contains jobs with widely varying execution times and data set sizes, representing a scenario where the cluster is used to run many different types of batch applications. We do not run the Microsoft code itself. Rather, we mimic the distribution characteristics of the job size by running *Terasort* application with various input data sizes. We scale the workload down in two ways: we reduce the overall number of jobs to 87, and eliminate the largest 10% of jobs and the smallest 20% of jobs. Table 6 shows the job size distribution of MSD workload used in the experiment and the job completion time achieved by different tuning approaches. The results demonstrate that Ant reduces the overall job completion time of MSD workload 12.5% and 8% compared with Heuristic and Starfish.

6.5 Effectiveness of Aggressive Selection

We compare the performance impact of applying different selection strategies in Ant (described in Section 4.4) from the perspective of job completion time and convergence rate respectively. The experimental results demonstrate that the aggressive selection approach of Ant can effectively improve application performance for various workloads.

Figure 11 compares the job completion time achieved by different parents selection approaches. It shows that the elitist strategy improves the job completion time 9% and the inferior strategy improves the job completion time 17% compared with the no-aggressive strategy, respectively. Figure 11 also illustrates that applying the two strategies (i.e., elitist and inferior) together improves the job completion time 22% on average for the three applications compared with applying the no-aggressive strategy.

Figure 12 illustrates the impact of convergence rate in terms of the standard deviation [19] by using the aggressive selection strategies of Ant. It shows that no-aggressive selection has the worst convergence rate since Roulette Wheel selection mechanism still has a low probability of selecting bad candidates to be parents. However, both elitist and inferior strategies improve the convergence rate slightly in terms of the standard deviation compared with the no-aggressive strategy. Furthermore, the results demonstrate that applying inferior and elitist strategies together can effectively accelerate the convergence rate during the job execution. Thus, we integrate two strategies in the task evolution of Ant to improve searching performance.

6.6 Virtual Cluster Environment

We also built a virtual cluster in our university cloud, which consists of 108-core CPUs and 704 GB memory. VMware vSphere 5.1 was used for server virtualization. VMware vSphere module controls the CPU usage limits in MHz allocated to the virtual machines (VMs). We created a pool of VMs with different hardware configurations from the virtualized blade server cluster and run them as Hadoop slave nodes. There are 24 VMs, each with 2 vCPU, 4 GB RAM and 80 GB hard disk space, which are hosted on three blade servers in our cloud. Each blade server also hosts a representative transactional application RUBiS as the interference producer, which models an online auction site. We created three VMs, each with 4 vCPUs, 8 GB RAM and 80 GB hard disk space, for the RUBiS benchmark application on the three servers respectively. The number of concurrent users is set to 800, 2500 and 4500 for the three servers. All VMs ran Ubuntu Server 12.04 with Linux kernel 3.2. The cluster-level configurations for Hadoop are the same as those in the physical cluster (Section 6.1). The number of reduce tasks is set to 42, which is 0.9 times the number of the available reduce slots in the cluster.

We empirically set the number of subclusters to three in

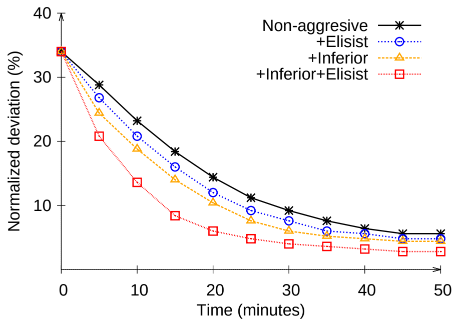


Figure 12: Effectiveness of aggressive selection in terms of convergence rate.

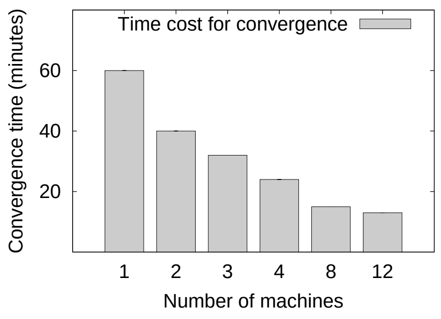


Figure 14: Searching speed analysis.

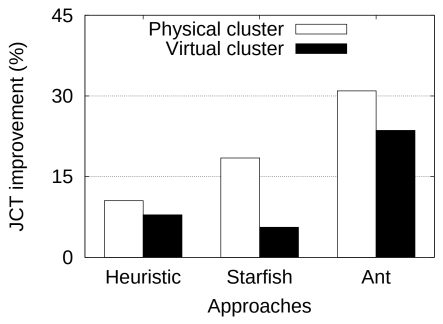


Figure 13: Impact of environments.

this experiment. We then divide the virtual cluster into three subclusters by using k-means clustering approach. VMs having similar CPU steal time and I/O rate are classified into the same subcluster. We obtain three subclusters as follows: a subcluster of 4 VMs with low steal time and high I/O rate, a subcluster of 8 VMs with middle steal time and I/O rate, and a subcluster of 12 VMs with high steal time and low I/O rate. Note that each subcluster is treated as an independent homogeneous cluster to evolve the task-level optimal configuration set as described in Section 4.1.

Performance improvement impact. Similar to the scenario of the physical cluster, Ant improves the average job completion time by 23%, 11% and 16% compared with Stock, Heuristic and Starfish on the virtual cluster, respectively. However, the performance of Starfish is slightly worse than that of Heuristic in this scenario since the resource interference in the virtual environment significantly reduces the accuracy of Starfish job profiling. Figure 13 compares the job completion time improvement achieved by Ant, Heuristic and Starfish under the physical and virtual environments, respectively. The results demonstrate that the average job completion time of Ant on the physical cluster is improved by 8% compared with that on the virtual cluster. It shows that Ant is more effective in the physical environment. This is due to the fact that MapReduce perception of cluster topology and resource isolation may be different from the actual settings on hardware when running in a virtual environment. As the result, the corresponding parameter tuning strategies are less effective in a virtual cluster than in a physical cluster. Figure 13 also shows that Ant improves the absolute job completion time at least by 9% and 7% compared with Heuristic and Starfish on the physical cluster and on the virtual cluster, respectively.

6.7 Search Speed

The searching speed of Ant represents the time cost to find a stable task configuration set for a running job on the cluster. A slow searching speed will reduce the benefit from a good task-level configuration, especially for small jobs. This is due to the fact that a small job may complete before Ant can find out an optimal setting for this job. This is also the reason that Ant is more effective for large jobs than for small jobs as shown in Section 6.2. Our experiments reveal that there are two main factors contributing significant impact on the searching speed of Ant.

Impact of cluster size. The number of available nodes in the cluster is an important factor that affects the searching speed of Ant. Figure 14 enumerates the convergence cost of Ant with various numbers of machines in our experiments. The result shows that the convergence cost decreases as the number of machines increases in the cluster. The searching process relies on the online task execution learning. The more nodes available in the cluster, the more opportunities there are to learn the characteristics of the running job.

Impact of machine type. Figure 15 demonstrates the searching speed of Ant in the physical cluster is faster than that in the virtual cluster. We define that the parameter value is stable when its standard deviation value is smaller than 10%, which is widely used in evaluation of the convergence speed in previous study [19]. The result in Figure 15 shows that Ant takes 18 and 35 minutes to find a good configuration in the physical cluster and in the virtual cluster, respectively. Note that the actual resource allocation for each node in the virtual cluster is typically time-varying due to the interference issues [25].

6.8 Discussions

Job-level parameters. Currently, job-level configurations mostly rely on personal experiences and/or job-profiling based approaches. We quantify the performance impact of an important job-level parameter, i.e., `mapred.reduce.tasks`. Our experimental results suggest that the number of reduce tasks should be set to 1-5 times the number of available reduce slots in the cluster. The default value of a single reduce task setting is worst while too many reduce task settings can also lead to performance degradation. Thus, for fairness, we set the value of parameter `mapred.reduce.tasks` to 0.9 times the number of available reduce slots in the cluster for

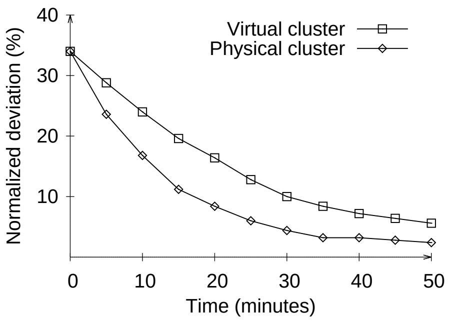


Figure 15: Searching speed analysis.

all jobs in our experiments.

Overhead of Ant. Although Ant relies on a centralized control framework, it does not launch any new process for the controller. Instead, it only sends a tiny per-task configuration description file to TaskTrackers when initializing each task. These task-level configuration files are distributed to individual slave nodes along with `LaunchTaskAction` by Hadoop default RPC connections. We find that the self-tuning optimization algorithm takes approximately 25-40 ms to complete. This overhead is negligible compared to the control interval of 5 minutes.

Multi-tenant scenario. As MapReduce scales to ever-larger platforms, concerns about cluster utilization grow [4]. Multi-tenancy support becomes an essential need in MapReduce. However, sharing a MapReduce cluster among multiple jobs from different users poses significant challenges in resource allocation and job configurations. For instance, interferences among different jobs may create straggler tasks that significantly slow down the entire job execution, and may lead to severe performance degradations [10, 28].

7. RELATED WORK

A number of previous studies have shown that MapReduce performance can be improved by various optimization approaches from following aspects.

Framework modification. Studies have demonstrated that it is effective to improve MapReduce performance by modifying the default Hadoop framework. Rao *et al.* proposed Sailfish [22], a new MapReduce framework for large-scale data processing. The core of Sailfish is aggregating intermediate data, specifically data produced by map tasks and consumed later by reduce tasks, to improve job performance by batching disk I/O. Jinda *et al.* [15] proposed a new data layout, namely coined Trojan Layout, which internally organizes data blocks into attribute groups according to the workload in order to improve data access times. It is able to schedule incoming MapReduce jobs to data block replicas with the most suitable Trojan Layout. Guo *et al.* proposed iShuffle [11], a novel user-transparent shuffle service that provides optimized data shuffling to improve job performance. It decouples shuffle from reduce tasks and proactively pushes data to be shuffled to Hadoop node via a novel shuffle-onwrite operation in map tasks. Dittrich *et al.* proposed Hadoop++ [9], a new index and join technique to improve runtime of MapReduce jobs. Vavilapalli *et al.* presented the next generation of the Hadoop compute platform known as YARN [24]. Similar to the first generation, it employs a unified configuration policy for all tasks. None of those approaches considered modifying the default Hadoop configurations to improve MapReduce performance.

Heterogeneous environment. As heterogeneous hardware is applied to Hadoop clusters, how to improve MapReduce performance in heterogeneous environments attracts much attention [2, 30, 31]. Ahmad *et al.* [2] identified key reasons for MapReduce poor performance on heterogeneous clusters. Accordingly, they proposed an optimization based approach, Tarazu, to improve MapReuce performance by communication-aware load balancing. Zaharia *et al.* [30] designed a robust MapReduce scheduling algorithm, LATE, to improve the completion time of MapReuce jobs in a heterogeneous environment. They paid little attention to optimizing Hadoop configurations, which has a significant impact on the performance of MapReduce jobs, especially in a heterogeneous Hadoop cluster.

Parameter configuration. Recently, a few studies start to explore how to optimize Hadoop configurations to improve job performance. Herodotou *et al.* [12] proposed several automatic optimization based approaches for MapReduce parameter configuration to improve job performance. Kambatla *et al.* [16] presented a Hadoop job provisioning approach by analyzing and comparing resource consumption of applications. It aimed to maximize job performance while minimizing the incurred cost. Lama and Zhou designed AROMA [17], an approach that automated resource allocation and configuration of Hadoop parameters for achieving the performance goals while minimizing the incurred cost. AROMA achieves the optimal configuration by running a small sample of submitted jobs. If the workload is complex and dynamic, e.g., *Gridmix*, its profiling may not be accurate. Herodotou *et al.* proposed Starfish [14], an optimization framework that hierarchically optimizes from jobs to workflows by searching for good parameter configurations. It utilizes dynamic job profiling to capture the runtime behavior of map and reduce at the granularity of phase level and helps users fine tune Hadoop job parameters. These approaches mostly rely on the default Hadoop framework and configure the parameters by static settings. They are often not effective when the workload changes or the cluster platform becomes heterogeneous.

8. CONCLUSION

Although an unified design framework, such as MapReduce, is convenient and easy to use for large-scale parallel and distributed programming, it ignores the differentiated needs in the presence of various platforms and workloads. In this paper, we tackle a practical yet challenging problem of automatic configuration of large-scale MapReduce workloads in heterogeneous environments. We have proposed and developed a self-adaptive task-level tuning approach, Ant, that automatically finds the optimal settings for individual jobs running on heterogeneous nodes. In Ant, tasks are customized with different settings to match the capabilities of heterogeneous nodes. It works best for large jobs with multiple rounds of map task execution. Our experimental results demonstrate that Ant can improve the average job completion time by 23%, 11%, and 16% compared to stock Hadoop, customized Hadoop with industry recommendations, and a profiling-based configuration approach, respectively. In future work, we plan to extend Ant to a multi-tenant MapReduce environment.

Acknowledgement

This research was supported in part by U.S. NSF CAREER award CNS-0844983, research grants CNS-1422119, CNS-1320122, CNS-1217979, and NSF of China research grant 61328203.

9. REFERENCES

[1] PUMA: Purdue mapreduce benchmark suite. `http://web.ics.purdue.edu/~fahmad/benchmarks.htm`.

[2] Ahmad, F., Chakradhar, S., Raghunathan, A., and Vijaykumar, T. N. Tarazu: optimizing mapreduce on heterogeneous clusters. In *Proc. Int'l Conf. on Architecture Support for Programming Language and Operating System (ASPLOS)* (2012).

[3] Appuswamy, R., Gkantsidis, C., Narayanan, D., Hodson, O., and Rowstron, A. Scale-up vs scale-out for hadoop: Time to rethink? In *Proc. ACM Symposium on Cloud Computing (SoCC)* (2013).

[4] Carrera, D., Steinder, M., Whalley, I., Torres, J., and Ayguadé, E. Enabling resource sharing between transactional and batch workloads using dynamic application placement. In *Proc. ACM/IFIP/USENIX Int'l Conf. on Middleware (Middleware)* (2008).

[5] Chiang, R. C., and Huang, H. H. Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proc. Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2011).

[6] Cho, B., Rahman, M., Chajed, T., Gupta, I., Abad, C., Roberts, N., and Lin, P. Natjam: Eviction policies for supporting priorities and deadlines in mapreduce clusters. In *Proc. ACM Symposium on Cloud Computing (SoCC)* (2013).

[7] Cloudera. Configuration parameters. http://blog.cloudera.com/blog/author/aaron/.

[8] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. on Evolutionary Computation 6* (2002), 182–197.

[9] Dittrich, J., Quiané-Ruiz, J.-A., Jindal, A., Kargin, Y., Setty, V., and Schad, J. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)* (2010).

[10] Guo, Y., Rao, J., Jiang, C., and Zhou, X. Moving mapreduce into the cloud with flexible slot management. In *Proc. Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2014).

[11] Guo, Y., Rao, J., and Zhou, X. ishuffle: Improving hadoop performance with shuffle-on-write. In *Proc. Int'l Conference on Autonomic Computing (ICAC)* (2013).

[12] Herodotou, H., and Babu, S. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. In *Proc. Int' Conf. on Very Large Data Bases (VLDB)* (2011).

[13] Herodotou, H., Dong, F., and Babu, S. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proc. ACM Symposium on Cloud Computing (SoCC)* (2011).

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

[15] Jinda, A., Quian-Ruiz, J., and Dittrich, J. Trojan data layouts: Right shoes for a running elephant. In *Proc. of ACM Symposium on Cloud Computing (SoCC)* (2011).

[16] Kambatla, K., Pathak, A., and Pucha, H. Towards optimizing hadoop provisioning in the cloud. In *Proc. USENIX HotCloud Workshop* (2009).

[17] Lama, P., and Zhou, X. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proc. Int'l Conf. on Autonomic computing (ICAC)* (2012).

[18] Li, X., Wang, Y., Jiao, Y., Xu, C., and Yu, W. Coomr: Cross-task coordination for efficient data management in mapreduce programs. In *Proc. Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2013).

[19] Li, Z., Cheng, Y., Liu, C., and Zhao, C. Minimum standard deviation difference-based thresholding. In *Proc. Int'l Conference on Measuring Technology and Mechatronics Automation (ICMTMA)* (2010).

[20] MapR. The executive's guide to big data. http://www.mapr.com/resources/white-papers.

[21] Pettijohn, E., Guo, Y., Lama, P., and Zhou, X. User-centric heterogeneity-aware mapreduce job provisioning in the public cloud. In *Proc. Int'l Conference on Autonomic Computing (ICAC)* (2014).

[22] Rao, S., Ramakrishnan, R., Silberstein, A., Ovsiannikov, M., and Reeves, D. Sailfish: A framework for large scale data processing. In *Proc. of ACM Symposium on Cloud Computing (SoCC)* (2012).

[23] Sharma, B., Wood, T., and Das, C. R. Hybridmr: A hierarchical mapreduce scheduler for hybrid data centers. In *Proc. IEEE Int'l Conference on Distributed Computing Systems (ICDCS)* (2013).

[24] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. Apache hadoop yarn: Yet another resource negotiator. In *Proc. ACM Symposium on Cloud Computing (SoCC)* (2013).

[25] Verma, A., Cherkasova, L., and Campbell, R. H. Resource provisioning framework for mapreduce jobs with performance goals. In *Proc. ACM/IFIP/USENIX Int'l Middleware Conference (Middleware)* (2011).

[26] Wang, C., Rayan, I. A., Eisenhauer, G., Schwan, K., Talwar, V., Wolf, M., and Huneycutt, C. Vscope: Middleware for troubleshooting time-sensitive data center applications. In *Proc. ACM/IFIP/USENIX Int'l Middleware Conference (Middleware)* (2012).

[27] White, T. *Hadoop: The Definitive Guide*, 3rd ed. O'Reilly Media / Yahoo Press, 2012.

[28] Wolf, J., Rajan, D., Hildrum, K., Khandekar, R., Kumar, V., Parekh, S., Wu, K., and Balmin, A. Flex: A slot allocation scheduling optimizer for mapreduce workloads. In *Proc. ACM/IFIP/USENIX Int'l Middleware Conference (Middleware)* (2010).

[29] Yigitbasi, N., Willke, T., Liao, G., and Epema, D. Towards machine learning-based auto-tuning of mapreduce. In *Proc. IEEE/ACM Int'l Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2013).

[30] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. Improving mapreduce performance in heterogeneous environments. In *Proc. USENIX Symposium on Operating System Design and Implementation (OSDI)* (2008).

[31] Zhang, Q., Zhani, M. F., Boutaba, R., and Hellerstein, J. L. Harmony: Dynamic heterogeneity-aware resource provisioning in the cloud. In *Proc. IEEE Int'l Conference on Distributed Computing Systems (ICDCS)* (2013).