Dynamic performance tuning of Hadoop

Kamal Kc and Vincent W. Freeh

{kkc,vwfreeh}@ncsu.edu
Department of Computer Science, North Carolina State University

September 19, 2014

Abstract

Hadoop is an open source mapreduce framework used by a large number of organizations. Its performance is important in increasing the usefulness of large scale data processing applications. Map slot value (MSV) is one of the configuration parameters that influences the resource allocation and the performance of a Hadoop application. MSV determines the number of map tasks that can run concurrently on a node. In this work, we develop an approach to dynamically change MSV during the execution of an application. Our approach converges to the best MSV for all types of applications and it does so with low overhead. Without dynamic approach, determining the best MSV of an application requires a very tedious process of measuring the map completion time for all MSV settings. Our approach also adjusts MSV for applications that may not have a single best MSV throughout their execution. Compared to the peformance of an application when it is using the best MSV, performance of our dynamic approach is within 4.6% with cold start and improves by as much as 5% with warm start.

1 Introduction

Hadoop [3] is an open source implementation of mapreduce framework [12]. Hadoop simplifies processing of large datasets across a large number of machines as a user only needs to implement map and reduce functions, and the framework takes care of all other operations such as creating tasks for each function, parallelizing the tasks, distributing data, and handling machine failures. Hadoop relies on more than 150 configuration parameters to change the allocations of tasks, intermediate buffers, and a large number of internal operations. Many of these changes influence Hadoop's performance, which is critical due to its widespread use in running a variety of applications such as indexing, log analysis, ecommerce, analytics, and machine learning [4]. Using appropriate configuration values can help an application to complete faster, whereas misconfiguration can slowdown an application. Previous works show slowdown up to 2.3 times [17], 1.5 to 4 times [16], and 1.5 times [15]. A common method to determine best configuration values is to manually try several possible values and select the ones that reduce the overall execution time [2]. This process quickly becomes tedious when finding best values for more than one application. Similarly, previous works also attempt to determine best configuration values by extensively profiling an application [14] or profiling an application's map task [17]. As these works show that best configuration values change according to the type of applications, these approaches require statically changing configuration values whenever a different application is run. Thus, it is desirable to have a mechanism that automatically selects the best configuration parameters during the execution of an application. In this work, we describe a mechanism that dynamically tunes the configuration parameter map slot value.

Map slot value (MSV) is a configuration parameter whose misconfiguration can cause significant performance degradation. It is the maximum number of map tasks that run concurrently on a machine. Our observations show that using best MSV results in faster completion of an application, whereas misconfiguration adversely affects performance. Our previous work [17] shows that an application suffers a performance degradation up to 132% when not using the best MSV. A naive way of knowing the best MSV is to profile the completion time for all MSVs. For example, in our experiments, we run each application for ten MSV settings to determine the best one. Out of the ten MSVs, seven are best for at least one application. While this technique finds the best MSV, it is mostly impractical in Hadoop deployments.

Our dynamic approach tunes MSV by using a feedback controller that changes MSV in response to change in resource pressure of a system. Our approach converges to one of the best MSV during the execution of an application. Dynamic tuning does not require any extensive profiling as is required by other static tuning methods [16, 15, 17]. It can dynamically adjust MSV for all types of applications. During an application's execution, feedback on instantaneous system metric values is used to determine whether MSV should be increased or decreased. The metrics are selected such that they can estimate the performance of an application.

In this paper, we present our feedback based approach to dynamically tune MSV. Compared to the profile determined best MSV setting, our approach achieves performance within 4.6% with cold start and improves by as much as 5% with warm start. Additionally, our approach saves significant time by not requiring extensive profiling. In the following sections of the paper, we present the related work, provide background description of Hadoop, describe our design and implementation, and evaluate the results.

2 Related work

Five areas of prior research are related to our work. The first area is Hadoop tuning. Research in this area explores tuning Hadoop using a trained model [27], using tuning rules in conjunction with Hill climbing search to explore configuration parameters [18], using cost based optimization over application profile [15], using prior knowledge of optimal values of other applications [16], and automatically using resource threshold as control [24]. Our work differs from previous tuning efforts because it does not require prior knowledge, tuning rules, or extensive profiling of applications. Additionally, as our work uses dynamic control, it does not rely on predetermined resource usage threshold.

The second related area is feedback based control. Several types of controller are used to adjust different systems. A PI controller is used to tune web server's response content quality by using its server utilization [9], and is also used to set CPU frequency of a machine by using its power consumption [19]. Another related work uses a PID controller to set CPU allocations by using a machine's response time [25]. Our work uses a PID controller, with some modifications, to control MSV by using the resource usage metrics as process variable.

The third related area is resource allocation. This area mostly focuses on resource sharing across multiple jobs by performing allocations based on the resource that is dominant among all the requested resources [13] or by creating a framework that enables an application to specify its resource requirements [23]. Such work ensures fairness across multiple applications where the user specifies resource requirements of each application. In contrast, our work maximizes the system resource utilization to achieve faster completion time for an application.

The fourth related area is optimization of applications by selecting best predefined techniques. Several techniques such as rules [5], program analysis [21], or selecting alternative implementations [26] are used to optimize the performance of an application. These approaches provide an additional method of improving system performance when an application's usage scenarios can be enumerated before execution or when an application can be reconfigured to use alternative implementations.

The fifth related area is workload analysis of Hadoop applications. Work in this area focuses on creating standardized benchmarks [11] or creating qualitative job classes such as small, medium, and large duration jobs [20]. Workload analysis helps to classify jobs. These classifications can be further used to correlate jobs and their performance behavior.

3 Background

Hadoop consists of a jobtracker server which coordinates with multiple tasktracker clients. Jobtracker accepts mapreduce jobs from users and a typical mapreduce job processes large amounts of data. Jobtracker and tasktrackers handle the computation part of the job. The data storage and retrieval is handled by the Hadoop distributed file system (HDFS). HDFS is formed by aggregating storage of each client node. It is managed by a namenode server, which coordinates with multiple datanode clients. The datanodes are responsible for data storage and run alongside tasktrackers in the client nodes. HDFS uses 64MB or larger block sizes.

After the jobtracker accepts an application, it creates a map task for each block of input data. It then assigns the map tasks to tasktrackers. The maximum number of tasks that can be assigned at a time is determined by the MSV setting of the tasktracker. A map task processes a block of data and produces key-value output pairs. The map output is partitioned according to the range of keys. The number of partitions is equal to the number of reduce tasks and a reduce

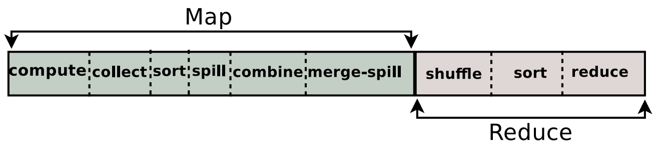


Figure 1: Map and reduce task phases.

task processes one specific partition. The jobtracker assigns reduce tasks based on the reduce slot values. When the reduce task is processing the data, it copies data belonging to the assigned partition from the tasktracker nodes. After reduce task completes, it writes the final output to HDFS.

A map task consists of six phases. These phases as shown in Figure 1 are compute, collect, sort, spill, combine, and merge-spill. In the compute phase, a map task performs the map function computation on each input key-value pair. In the collect phase, the map task stores the processed key-value pairs in a map output buffer. The map output is logically divided into partitions equal to the total number of reduce tasks. In the sort phase, the map task sorts the output key-value pairs of each partition. When the map output buffer is full, the map task spills the content of the buffer to a spill file in the local disk. This is the spill phase. The combine phase is optional and when present, the map task performs a local reduce operation on the output key-value pairs. At the end of the map task, if there are multiple spill files then they are merged together to produce a single map output file. This is the merge-spill phase. Each map task phase is either CPU or IO intensive. The CPU intensive phases are compute, collect, sort, and combine. The IO intensive phases are spill and merge-spill.

YARN [23, 1] rearchitects Hadoop by separating the functions for resource management and mapreduce application into separate modules. Even with this approach, resources are allocated using static configuration parameters. The resource management framework consists of a resourcemanager and multiple nodemanagers. Applications run in YARN as containers, which are provisioned by the resourcemanager. Each nodemanager updates its memory and virtual core [6, 7] limits to the resourcemanager. Using these limits and the memory and virtual core request of an application container, the resourcemanager determines the node in which the container is run. When an application is deployed, it creates an application master that itself runs as a container in one of the slave nodes. The application master requests resourcemanager for containers to run map and reduce tasks. After the resourcemanager assigns nodes to run the containers, the application master coordinates their execution in conjunction with nodemanagers. Containers are freed after they complete map or reduce operations.

As the number of concurrent containers that run in a node is determined by memory and virtual core limit specified in the configuration files, the problem of misconfiguration exists in YARN as well. Misconfiguration may result in fewer or many concurrent containers running at the same time, which is similar to running fewer or many concurrent map tasks and may cause performance problems [17].

4 Design and implementation

Our dynamic tuning system consists of a feedback controller that uses system metric values to dynamically adjust MSV. We describe each part of the design in the following subsections.

4.1 Performance metrics

Performance metrics can be collected from two different sources, which are Hadoop specific counters and the system statistics generated by operating system in `/proc/stat` file [8]. Each of them has advantages as well as disadvantages. Hadoop counters have advantage of being internal to Hadoop, with information available on processed data and the time spent by a task for compute and IO. Counters are suitable for characterizing applications as we have done in our previous work [17]. From our previous work, we also found that counters map to wide ranges of best MSVs. The disadvantage of using counters is the difficulty in correlating them with fine grained best MSV selection. For example, to know the extent of delay due to increased IO pressure, the counters at best can provide information on time taken to perform IO operations of map tasks. But, it requires extensive profiling to know what time duration indicates increased delay due to IO pressure. However, metrics collected from `/proc/stat` have an advantage of providing such fine grained information. *procs_blocked* is a metric provided by `/proc/stat` that shows the number of processes blocked due to IO pressure in the system. This information can then be used to determine the extent of the IO pressure. The disadvantage of using `/proc/stat` is that it requires a system call to read the metric values. Our observation during this work did not show any noticeable difference in map completion time when collecting the metrics from `/proc/stat`. Thus, in this work, we use the metrics collected from `/proc/stat` because their advantages outweigh that of Hadoop counters when using them for dynamic control of MSV.

There are many metrics available in `/proc/stat` [8]. In order to identify which metrics are reliable indicators of good and bad performance, we measure these metric values together with map completion time. We do not take into account network related metrics, as network usage during map computation is only significant when there are a large number of non local map tasks. In this work, we have ensured that all map tasks are scheduled to local data blocks. We also do not account for memory related metrics, as our setup has sufficient memory to accommodate all map tasks for both static and dynamic MSV settings.

Figure 2 shows the relation of the `/proc/stat` metrics with map completion time. Each point in the figures represents a 30-second average of the metric values. Each figure contains measurements for thirteen diverse applications. The map completion time for each point is for the application run in which the sample measurement is taken. A higher map completion time indicates bad performance whereas a smaller time indicates good performance. Ideally, a metric can be used to determine an application's performance if it relates with map completion time such that a metric value can be distinctly mapped to a performance value. From the figures, none of the metrics distinctly map to a performance value. However, metrics that can at least separate good and bad performance can be useful for determining performance variations. For *system* time, *iowait*, *procs_running*, *readrate*, *writerate*, and *timeio* in Figures 2a, 2b, 2c, 2d, 2e, and 2f, there are no separation of good and bad performance values throughout the metrics range. For example, *system* time in Figure 2a does not have any range of values that distinctly maps to either good or bad performance. Its entire range maps to both good and bad performance with wider performance fluctuations occurring for smaller metric values. In contrast to these metrics, *user_cpu*, *procs_blocked*, and *ctxt* have metric ranges which map to only good or a bad performance. The higher values of metric *user_cpu* in Figure 2g maps to good performance, the lower values have ambiguous performance. The higher values of *procs_blocked* in Figure 2h maps to bad perfomance. The higher values of *ctxt* in Figure 2i maps to good performance with a small high value range which also maps to bad performance.

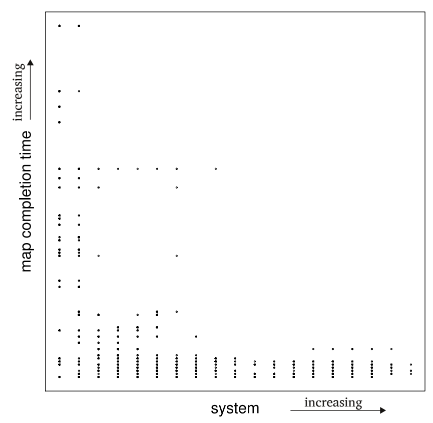
The metric *user_cpu* represents the percentage amount of time spent by CPU in user mode, *procs_blocked* represents the number of processes blocked during IO, and *ctxt* represents the number of context switches performed by the system. Individually, none of the three metrics separate both good and bad performance. But *user_cpu* separates good performance, *procs_blocked* separates bad performance, and *ctxt* in its higher range partly separates bad performance. Therefore, in order to use the performance separation qualities of each of the three metrics, we combine them together to form a single metric called score, which is shown in Equation 1. As the range of each metric value differs, we use coefficients shown in the equation to scale these values such that a 100 represents a highest value for each metric. The highest value of *user_cpu* is 100 and its coefficient is 1. For *procs_blocked*, a value close to the number of CPU contexts in a system can be considered to be the highest value as this means that there are processes blocked on all CPU contexts. We use a system with 64 CPU threads, therefore, the coefficient of *procs_blocked* is $\frac{100}{64} = 1.56$. For systems with different number of CPU threads, the coefficient will be different and can be derived by dividing 100 by the number of CPU threads. The *ctxt* metric is measured per 10 microseconds and requires a coefficient of 5 to scale it to a highest value of 100. The metrics are added or subtracted based on the effect of each metric on the performance. The higher the *user_cpu*, the lower is the map completion time. The higher the *procs_blocked* or *ctxt*, the higher is the map completion time. Thus, when forming the score, *user_cpu* is added and *procs_blocked* and *ctxt* are subtracted. Figure 3 shows the relation between score and map completion time. The figure shows that a lower score indicates a bad performance, whereas a higher score indicates a good performance. We use this score to develop our dynamic tuning method.

$$\text{score} = \mathit{user\_cpu} + (-1.56)\mathit{procs\_blocked} + (-5)\mathit{ctxt} \tag{1}$$

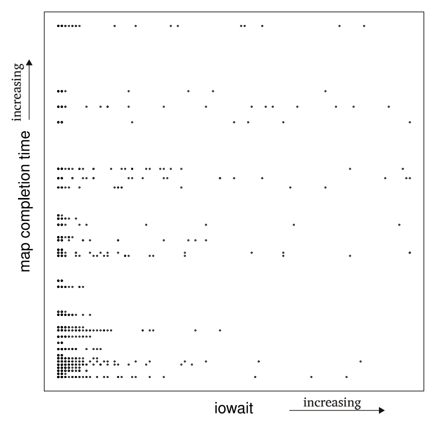
4.2 PID Controller

In this work, we use a PID [22] controller to dynamically change MSV by using the score values. The goal of the PID controller is to set MSV such that the score is maximum for an application. From Figure 3, we know that as score increases the map completion time decreases. Thus, the MSV setting for the highest score would achieve the best map completion time.

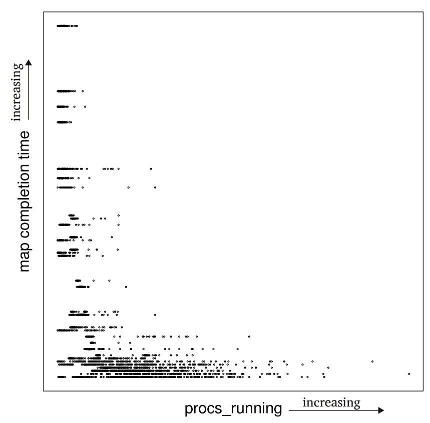
A PID controller relies on error parameter to tune the control parameter, which in our case is MSV. If the maximum achievable score of an application is known, then the error parameter is the difference between the maximum score and



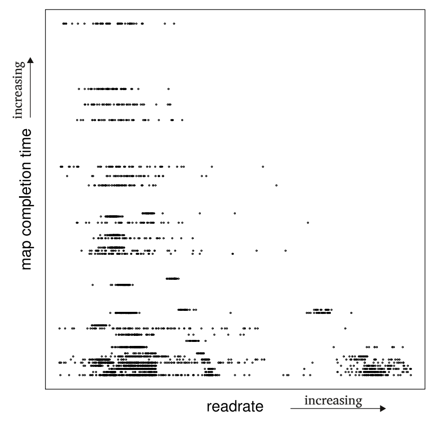
(a) *system* time and map time



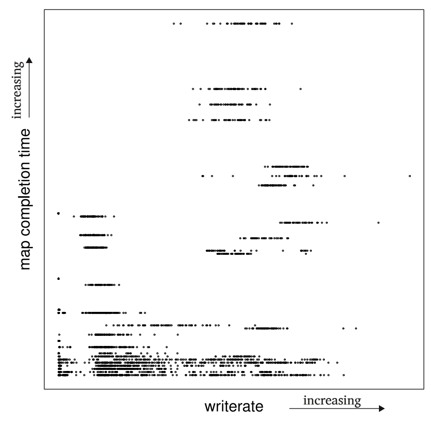
(b) *iowait* time and map time



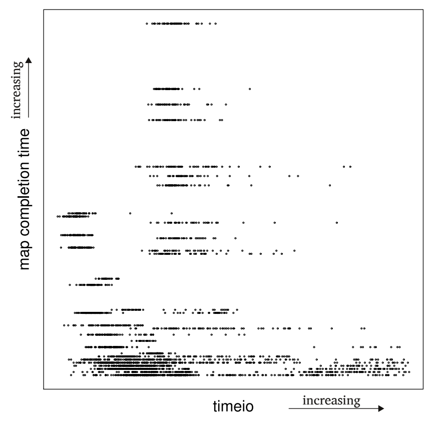
(c) *procs_running* and map time



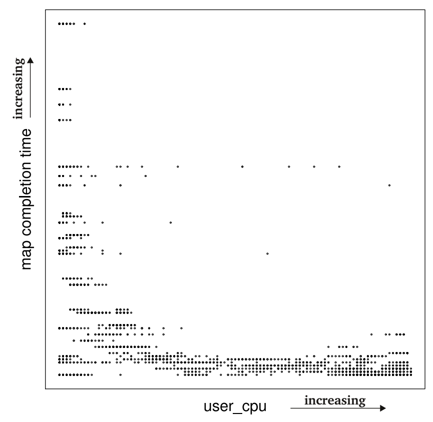
(d) *readrate* and map time



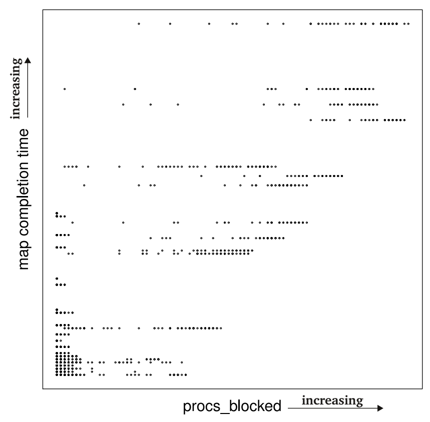
(e) *writerate* and map time



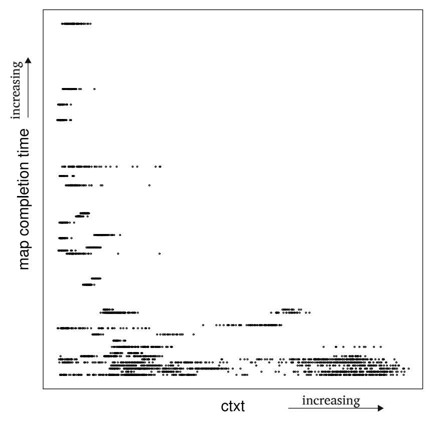
(f) *timeio* and map time



(g) *user_cpu* and map time.



(h) *procs_blocked* and map time.



(i) *ctxt* and map time.

Figure 2: Relation of /proc/stat metrics with map completion time.

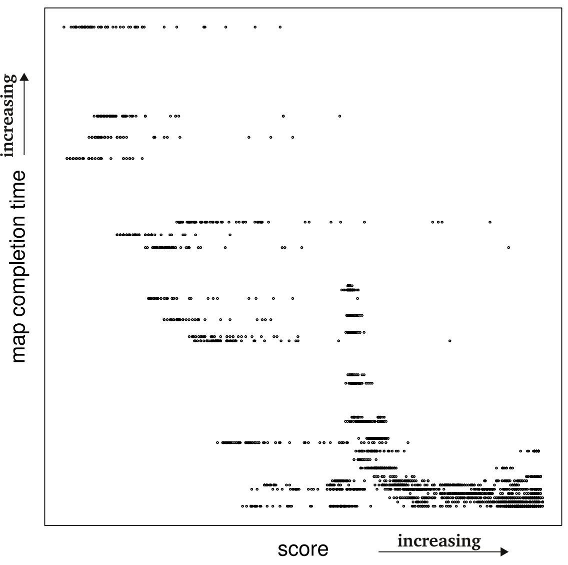


Figure 3: Relation between score and map time.

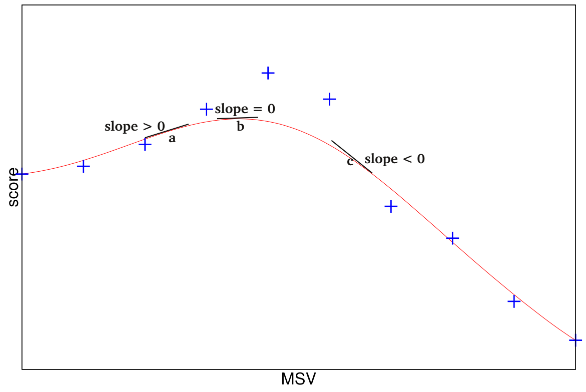


Figure 4: Bezier approximation of MSV against score for a Hadoop application.

the measured score. However, as the maximum score for an application is not known before executing the application, we construct an error parameter such that the error parameter computes to 0 when the score is maximum with best MSV setting, and computes to positive and negative values when the scores are low with lower and higher MSV settings. The error parameter $E(k)$ is shown in Equation 2.

$$E(k) = \frac{score(k) - score(k-1)}{MSV(k) - MSV(k-1)} \tag{2}$$

E(k) is the slope of the curve between score and MSV. Figure 4 shows the Bezier approximation of score against MSV for a Hadoop application. The plot shows that as MSV increases, the score initially increases, peaks and then gradually decreases. This behavior is similar for other applications, with main difference on how early the score peaks. From Figure 3, we know that as score increases, the map completion time decreases. Thus, when $E(k)$ is 0, the score is largest for an application and that corresponds to the smallest map completion time. In this case, the controller maintains the same MSV. When $E(k)$ is positive, it indicates that the score is in increasing trend towards peak score. In this case, the controller increases the MSV. When $E(k)$ is negative, the score is in decreasing trend away from peak score. In this case, the controller decreases the MSV.

The amount by which the MSV changes is given by equation 3. The three components in the equation represent P, I, and D components and K_p, K_i, and K_d are the gain parameters for each component.

$$\begin{aligned} \Delta MSV = & K_p E(k) + \\ & K_i \sum E(k) + K_d(E(k) - E(k-1)) \end{aligned} \tag{3}$$

The gain parameters were tuned by iterating through several values. The selected gain parameters provide quick ramp up time for all Hadoop applications. In order to avoid instability in regions close to the best MSV, we use two techniques to change gain parameters. This is similar to gain scheduling approach used to vary gain parameters for different operating conditions [22]. The first technique lowers gain parameters after the system has achieved a score close to the maximum for a particular application. This maximum score is the one achieved by the application prior to decrease of score due to increasing MSV. Without changing the gain parameters in such scenario, small score changes bring large MSV changes that destabilizes the system. The second technique reduces gain parameters to avoid repetitive overshoot and undershoot of MSV. This occurs for applications with lower best MSV. In such cases, the gain is reduced whenever there is an undershoot of MSV.

There are specific occasions when the controller prediction is not used or the system may not wait for the number of map tasks to reach the PID controller recommended lower MSV count. The first case occurs when the controller recommends a higher MSV during change of score from lower to higher negative values. A negative score represents higher *procs_blocked* or *ctxt*, which occurs during system degradation. Increasing MSV during system degradation is detrimental because once a map task starts, it takes system resources and cannot be terminated until it finishes. The second case occurs when decreasing MSV. As our approach does not immediately terminate or suspend extra map tasks, it has to wait until the extra map tasks finish and the number of running map tasks reduces to the new MSV. As this may result in a long wait time, the controller is invoked earlier if the score increases during the transition to the lower MSV.

Our system is built into tasktracker as a daemon thread and consists of three parts. The first part of our system is the monitoring module. It collects the metrics from `/proc/stat` and invokes the controller every 30 seconds. Time interval of 30 seconds gives steady and reliable measurements due to sufficient activity in the system. Intervals shorter than 30 seconds create measurement jitters and a longer interval is less responsive. There is a scenario when the controller is invoked sooner than 30 seconds. When the PID controller recommends a lower MSV value and the system is waiting for the number of map tasks to decrease, the score of the system may increase before the number of map tasks reaches the lower MSV. This happens because at a higher MSV value, the *procs_blocked* can increase, resulting in a low score; but when the MSV decreases, initially the IO contention decreases as no new processes are introduced due to which more blocked processes get turn to execute and the *procs_blocked* decreases, resulting in a relatively higher score. As the overall goal is to increase the score, the monitoring module invokes the controller early rather than waiting a longer time. This helps to decrease the map completion time. The second part of our system is the PID controller. It is invoked by the monitoring module. Based on the current and previous scores and MSVs, it calculates the next MSV using equation 3. It also performs gain scheduling. The third part of our system is the Java Virtual Machine (JVM) management module. It dynamically instantiates JVMs for map tasks without placing restrictions on the number of concurrent JVMs. In current model of Hadoop, only a fixed number of JVMs can exist at a time.

5 Evaluation

In this section, we describe the experimental setup, analyze the performance of Hadoop applications for static and dynamic MSV settings, and show the system behavior for static and dynamic MSVs.

5.1 Experimental setup

Experiments are performed on a ten node IBM PowerPC cluster. Each node consists of two POWER7 processors with 16 cores and 64 total CPU threads, 124GB RAM, and a 10 Gbps Ethernet network link. Hadoop is configured with one jobtracker and nine tasktrackers. HDFS is configured with one namenode and nine datanodes.

Our experiments use sixteen Hadoop applications. Six of the applications are from the PUMA benchmark [10], seven are customized versions of *terasort*, and three are applications each formed by combining two of the previous thirteen applications. The PUMA applications are *grep, word count, invertedindex, rankedinvertedindex, terasort,* and *termvectorperhost*.

The seven customized versions of *terasort* were used to diversify the applications in terms of per map task CPU utilization (CPU_UTIL) and write IO demand (IO_THRPUT). Table 1 shows CPU_UTIL and IO_THRPUT values of these applications. CPU_UTIL is the percentage of total map task time that is spent on the CPU intensive phases of a map task. IO_THRPUT is the rate at which a map task writes data to its output files and it is measured in megabytes per second (MB/s). The table shows that IO_THRPUT is inversely proportional to CPU_UTIL. Therefore, CPU_UTIL values can infer IO characteristics of an application. In our previous work [17], we found that applications can be grouped based on CPU_UTIL values and the groups can be used to predict best MSV range. Having applications with all ranges of CPU_UTIL (42% to 99%) and IO_THRPUT (10.58 MB/s to 0.01 MB/s) ensures that our results are applicable to all types of Hadoop applications. The customized *terasort* applications were created by adding variable extra busy loops and controlling the amount of output data in the map function. The former helps to vary the computation and the latter helps to change the IO throughput. For the terasort variant *terasort(L10,D100)*, L represents the number of loops and D represents the percentage of input data that is converted to output. In this case, *terasort(L10,D100)* executes 10 extra busy loops for each key-value pair and outputs 100% of the input data. The highest number of busy loops is 500 and the lowest amount of output data is 1%.

We combined multiple applications in order to represent applications that do not have a single best MSV. They are formed by combining applications from three different regions. These regions are based on CPU_UTIL values and each

Applications	CPU_UTIL (%)	IO_THRPUT (MB/s)	Region
terasort	42	10.58	*IO-intensive*
rankedinvertedindex	50	7.93	
terasort(L10,D100)	59	7.84	
word count	63	4.97	*Balanced*
terasort(L30,D100)	70	5.94	
invertedindex	72	5.45	
termvectorperhost	74	5.53	
terasort(L60,D100)	78	4.54	
terasort(L100,D100)	84	3.37	
terasort(L200,D100)	90	2.06	
terasort(L500,D100)	92	0.94	
terasort(L10,D1)	96	0.18	*CPU-intensive*
grep	99	0.01	

Table 1: CPU utilization and IO throughput of an application's map task in the ascending order of CPU_UTIL.

region has a different range of best MSVs [17]. These regions are *IO-intensive*, *Balanced*, and *CPU-intensive* and applications in these regions have respectively low, medium, and high CPU_UTIL values. Each application is categorized into one of the region using its CPU_UTIL value. As applications belonging to a different region have different ranges of best MSVs, combining them results in applications with multiple best MSVs. Table 1 shows that the first two applications with CPU_UTIL from 42% to 59% belong to *IO-intensive* region, next eight with CPU_UTIL from 63% to 92% belong to *Balanced* region, and the last two with CPU_UTIL from 96% to 99% belong to *CPU-intensive* region. The combined applications are *rankedinvertedindexterasort(L200,D100)* formed by combining *rankedinvertedindex* and *terasort(L200,D1000)*, *termvectorperhostterasort(L10,D1)* formed by combining *termvectorperhost* and *terasort(L10,D1)*, and *rankedinvertedindexterasort(L10,D1)* formed by combining *rankedinvertedindex* and *terasort(L10,D1)*.

The PUMA applications use wikipedia dataset. The dataset size of wikipedia is 900GB, which is formed by combining multiple copies of the original 300GB wikipedia PUMA dataset. Terasort and its variants use the data generated by teragen. In order to keep the total experiment time within a reasonable duration, *terasort* variants use three different dataset sizes. 900GB dataset is used by *terasort*, *terasort(L10,D100)*, and *terasort(L10,D1)*. 600GB dataset is used by *terasort(L30,D100)* and *terasort(L60, D100)*. 300GB dataset is used by *terasort(L100,D100)*, *terasort(L200,D100)*, and *terasort(L500,D100)*.

5.2 Performance analysis

Table 2 shows the comparison of normalized map completion time for ten static MSVs and two dynamic MSVs. The best static MSV performance value is 1 and it denotes the shortest completion time of an application for the set of static MSVs used in the experiments. The ideal average performance value of all applications for a static MSV setting is 1. The shortest completion time for each application is shown in parentheses alongside the best MSV performance value of 1. The static MSV is set to values from 16 to 88 with increments of 8. Below 16 and beyond 88, the applications suffer slowdown and those results are omitted.

In Table 2, there are two columns representing dynamic MSV, which are cold and warm dynamic MSVs. The cold dynamic MSV indicates the first run of an application, where the controller is not aware of the right settings for initial MSV and gain parameters. For cold start, we use an initial MSV of 32 and the gain parameters values of $K_p = 10.0$, $K_i = 0.5$, and $K_d = 1.0$. The initial MSV value is equal to half of total CPU threads in the system. This value is appropriate for quick ramp up or ramp down if the application has a higher or a lower best MSV. We also tested with initial MSV of 16 but no discernable difference was observed. The gain parameter values are obtained by tuning the controller, first tuning K_p for P control, then tuning K_p and K_i for PI control, and K_p, K_i, K_d for PID control. While tuning K_p, focus is on reducing the ramp up time. While tuning K_i, focus is on allowing MSV to change even when the score change is minimal. This helps to increase parallelism when the score fluctuation is very small. While tuning K_d, focus is on reducing overshoot and undershoot of MSV. Once K_p, K_i, and K_d are tuned, those values are used for cold starts of all applications. During the application execution, the gain scheduling reduces the parameters by one-third every time there is an overshoot or an

CPU cores per node=16, CPU threads per node=64												
Application	**Static MSV**										**Dynamic**	**Dynamic**
	16	**24**	**32**	**40**	**48**	**56**	**64**	**72**	**80**	**88**	(Cold)	(Warm)
terasort	1.04	1.05	**1(1022s)**	1.15	1.4	1.44	1.49	1.64	1.87	1.82	0.96	0.95
rankedinvertedindex	1.06	**1(1190s)**	1.07	1.15	1.39	1.67	1.61	1.74	1.92	2.13	1.03	0.99
terasort(L10,D100)	1.19	1.13	**1(971s)**	1.22	1.36	1.75	1.57	1.83	2.17	2.29	1.08	1.05
word count	1.51	1.29	1.14	1.06	1.03	**1(1351s)**	1.02	1.02	1.03	1.07	1.08	1.02
terasort(L30,D100)	1.2	1.05	1.04	**1(893s)**	1.03	1.15	1.14	1.25	1.57	1.36	1.08	1.04
invertedindex	1.45	1.2	1.1	1.04	1.02	1.02	**1(1520s)**	1.02	1.01	1.13	1.08	1.03
termvectorperhost	1.41	1.2	1.11	1.04	1.03	1.06	**1(1619s)**	1.02	1.02	1.04	1.05	1.04
terasort(L60,D100)	1.19	1.07	1.05	1.04	1.04	1.01	1.03	**1(1966s)**	1.01	1.16	1.03	1.03
terasort(L100,D100)	1.15	1.05	1.04	1.05	1.03	1.04	1.01	1.01	**1(1997s)**	1.01	1.03	1.02
terasort(L200,D100)	1.16	1.09	1.08	1.07	1.1	1.08	1.1	1.01	**1(1916s)**	1.09	1.08	1.02
terasort(L500,D100)	1.17	1.14	1.14	1.13	1.17	1.16	1.1	1.05	**1(4171s)**	1.04	1.09	1.03
terasort(L10,D1)	1.2	1.05	1.02	**1(390s)**	1.01	1.01	1.09	1.01	1.04	1.05	1.03	1.02
grep	1.52	1.31	1.22	1.11	1.07	1.05	**1(766s)**	1.01	1.01	1.02	1.02	1.01
Average	**1.25**	**1.13**	**1.08**	**1.08**	**1.13**	**1.19**	**1.17**	**1.2**	**1.28**	**1.32**	**1.05**	**1.02**
# of best values	**0**	**1**	**2**	**2**	**0**	**1**	**3**	**1**	**3**	**0**	**-**	**-**
Combined applications												
rankedinvertedindexterasort(L200,D100)	1.06	**1(3258s)**	1.04	1.05	1.13	1.19	1.26	1.22	1.3	1.44	1.02	1
termvectorperhostterasort(L10,D1)	1.08	**1(1589s)**	1.04	1.11	1.33	1.39	1.52	1.6	1.89	1.94	0.98	0.99
rankedinvertedindexterasort(L10,D1)	1.37	1.17	1.08	1.03	1.01	1.01	**1(2010s)**	1.02	1.04	1.05	1.04	1.04
Average	**1.17**	**1.06**	**1.04**	**1.06**	**1.16**	**1.2**	**1.26**	**1.28**	**1.41**	**1.48**	**1.01**	**1.01**
# of best values	**0**	**2**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**-**	**-**
Aggregate	**1.21**	**1.1**	**1.06**	**1.07**	**1.15**	**1.2**	**1.22**	**1.24**	**1.35**	**1.4**	**1.03**	**1.02**
# of best values	**0**	**3**	**2**	**2**	**0**	**1**	**4**	**1**	**3**	**0**	**-**	**-**

Table 2: Normalized performance and the best completion time (in parentheses) for static and dynamic MSVs.

undershoot. Repeated overshoot and undershoot can reduce the parameters close to zero and stagnate the system, therefore the gain parameters have a lower limit of $K_p = 0.3$, $K_i = 0.01$, and $K_d = 0.03$. After an application runs once, all subsequent runs are warm start. The warm dynamic MSV indicates that the controller uses initial MSV and gain parameter settings for an application from the stabilized values of its prior runs.

Table 2 shows the median performance value of three measurements for each MSV setting for each application. The variation of the three measurements for each MSV setting of an application is within 0.08%. The table shows that each application has a best performance value and there is not a single static MSV that is best for all applications. Every application in the table has select best MSV or MSVs (eight applications have more than one MSV within 2% of the best). For example, *terasort* and *word count* have best static MSVs of 32 and 56. Additionally, MSVs 24, 32, 40, 56, 64, 72, and 80 are best for 1, 2, 2, 1, 3, 1, and 3 of the thirteen individual applications, but none are best for all. Two MSVs 32 and 40 have lowest average performance value of 1.08 for the first thirteen applications. But, they have maximum slowdown of 22% for grep and *terasort(L10,D100)* respectively. In contrast, cold start dynamic MSV has an average performance of 1.05. Furthermore, the maximum slowdown caused by cold start dynamic MSV is only 9%, which is for *terasort(L500,D100)*. These numbers improve more for warm dynamic MSV. The average peformance value is 1.02. The maximum slowdown is only 5% for *terasort(L10,D100)*. Additionally, it has 5% and 1% performance gain for *terasort* and *rankedinvertedindex*.

Table 2 also shows that for three combined applications, MSVs 24 and 64 are best for 2 and 1 applications. The lowest average performance value is 1.06 for MSV of 32. Cold and warm dynamic MSVs have average performance value of 1.01. Both have better performance than static MSVs for *termvectorperhostterasort(L10,D1)*.

From table 2, the aggregate performance for all sixteen applications shows that the static MSVs 24, 32, 40, 56, 64, 72, and 80 are best for 3, 2, 2, 1, 4, 1, and 3 applications. MSV 32 has the lowest average performance of 1.06. Cold dynamic MSV has an average performance of 1.03. For two applications, cold dynamic MSV has a performance improvement over the best static MSV. Warm dynamic MSV has an average performance of 1.02 and has a performance improvement over the best static MSV for three applications.

In order to compare the execution progess between static MSVs and dynamic MSVs, Figure 5 shows the execution progress of *rankedinvertedindex* for two static MSVs and the two dynamic MSVs. 24 is the best static MSV for *rankedinvertedindex* and completes faster than cold dynamic MSV by 3%. Warm dynamic MSV instead is 1% faster than the static

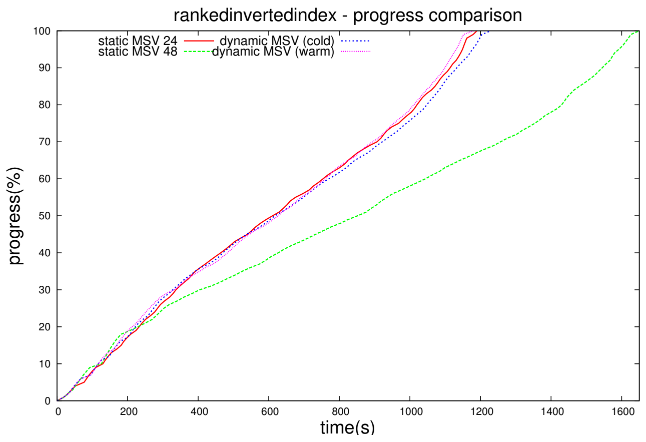


Figure 5: Map completion progress comparison for rankedinvertedindex for static and dynamic MSVs.

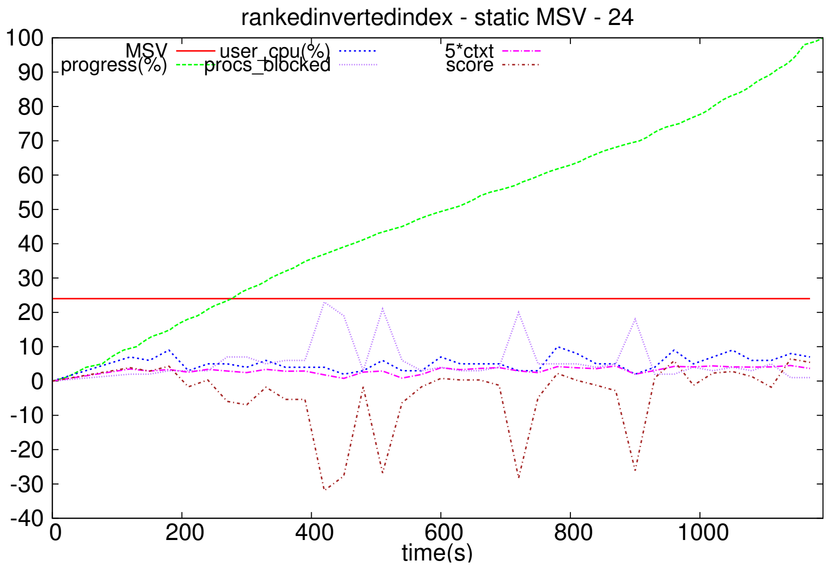


Figure 6: Characteristics of a tasktracker node during map execution of rankedinvertedindex for static MSV 24.

MSV of 24. Static MSV 48 takes the longest time compared to the static 24 and the two dynamic MSVs. Table 2 and Figure 5 show that dynamic approach improves performance when compared to static MSV values.

5.3 System behavior for static MSVs

Figures 6 and 7 show the system behavior of a single tasktracker node during the map execution of *rankedinvertedindex* for static MSV settings. This behavior is similar on other tasktracker nodes as well. Figure 6 shows the behavior for MSV setting of 24, which has the best map completion time for *rankedinvertedindex*. The figure shows that the score remains mostly at low positive value with occasional drop to low negative value due to increase in *procs_blocked*. The metrics *user_cpu* and *ctxt* remain low throughout the application's execution. As *ctxt* value is quite low, in the figure it is scaled by a factor of 5, which is the coefficient used to derive the score. Figure 7 shows the behavior for MSV setting of 48, which has one of the worst running time for *rankedinvertedindex*. The figure shows that the metric *procs_blocked* remains high throughout the execution. The score remains low negative throughout the execution. The metrics *user_cpu* and *ctxt* have low values for this MSV setting as well. As MSV does not change during the application's execution, the score does not recover to a positive or a higher value and hence its performance suffers. The figures show a gap between the metrics and

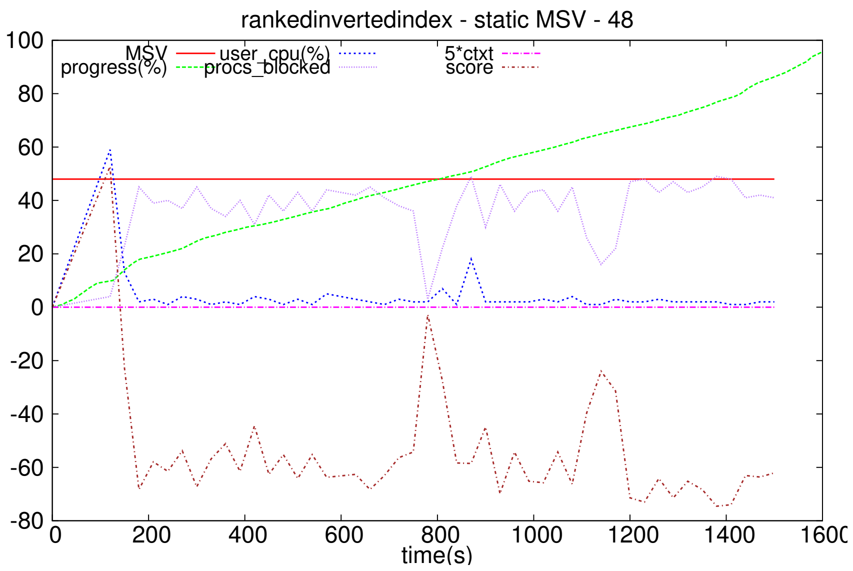


Figure 7: Characteristics of a tasktracker node during map execution of rankedinvertedindex for static MSV 48.

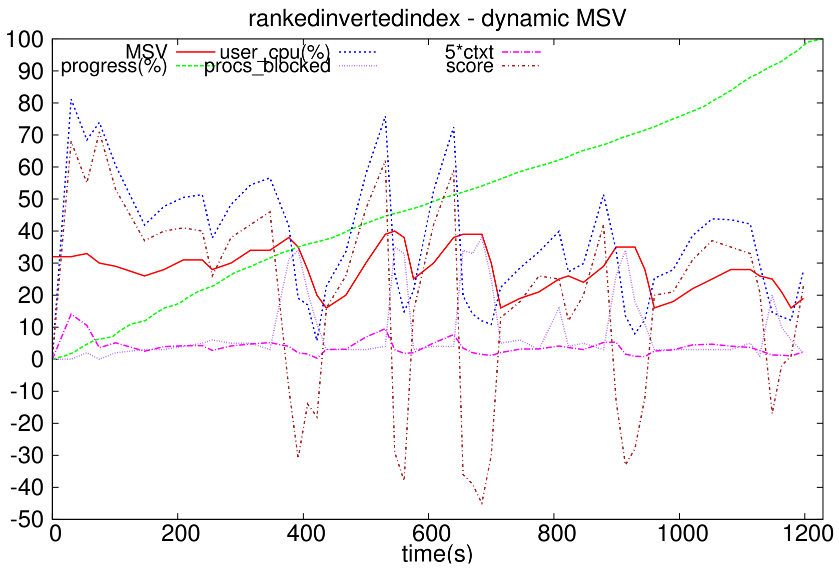


Figure 8: Characteristics of a tasktracker node during map execution of rankedinvertedindex for dynamic MSV.

the 100% progress mark, because in Figure 6, the application completes just before the last measurement sample and in Figure 7, there are a small number of map tasks in other nodes that are waiting to complete. As *rankedinvertedindex* belongs to *IO-intensive* region, its system behavior shows high IO activity indicated by high *procs_blocked* values in the figures.

5.4 System behavior for dynamic MSVs

Figures 8, 9, and 10 show the system behavior of a tasktracker node during the map execution of three different applications with dynamic MSV setting. These figures help to explain the changes in system resource pressure for dynamic MSV values. They also help to explain the response of the controller to change MSV in order to maintain the score as high as possible, which helps to reduce the system resource pressure.

Figure 8 shows the system behavior for *rankedinvertedindex*, which belongs to the *IO-intensive* region and has a rela-

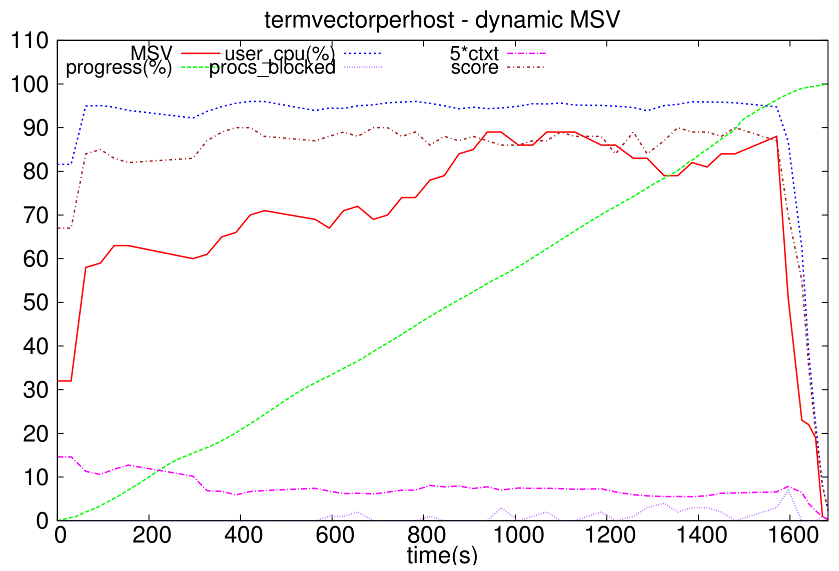


Figure 9: Characteristics of a tasktracker node during map execution of termvectorperhost for dynamic MSV.

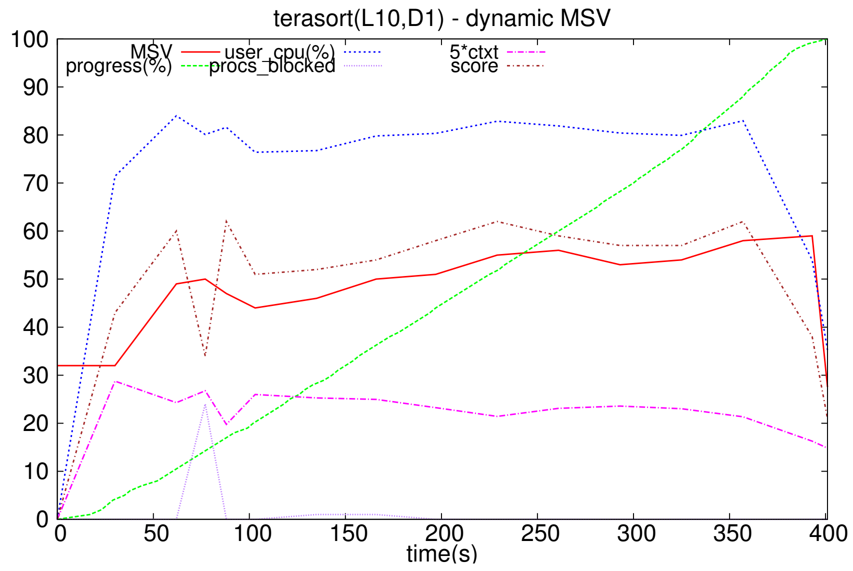


Figure 10: Characteristics of a tasktracker node during map execution of terasort(L10,D1) for dynamic MSV.

tively lower best MSV. As shown in the figure, initially the controller increases the MSV from 32 by a small number. As a result, the score increases and as the score is quite large, the controller performs gain scheduling and reduces the gain parameters. Due to this reason, even as the score decreases at around 50 seconds, the MSV does not change by a large amount. Later when the score decreases drastically at 400 seconds due to high *procs_blocked*, the controller reduces the MSV. This helps to reduce resource pressure, and then the score gradually starts increasing. The controller increases MSV due to the absence of resource pressure. After a while around 600s, the resource pressure increases again, and the MSV drops to low value. The controller controls the MSV setting and does not allow the MSV to exceed values that create resource pressure. We can observe this behavior till the end of the map execution. The rankedinvertedindex test completes at 1229 seconds, just before the next measurement sample, therefore, Figure 8 does not show the decrease in MSV that occurred.

Figure 9 shows the system behavior for *termvectorperhost*, which belongs to *Balanced* region and has a relatively higher best MSV. For this application, the gain scheduling occurs at around 150 seconds and the MSV rises to a value as high as 80.

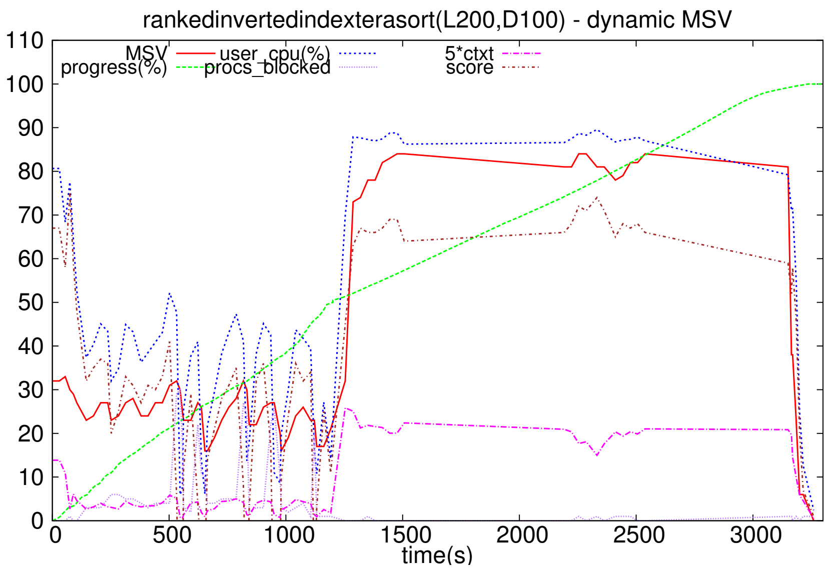


Figure 11: Characteristics of a tasktracker node during map execution of rankedinvertedindexterasort(L200,D100) for dynamic MSV.

During the remainder of the execution, *procs_blocked* and *ctxt* remain low and *user_cpu* remains high. Due to this reason, the score remains high. The resource pressure is fairly low for this application.

Figure 10 shows the system behavior for *terasort(L10,D1)*, which belongs to *CPU-intensive* region and has a relatively moderate best MSV. The figure shows that the MSV remains steady with no significant change in *user_cpu* as well as *procs_blocked*. The MSV varies between 50 and 60. Compared to the previous two applications (Figures 8 and 9), *terasort(L10,D1)* has relatively higher *ctxt* value, almost zero *procs_blocked*, and fairly high *user_cpu*.

5.5 System behavior for combined application

In addition to the single applications, we also analyzed the system behavior for the combined applications. Figure 11 shows the system behavior for *rankedinvertedindexterasort(L200,D100)*. The application runs *rankedinvertedindex* and *terasort(L200,D100)* one after another. At the beginning of the execution, the controller detects a high score and immediately performs gain scheduling. This avoids bigger MSV changes for small changes in the metric values. This behavior is similar to that of *rankedinvertedindex*. The MSV fluctuates between 18 to 28 for 1200 seconds. During this period, there are occasional high *procs_blocked* values, which causes the score to decrease. After 1200 seconds, the terasort(L200,D100) part of the application starts and the controller increases MSV due to the steady increase in score value. Throughout the rest of the map execution, *procs_blocked* remains low, *user_cpu* remains high, and the MSV is tuned to a higher value.

5.6 Overhead

There is little direct overhead of changing MSV because it occurs infrequently compared to the duration of the map execution. The PID controller performs simple calculation, the overhead of metric measurement is unnoticeable as we did not observe any difference in map completion time while measuring metrics during static MSV settings, and the controller only sets a single runtime parameter in Hadoop, which is a store instruction. However, there is an indirect overhead, which is the delay to affect the MSV change. An increase in MSV is effective when the tasktracker receives new tasks at the next heartbeat interval (500ms). A decrease is effective only after extra running tasks are complete. While decreasing, no new task is spawned until the target MSV is achieved.

6 Conclusion

In this paper, we explored a dynamic approach to tune the configuration parameter map slot value (MSV) and improve performance of Hadoop applications. Knowing the best MSV of an application requires extensive profiling, which is quite tedious and mostly impractical in Hadoop deployments. Our approach overcomes the limitation of having to know the best MSV by using a PID controller that changes MSV in response to changing resource pressure. We use a combined value of three different metrics called score to know the extent of resource pressure in the system. Our observation showed that the score can be used to separate good and bad performance of a Hadoop application. Our experiments on sixteen different types of applications show that our approach converges the MSV to the best MSV for all applications. Additionally, it also adjusts MSV for applications that have multiple best MSVs throughout their execution. Compared to the performance of best MSV of an application, which can be known only by extensive profiling, our approach achieves performance improvement as high as 5% with warm start and performance loss of only 4.6% with cold start. Therefore, our approach provides an efficient method of tuning the Hadoop configuration parameter MSV.

References

[1] Apache Hadoop Next Generation MapReduce (YARN). `http://hadoop.apache.org/docs/r2.4.0/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[2] Avoiding common hadoop administration issues. `http://blog.cloudera.com/blog/2010/08/avoiding-common-hadoop-administration-issues`.

[3] Hadoop. `http://hadoop.apache.org`.

[4] Hadoop poweredby. `http://wiki.apache.org/hadoop/PoweredBy`.

[5] Hadoop vaidya. `http://hadoop.apache.org/docs/stable/vaidya.html`.

[6] JIRA YARN-1024 Define a CPU resource(s) unambiguously. `https://issues.apache.org/jira/browse/YARN-1024`.

[7] JIRA YARN-1089 Add YARN compute units alongside virtual cores. `https://issues.apache.org/jira/browse/YARN-1089`.

[8] PROC(5) - Linux Programmer's manual. `http://man7.org/linux/man-pages/man5/proc.5.html`.

[9] T.F. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In *International Workshop on Quality of Service (IWQoS)*, 1999.

[10] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and T. N. Vijaykumar. Puma: Purdue mapreduce benchmarks suite. *Technical Report, Purdue University*, 2012.

[11] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: a cross-industry study of MapReduce workloads. In *International Conference on Very Large Data Bases (VLDB)*, 2012.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design & Implementation (OSDI)*, 2004.

[13] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[14] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. In *International Conference on Very Large Data Bases (VLDB)*, 2011.

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

[16] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Towards optimizing hadoop provisioning in the cloud. In *Conference on Hot topics in cloud computing*. USENIX Association, 2009.

[17] Kamal Kc and Vincent W. Freeh. Tuning Hadoop map slot value using CPU metric. In *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware (BPOE)*, 2014.

[18] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R. Butt, and Nicholas Fuller. MRONLINE: MapReduce Online Performance Tuning. In *International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2014.

[19] Robert J. Minerick, Vincent W. Freeh, and Peter M. Kogge. Dynamic Power Management using Feedback. In *Workshop on Compilers and Operating Systems for Low Power*, 2002.

[20] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. Towards characterizing cloud backend workloads: insights from Google compute clusters. In *SIGMETRICS*, 2010.

[21] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Technical Conference*, 2008.

[22] Dale E. Seborg, Duncan A. Mellichamp, Thomas F. Edgar, and Francis J. Doyle III, editors. *Process Dynamics and Control*. Wiley.

[23] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Symposium on Cloud Computing (SOCC)*, 2013.

[24] Kun Wang, Ben Tan, Juwei Shi, and Bo Yang. Automatic Task Slots Assignment in Hadoop MapReduce. In *Workshop on Architectures and Systems for Big Data (ASBD)*, 2011.

[25] Xiaorui Wang and Yefu Wang. Coordinating Power Control and Performance Management for Virtualized Server Clusters. *IEEE Transactions on Parallel and Distributed Systems,*, 2011.

[26] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 2000.

[27] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. AutoTune: Optimizing Execution Concurrency and Resource Usage in MapReduce Workflows. In *International Conference on Autonomic Computing (ICAC)*, 2013.