Gunther: Search-Based Auto-Tuning of MapReduce

Guangdeng Liao, Kushal Datta, and Theodore L.Willke

Intel Labs, Hillsboro, Oregon, USA
{guangdeng.liao,kushal.datta,theodore.l.willke}@intel.com

Abstract. MapReduce has emerged as a very popular programming model for large-scale data analytics. Despite its industry-wide acceptance, the open source Apache™ Hadoop™ framework for MapReduce remains difficult to optimize, particularly in large-scale production environments. The vast search space defined by the hundreds of MapReduce configuration parameters and the complex interactions between them makes it time consuming to rely on *manual tuning*. Hence something more is needed. In this paper we evaluate approaches to the automatic tuning of Hadoop MapReduce including ones based on cost-based and machine learning models. We determine that they are inadequate and instead propose a search-based approach called Gunther for Hadoop MapReduce optimization. Gunther uses a Genetic Algorithm which is specially designed to aggressively identify parameter settings that result in near-optimal job execution time. We evaluate Gunther on two types of clusters with different resource characteristics. Our experiments demonstrate that Gunther can obtain near-optimal performance within a small number of trials (<30), outperforming existing auto-tuning solutions and industry recommended configurations. We also describe a methodology for reducing the dimensionality of the auto-tuning problem, further improving search efficiency without sacrificing performance improvement.

Keywords: Hadoop, Genetic Algorithm, Parameter Optimization, Auto-tuning.

1 Introduction

MapReduce is a distributed programming model used to process large datasets across thousands of machines. It has gained much popularity due to its simple yet expressive interface, scalability and fine-grained fault tolerance [5, 9]. Apache™ Hadoop™ [9] is an open-source implementation of the MapReduce model and is widely used for data mining, log processing and machine learning [7, 16, 17, 18, 22]. Hadoop exposes 200+ parameters providing users the flexibility to customize it according to their need [26]. Some parameters have significant performance impact. The major challenge lies in quickly identifying the best parameter settings for a particular application on a given cluster [1, 4, 25].

The common practice is to tune up Hadoop using rule-of-thumb settings published by industry leaders, such as Cloudera and MapR [4, 9, 25], but these recommendations are too general and fail to capture the specific requirements of a given application and resource constraints (i.e., amount of CPU, network and storage) of a given cluster. Additionally, the large parameter space, with its complex inter-dependencies,

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 406–419, 2013.
© Springer-Verlag Berlin Heidelberg 2013

and the sheer scale of many clusters increases the complexity of manual tuning, in which a person repeatedly runs jobs in an attempt to identify the best parameter settings using trial-and-error. An *efficient*, *effective* and *automated* approach to parameter optimization is the only viable solution.

Cost-based auto-tuning is used in database systems [3]. Motivated by this, researchers proposed a cost-based approach for Hadoop MapReduce optimization [12, 13]. However, it is extremely difficult for simple cost-based models to accurately predict the performance of a wide range of Hadoop applications over a wide range of clusters provisioned with different CPU, storage, memory and network technologies. Furthermore, cost-based models are strictly bound to a particular version of a framework and do not evolve with the framework. Machine learning models are another popular approach [3, 15, 24]. In contrast to cost-based models they rely on training sets to "learn" model coefficients and hence are more adaptive and flexible. Unfortunately, our studies show that it is difficult to construct an accurate machine learning model without a large training set involving hundreds or potentially thousands of trials.

In this paper we propose Gunther, a search-based auto-tuner for Hadoop MapReduce that addresses these challenges. It employs a search algorithm that iteratively evaluates the variation in performance of a MapReduce application for different configuration settings and often attains a near-optimal solution. Our method extends to any version of the Hadoop framework and different types of clusters and thus is flexible and adaptive.

We evaluate auto-tuning approaches using the two metrics – (i) *efficiency* or how fast the search can find a good configuration, and (ii) *effectiveness* which measures the performance improvement achieved. We study the performance of Hadoop on a number of clusters and discover that it is a nonlinear and multimodal function of Hadoop's configuration settings. We evaluate search algorithms that are popular for finding global optima on multimodal surfaces and select Genetic Algorithm (GA) as our search strategy [23]. We then optimize GA for the Hadoop auto-tuning problem to strike the right balance between search efficiency and effectiveness. We evaluate the result on two clusters with different resource characteristics for several applications. Our experiments demonstrate that Gunther obtains near-optimal configurations within 30 trials in both types of clusters, and yields better performance improvement than configurations recommended by a cost-based approach and industry rule-of-thumb settings. Studies of workload characteristics show less than 10% of the jobs have runtimes of 5 hours or more [31]. Hence for majority of Hadoop users 30 trials is a small price. For larger jobs tuning time is ameliorated when many users keep running their applications for years.

While Gunther is very effective at identifying near-optimal configurations, its efficiency can be further improved by reducing the dimensionality of the search space by ignoring parameters that have little performance effect. We propose a methodology that uses job counters to classify applications into groups that are sensitive to the same or nearly the same subset of parameters. Once applications are classified we limit the search to the subset and achieve near-optimal performance while reducing the search time.

2 Background and Motivation

2.1 Hadoop and MapReduce

In MapReduce, users need only implement map and reduce functions and the rest is handled by the framework. The Hadoop MapReduce framework takes care of task scheduling,

parallelization, inter-process communication, load balancing, and fault tolerance. Large datasets are partitioned into small blocks by Hadoop Distributed File System (HDFS). Execution proceeds in three phases: map, shuffle, and reduce. Instances of the user-defined map function, or map tasks, process key-value pairs from one data block in HDFS to emit a list of intermediate key-values. These pairs are then operated on by instances of reduce tasks which aggregate them and store them back to HDFS. After the completion of map tasks the intermediate files are copied to the reduce nodes. This comprises the network-intensive shuffle phase. Reduce tasks fetch the key-value pairs from map output files and then sort and merge them before processing them. After processing reduce tasks write the final key-value pairs back to HDFS.

2.2 Motivation of Parameter Tuning

Hadoop is parameterized to permit users to manage the dataflow in different phases. Hadoop versions beyond 0.20 expose 200+ tunable parameters and 10+ parameters affect its performance [4, 9]. Figure 1 shows how the total number of reduce tasks affect sort performance on an 8-node Intel® Xeon® processor E3 cluster. As the number of tasks is increased from the default of 8 to 80, the job execution time improves by a factor of 1.9X. This is because increasing the number of reduce tasks reduces the amount of data per reduce task and alleviates storage contention during shuffling. A strong impact is also observed for memory buffer size for sort (i.e., *io.sort.mb*) in Figure 1. The performance difference between the best and worst settings is 15%. *io.sort.mb* determines when to sort and spill data to storage. It affects both CPU sorting time and storage write time. All these demonstrate that the performance of Hadoop MapReduce applications can be substantially improved by tuning these parameters.

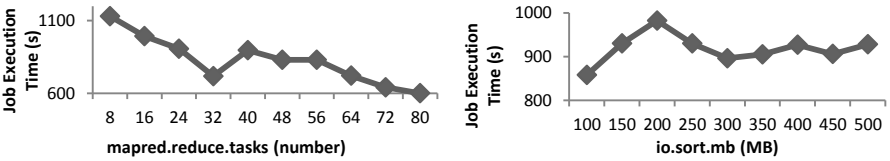
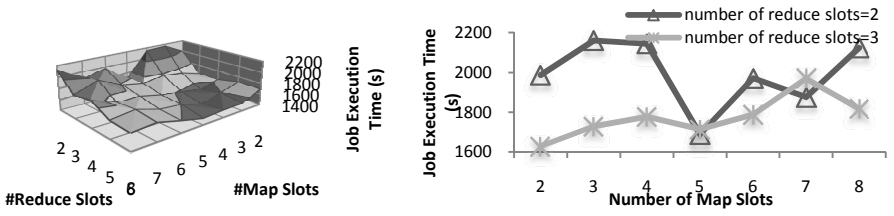


Fig. 1. The performance impact of *mapred.reduce.tasks* and *io.sort.mb*



(a) (b)

Fig. 2. Sort performance for (a) different map and reduce slots; (b) for reduce slots = 2 and 3

2.3 Issues with Manual Tuning

Hadoop parameter optimization requires domain-specific knowledge of the application, the Hadoop framework, and the cluster systems architecture. And *manual tuning* is extremely challenging given explicit and implicit dependencies between different configuration parameters and their effect on the different aspects of the system. Figure 2(a) illustrates a scenario where joint exploration of two parameters, namely the number of map and reduce slots, results in a complicated non-linear surface representing the runtimes of a Sort job. This problem is exacerbated as more parameters are added to the exploration. Figure 2(b) presents the performance as a function of the number of map slots and reduce slots 2 and 3. Clearly, the optimal number of map slots depends on the number of reduce slots. Hence, the parameters need to be examined jointly to locate the global optimal configuration. The manual evaluation of all possible combinations of all performance-related configuration parameters may take months, rendering it impractical. These issues motivate auto-tuning approaches.

3 Approaches to Auto-Tuning

Performance models are used to automatically tune databases and other complex systems [3, 8, 15, 21, 24]. The common approaches involve cost-based or machine learning models. Cost-based models are constructed a priori and calibrated by evaluating the costs of various operations. Machine learning models are used similarly but derived by learning from training sets.

3.1 Cost-Based Models

Cost-based models are built using domain-specific knowledge. In the context of MapReduce, researchers at Duke University recently proposed a Hadoop auto-tuner called Starfish [1, 11, 12, 13]. In Starfish, cost is measured in terms of CPU cycles of different execution phases, such as CPU cost of processing a key-value pair in the map function. To the best of our knowledge, Starfish is the first attempt to address the Hadoop auto-tuning problem. In this subsection, we use it as a case study and reveal its limitations due to model inaccuracy.

Starfish's model uses the average CPU and I/O cost of reads and writes to estimate average map execution time [11]. These costs are measured by profiling a job with a single configuration and the model predicts the same map time as the number of reduce tasks is swept from 64 to 256, as shown in Figure 3, even though the actual time changes considerably.

Additionally, we present task execution information for three configurations in Table 1. The map times are highly variable, with standard deviations of up to ~90%. This complicates task scheduling and could skew the job execution time. The model does not consider these effects, thus introducing errors. The above problems occur because the contention for hardware resources between map and reduce tasks

changes with configurations and this contention largely affects task time. While a more complex cost-based model (e.g., one that sensitizes costs to these effects) may address these issues, it is challenging to build such a model.

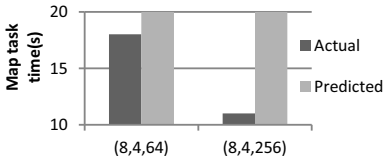


Fig. 3. Map time for sort with 8 map, 4 reduce slots, and 64 or 256 reduce tasks

Table 1. Map task times for three configurations

(map slots, reduce slots, reduce tasks)	Job time(s)	Average map time(s)	Standard deviation of map time(s)
(8, 4, 64)	1847	18	16
(8, 4, 256)	1600	11	8
(6, 6, 384)	1417	9.8	6

The other limitation of cost-based modeling is its inability to adapt as frameworks evolve. Hadoop MapReduce is a relatively new framework and is evolving at a very fast pace. For instance, the next generation framework called YARN [27] has a significantly different architecture. Hence existing cost-based models may not work and need to be redesigned. The same issue arises as clusters evolve with new processors, memory, and storage technologies.

3.2 Machine Learning Models

Machine Learning (ML) models have been proposed in many fields to estimate the performance of complex systems [8, 15, 24]. However, they are impractical for MapReduce auto-tuning as they require large training sets in order to build an accurate model. Our exploration of this method using models like artificial neural network, support vector regression, multiple linear regression and M5 decision tree revealed that more than 200 evaluations are needed to obtain an accuracy of ~90% with five configuration parameters. Since most MapReduce applications involve batch processing with long execution times (tens of minutes to hours), collection of training sets is slow. Although we can use logs as training sets, they typically capture a small number of configurations. This leads to data under-fitting.

4 Gunther: A Search-Based Auto-Tuner

To overcome the inadequacies of cost-based and ML models, we propose a search-based auto-tuner, called Gunther, to optimize configuration settings in Hadoop MapReduce. We perceive auto-tuning as a black-box optimization problem and use search algorithms to solve it. The objective of the search is to evaluate candidate solutions with the stimulus of different parameters to minimize an objective function. The minimization problem is expressed below:

$$f(X^*) = \arg\min_{X_i \in range(i)} f(X_1, X_2, ..., X_D), 1 \le i \le D \quad (1)$$

where $f: X \rightarrow Y$ is the response function, X_i denotes the i^{th} parameter, D is the number of parameter dimensions, $range(i)$ is range of the i^{th} parameter X_i, and X^* is the optimal configuration for which $f(X)$ attains minimum value. In our case, the Hadoop MapReduce configuration parameters define the parameter space and each functional evaluation involves measuring the execution time of a Hadoop job. This approach offers three benefits. First, we overcome modeling inaccuracy since we use job execution times during search. Second, our approach can be used to optimize future MapReduce frameworks. Hence it is more adaptive. Third, our approach does not require the large training set, hence, is more practical.

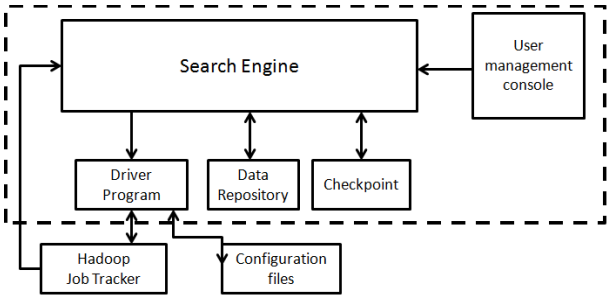


Fig. 4. Gunther Architecture

4.1 Gunther Overview

Gunther's architecture is illustrated in Figure 4. The search algorithm is implemented in the search engine (SE). SE generates a new configuration and asks the driver program to run the application through the JobTracker with the new configuration. After the run is complete, SE writes log files to the repository and analyzes them to obtain execution times. The search terminates after the algorithm meets the convergence criteria or reaches a specified number of trials, and we have designed management console to facilitate progress monitoring.

4.2 Evaluation of Search Algorithm

The effectiveness and efficiency of the search algorithm is critical. To select the right search algorithm, we evaluated both local and global search algorithms on Hadoop performance surfaces using three parameters: map slots, reduce slots and number of reduce tasks. For these, we exhaustively evaluated job execution time. Figure 2 illustrates the response surface of Hadoop sort for two dimensions. We do not show data for other applications but they exhibit similar behavior. From experiments, we observe that Hadoop surfaces are non-linear and multimodal, with many local minima. Local search techniques widely used for unimodal surfaces, such as *Nelder Mead* and *Powell* Search, are inadequate for multimodal surfaces because they easily get stuck at local minima. We found that *Nelder Mead* and *Powell* Search easily settle in local minima on our response surfaces, which are 15% worse than the optimal solution in most cases. This indicates that global search techniques may be more effective.

Global search algorithms are classified into gradient-based search, stochastic search and evolutionary search. Due to the lack of gradient information, we rule out

the gradient based techniques. We identify four stochastic and evolutionary search methods for evaluation: simulated annealing (SA) [20], genetic algorithm (GA) [23], particle swarm optimization (PSO) [19] and recursive random search (RRS) [28]. We used Rastrigin function [28] to perform evaluation. This function generates a multi-modal surface and is challenging due to its large search space and number of local minima. We searched the surface using SA, RRS, GA, and PSO on 10-dimensional surfaces. We observed that SA exhibits the worst performance and GA is the best, followed by PSO. Due to page limits, we do not present detailed results here.

4.3 Applying Genetic Algorithms to the Auto-Tuning of MapReduce

The studies in the previous section motivated us to apply GA to our problem. Algorithm 1 describes a typical GA cycle. The algorithm encodes the potential solutions to a problem as candidates. The solutions are evaluated with a fitness (or objective) function that is tailored to the problem. Candidates with higher fitness scores are deemed better solutions.

Algorithm 1. Traditional GA

Input: P_0 : A randomly selected initial population of size M
Output: C_{Best}
1. $P = P_0$
2. **For** all C_i in P **do** evaluate $\boldsymbol{fitness}(C_i)$
3. $C_{Best} = maximum(\boldsymbol{fitness}(C_i))$
4. **For** N generations (or while search converges) **do**
5. **For** $i = 1,2,\ldots,\frac{M}{2}$ **do**
6. ***Select*** parents from P
7. With probability p_c ***crossover*** $parent_i$ and $parent_{i+1}$ to create candidate $child_i$ and $child_{i+1}$
8. With probability p_m ***mutate*** $child_i$ and $child_{i+1}$
9. Evaluate $\boldsymbol{fitness}(child_i)$ and $\boldsymbol{fitness}(child_{i+1})$
10. ***Update*** P
11. Recalculate C_{Best}

GA begins with an initial population of randomly generated candidates. It evolves the population during each generation by using the genetic operators select, crossover, mutate, and update. A popular selection operator is the *Roulette Wheel* mechanism. In this method, if $\mathrm{fitness}(\mathrm{C_i})$is the fitness of candidate $\mathrm{C_i}$ in the population, its probability of being selected is $\mathrm{P_i} = \frac{\mathrm{fitness}(\mathrm{C_i})}{\sum_{\mathrm{i=1}}^{\mathrm{M}} \mathrm{fitness}(\mathrm{C_i})}$, where M is population size. This allows candidates with good fitness values to have a higher probability of being selected as parents. A crossover function is called with a given probability. It is used to cut the sequence of elements from two chosen parents/candidates and swap them to produce two children/candidates. The mutation function aims to avoid local optima by randomly mutating an element with a given probability. Both crossover and mutation probabilities are input parameters of GA. At the end of each generation, a new population replaces the current population.

In our application, each element g_i represents a Hadoop parameter. A candidate C_i consisting of all parameters, denoted as $C_i = g_1 g_2 .. g_D$, represents a Hadoop job configuration, where D is the number of parameters. The fitness of a candidate is calculated using $\text{fitness}(C_i) = \frac{1}{\text{Job completion time}_i}$. We choose a population of size 2D. GA is a generic search strategy and its operators need to be implemented in an application-specific manner. We define the operators for our problem as follows.

Select Operator

Input: P

Output: parent_i and parent_{i+1}

1. List L = Sort P in decreasing order of $\text{fitness}(C_i)$
2. $\text{Avg}_P = \frac{1}{M}\sum_{j=1}^{M} \text{fitness}(C_j)$
3. if $\text{fitness}(L_k) > \text{Avg}_P$ then $\text{parent}_i = L_k$
4. if $\text{fitness}(L_{k+1}) > \text{Avg}_P$ then $\text{parent}_{i+1} = L_{k+1}$

Select Operator: From experimentation we observe that it is unlikely that two low fitness candidates will produce an offspring with high fitness. This is because, in real clusters, bad performance is often caused by the improper configuration of a few key parameters and these bad settings continue to be inherited. 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. We also observe that good candidates are more likely to produce good offspring/candidates.

The popular Roulette-Wheel selection mechanism has a higher probability of selecting good candidates to be parents than bad ones, but this approach still results in too many job evaluations. Therefore, our selection procedure is more aggressive and deterministically selects good candidates to be parents. The idea is to quickly eliminate poor candidates from the population. To do this, we calculate the mean fitness of the population for each generation and only select parents with fitness scores that exceed the mean.

Update Operator

Input: child_i, child_{i+1}, L

Output: L

1. k = sizeof(L)
2. If $\text{fitness}(\text{child}_i) > L_k$ then $L_k = \text{child}_i$
3. If $\text{fitness}(\text{child}_{i+1}) > L_{k-1}$ then $L_{k-1} = \text{child}_{i+1}$

Update Operator: In GA, the update operator directly replaces parents with their offspring, even if their offspring have lower fitness values. Thus the algorithm does not always retain better solutions, which slows convergence. We modify the update procedure so that child_i and child_{i+1} only replace poorer solutions, i.e., parents whose fitness values are lower than the created candidates in the population. If the offspring

are less fit, they are discarded and the parents survive. This directs the search more efficiently toward better solutions.

Mutate: The mutated value of a parameter is randomly chosen from its range. Since our select aggressively prunes poor regions, we can use an atypically high mutation rate (e.g., p_m=0.1) without impacting convergence. The value of p_m is empirically determined.

Crossover: We use a one-point crossover. A cut point is randomly chosen in each parent's candidate/job configuration and all parameters beyond that point are swapped between the two parents to produce two children. We empirically set crossover probability p_c to be 0.7.

Non-redundancy: Classical GA does not remember prior search and is likely to evaluate some regions more than once. We enhanced our GA to remember search to avoid duplication.

5 Performance Evaluation

5.1 Experimental Setup

We deployed Gunther with Hadoop 0.20.3 on Cluster1 and Cluster2. Each cluster has one master node and 16 slaves. Each node is configured with 16GB memory and one Quad Core Intel Xeon processor E3 with HT enabled. The nodes are interconnected through a 1GbE switch. Cluster1 was designed to be storage bottlenecked and used 3 1TB HDDs for HDFS and intermediate data and a separate 1TB HDD for OS, and Cluster2 was designed to be network bottlenecked, with 3 240GB Intel SSD (520 Series) [14] for HDFS and intermediate data. We selected Sort, Nutch, Kmeans and Terasort from the HiBench suite as our applications. We use rule-of-thumb (RoT) configurations as our baseline. We used Starfish 3.0 in the comparison.

Table 2. Hadoop parameters considered

Parameter Name	Range	Rule-of-Thumb	Description
mapred.tasktracker.map.tasks.maximum	2:12::1	8	Maximum number of map tasks for a node
mapred.tasktracker.reduce.tasks.maximum	2:12::1	4	Maximum number of reduce slots for a node
mapred.reduce.tasks	4N:16N:4N	4N	# reduce tasks in a job. N is the number of nodes
io.sort.mb	100:500::50	100	Size (MB) of buffer to use while sorting map output
mapred.output.compress	True/False	False	Compress the output of the job
mapred.compress.map.output	True/False	False	Compress the output of each map task

Table 2 shows the ranges and recommended values of the six parameters we tuned. Our motivation for exploring these parameters is two-fold. First, these parameters affect the utilization of different resources, such as CPU, memory, storage, and network. By tuning them, we believe we can achieve better balance among these

resources and improve performance. Second, these parameters impact both task- and cluster-level performance. *io.sort.mb* affects task-level performance and the rest of parameters change the distributed system's data flow and have cluster-level performance effects. The second column in the table describes the parameter bounds and step sizes explored (e.g., map slots vary between 2 and 12 in steps of 1).

5.2 Search Effectiveness and Efficiency

In Figure 5a, we compare the best execution time found by Gunther and Starfish with the RoT and best performance for Cluster1. We randomly sampled the space by running a large number of experiments offline for each application. The best performance found is considered best. The figure demonstrates that Gunther achieves near-optimal performance and is more effective than Starfish. Compared to RoT, Gunther yields a 25% performance improvement on average across all workloads. The maximum improvement of 30% is achieved for Terasort. Correspondingly, Starfish achieves an 11% improvement on average, with a maximum improvement of 29%. Figure 5b presents results from Cluster2. Starfish shows no improvement compared to RoT. This is because Starfish assumes sufficient network bandwidth is available, which leads to inaccurate estimation of shuffle time in Cluster2. However, Gunther is able to capture the network bottleneck. As a result, Gunther improves performance by up to 33%, which is close to best. Note that in this cluster RoT is indeed the best configuration for Terasort and there is no opportunity to improve performance. Table 3 enumerates the number of trials it took Gunther to converge. It was able to converge within 30 trials in all cases.

Fig. 5. a) Comparison on Cluster1 b) Comparison on Cluster2

Table 3. Number of trials to reach the best performance

	Sort	Terasort	Nut	Kmeans
Cluster1	20	15	14	12
Cluster2	24	10	21	10

5.3 Comparison with Other Algorithms

Figure 6 shows comparison of Gunther's modified GA with PSO and RRS on Cluster1 and Cluster2 for all four applications. The figure shows the best execution times

achieved for budgets of 20, 30, and 40 trials, normalized to RoT (lower is better). Overall, Gunther achieves higher performance than PSO and RRS and also converges in ~ 20 trials, whereas PSO and RRS often take more than 40 trials. Our results demonstrate that Gunther is more effective and efficient than PSO and RRS.

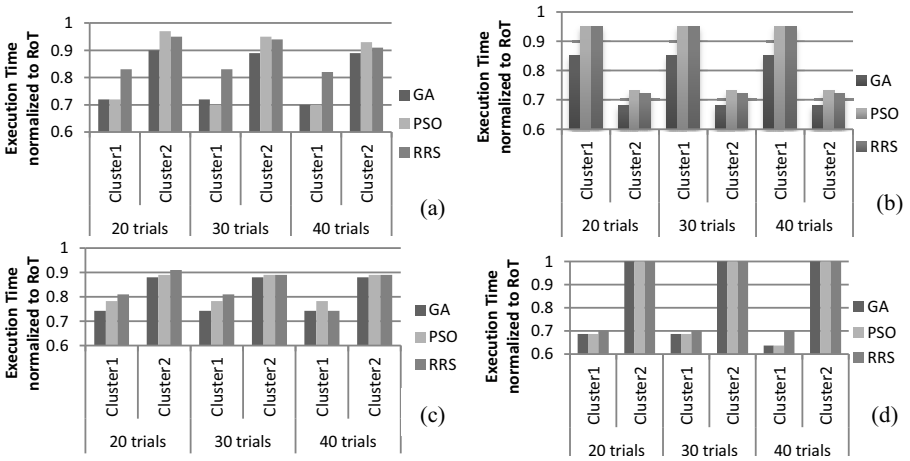


Fig. 6. Search algorithms on clusters for (a) Sort, (b) Nutch, (c) Kmeans, and (d) Terasort benchmarks

6 Using Classification to Improve Search Efficiency

The efficiency of search can be further improved with a minor impact on effectiveness by selectively reducing the dimensionality of the search space. This is possible because some parameters affect performance more than others and the impact depends on what resources are bottlenecked. By profiling an application once with RoT settings, we can rule out parameters that affect resources that are not bottlenecked.

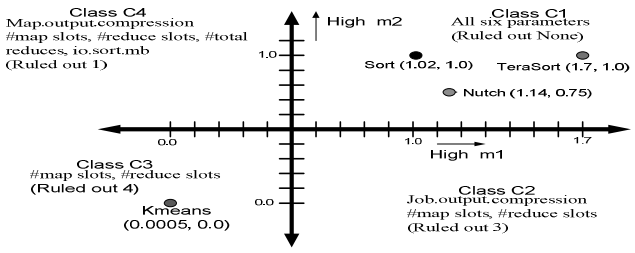


Fig. 7. Classification of applications based on *m1* and *m2*

In this section, we present an application classification methodology based on two metrics that are calculated from five Hadoop job counters. The metric *m1* is defined as the ratio between the count of spilled records and the sum of counts of map input records and reduce input records. Spill records are the number of key-value pairs written from memory to storage during the shuffling phase. This metric represents shuffle intensity. We can rule out *io.sort.mb*, *mapred.compress.map.output*, *mapred.reduce.tasks* when *m1* values are low because they primarily impact shuffle performance. The metric *m2* is defined as the ratio of the count of *hdfs_bytes_written* to *hdfs_bytes_read*. These count the amount of data read and written by map and reduce tasks, respectively. Low *m2* indicates the compression controlled via *mapred.output.compress* will have small performance effects and can be ruled out.

We classify applications into four classes (C1, C2, C3, and C4) and results are shown in Figure 7. The threshold of 0.5 is derived empirically and in the future we intend to automatically determine its value using a clustering algorithm. Sort, Terasort and Nutch are placed into Class C1 with high *m1* and *m2*, indicating that we cannot rule out any parameters. On the other hand, Kmeans is placed into Class C3 with low *m1* and *m2* because it is compute-intensive, with low I/O utilization. This means we can rule out 4 parameters prior to search. Experimental results in Table 4 are supportive. Limiting the Kmeans (Class C3) search to 2 dimensions had a negligible effect on search effectiveness but cut the search time in half. But, when we ruled out the same parameters for Class C1 applications, both the effectiveness and efficiency were impacted. For example, Sort on Cluster1 improved only 3.9% when 4 parameters were rule out compared to 29.5% when no parameters were ruled out. The improvement on Cluster2 was also impacted. We observe similar patterns for Nutch and Terasort.

Interestingly, applications in C1 are more strongly affected by dimensionality reduction on Cluster1 than Cluster2 even though the metrics *m1* and *m2* are the same. This is because our metrics do not reflect differences in dynamic cluster runtime information (e.g., resource utilization). We believe that developing new metrics considering dynamic information will improve the selectivity of classification. Moreover, our classification only rules out parameters and cannot rule them in. A more powerful method would do both. Finally, the current metrics were selected based on domain expertise. In future, we plan to use machine learning-based classifiers (e.g. principal component analysis, etc.) to automate the selection process.

Table 4. Results of dimensionality reduction

Cluster	Workload	2 Dimensions		6 Dimensions	
		Trials	Improvement (%)	Trials	Improvement (%)
Cluster1	Sort	14	3.90	20	29.48
	Kmeans	**7**	**24.77**	**12**	**25**
	TeraSort	14	0	15	30
	Nutch	4	6.77	14	15
Cluster2	Sort	10	7	24	11
	Kmeans	**5**	**11.5**	**12**	**12**
	TeraSort	5	0	10	0
	Nutch	14	25	21	33

7 Other Related Work

7.1 Hadoop Optimization

Hadoop tuning has been studied [1, 4, 11, 12, 13, 22]. The conventional practice is to rely on rules-of-thumb to find good configurations for applications [4]. In contrast to rule-based approaches, Starfish [11, 12, 13] leverages a cost-based model to tune Hadoop applications. Some studies look beyond Hadoop tuning to library extensions and runtime improvements. Manimal [16] performs static analysis of Hadoop programs and deploys optimizations to avoid reads of unneeded data. Panacea [22] proposes a compiler that performs transformations for Hadoop applications to reduce overheads of iterative applications. Twister [7] proposes a new in-memory library to improve the performance of iterative MapReduce applications.

7.2 Auto-Tuning Other Systems

Cost-based, ML and search-based models are used to auto-tune complex systems [3, 6, 8, 15, 21, 24, 28, 29]. Most of the work in database systems uses cost-based models to find the optimal configuration. For instance, IBM DB2 [21] provides an advisor for setting default values for a large number of parameters, which relies on built-in cost models. Similarly, machine learning models are used for auto-tuning many systems. Ganapathi et al. [8] proposed KCCA to derive the relationship between configurations and performance. The search algorithms evaluated in this paper have been used to identify near-optimal configurations for other complex systems. For example, Ye et al. [28] used RRS to tackle network configuration. In addition, Zheng et al. [29] constructed a parameter dependency graph and applied a simplex search method to find good configurations for web services. Duan et al. [6] tuned database parameters by developing adaptive sampling based on a Gaussian process. Zhu et al. [30] used an online learning algorithm to adjust the parameters of applications and optimize performance.

8 Conclusion and Future Work

In this paper, we assessed model-based approaches for Hadoop MapReduce optimization and identified major limitations. Our findings motivated us to propose and implement Gunther, a search-based auto-tuner. We studied several global search algorithms and selected GA as our search strategy. We modified GA to strike the right balance between search efficiency and effectiveness and evaluated the resulting search algorithm on two clusters. Experimental results demonstrate that Gunther achieves near-optimal performance in a small number of trials (<30) and yields better performance improvement than rule-of-thumb settings and a cost-based auto-tuner. We also proposed an application classification method that further improves the search efficiency of Gunther by ruling out less important parameters. Our preliminary results are very encouraging, demonstrating that the number of trials can be reduced by half without sacrificing performance. In the future, we intend to extend our classification method to include both static application characteristics and cluster runtime information.

References

1. Babu, S.: Towards Automatic Optimization of MapReduce Programs. In: SOCC, pp. 137–142 (2010)
2. Beck, A.: A Fast Iterative Shrinkage-Threshold Algorithm for Linear Inverse Problems. In: SIAM (2009)
3. Chaudhuri, S., Narasayya, V.: Self-tuning database systems: a decade of progress. In: VLDB 2007 (2007)
4. Cloudera: 7 tips for Improving MapReduce Performance
5. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI (2004)
6. Duan, S., Thummala, V., Babu, S.: Tuning database configuration parameters with iTuned. In: VLDB 2009 (2009)
7. Ekanayake, J., et al.: Twister: a runtime for iterative mapreduce. In: HPDC (2010)
8. Ganapathi, A., et al.: A case for machine learning to optimize multicore performance. In: HotPar (2009)
9. Hadoop mapreduce, `http://hadoop.apache.org`
10. HiBench, `https://github.com/hibench/HiBench-2`
11. Herodotou, H.: Hadoop Performance Models. Technical report, Duke Univ. (2010)
12. Herodotou, H., et al.: What-if Analysis, and Cost-based Optimization of MapReduce Programs. In: PVLDB (2011)
13. Herodoto, H., et al.: Starfish: A Self-tuning System for Big Data Analytics. In: CIDR (2011)
14. Intel SSD, `http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-ssd.html`
15. Ipek, E., de Supinski, B.R., Schulz, M., McKee, S.A.: An approach to performance prediction for parallel applications. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 196–205. Springer, Heidelberg (2005)
16. Jahani, E., et al.: Automatic Optimization of MapReduce Programs. In: PVLDB (2011)
17. Jiang, D., et al.: The Performance of MapReduce: An In-depth Study. In: PVLDB (2010)
18. Kambatla, K., et al.: Towards optimizing hadoop provisioning in the cloud. In: HotCloud (2009)
19. Kennedy, J., et al.: Particle Swarm Optimization. IEEE ICNN (1995)
20. Kirkpatrick, S., Gelatt, D.C., Vechhi, M.P.: Optimization by simulated annealing. Science (1983)
21. Kwan, S., et al.: Automatic Configuration of IBM DB2 Universal Database. IBM TR (2002)
22. Liu, J., et al.: Panacea: Towards Holistic Optimization of MapReduce Applications. In: CGO 2012 (2012)
23. Mitchell, M.: An Introduction to Genetic Algorithms. The MIT Press (1996)
24. Singer, J., et al.: Garbage collection auto-tuning for java mapreduce on multi-cores. In: ISMM (2011)
25. `http://Vaidya.hadoop.apache.org/mapreduce/docs/r0.21.0/vaidya.html`
26. White, T.: Hadoop: The Definitive Guide. Yahoo Press (2010)
27. YARN, `http://hadoop.apache.org/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html`
28. Ye, T., Kalyanaraman, S.: A Recursive Random Search Algorithm for Large-Scale Network Parameter Configuration. In: SIGMETRICS, pp. 196–205 (2003)
29. Zheng, W., Bianchini, R., Nguyen, T.D.: Automatic Configuration of Internet Services. In: Eurosys 2007 (2007)
30. Zhu, Q., et al.: Automatic tuning of interactive perception applications. UAI (2010)
31. Gridmix3 - Emulating Production Workload for Apache Hadoop: `http://developer.yahoo.com/blogs/hadoop/gridmix3-emulating-production-workload-apache-hadoop-450.html`