Storage Load Control Through Meta-Scheduler Using Predictive Analytics

Kumar Dheenadayalan, V.N. Muralidhara, and G. Srinivasaraghavan

International Institute of Information Technology, Bangalore, India
d.kumar@iiitb.org, {murali,gsr}@iiitb.ac.in

Abstract. The gap between computing capability of servers and storage systems is ever increasing. Genesis of I/O intensive applications capable of generating Gigabytes to Exabytes of data has led to saturation of I/O performance on the storage system. This paper provides an insight on the load controlling capability on the storage system through learning algorithms in a Grid Computing environment. Storage load control driven by meta schedulers and the effects of load control on the popular scheduling schemes of a meta-scheduler are presented here. Random Forest regression is used to predict the current response state of the storage system and Auto Regression is used to forecast the future response behavior. Based on the forecast, time-sharing of I/O intensive jobs is used to take proactive decision and prevent overloading of individual volumes on the storage system. Time-sharing between multiple synthetic and industry specific I/O intensive jobs have shown to have superior total completion time and total flow time compared to traditional approaches like FCFS and Backfilling. Proposed scheme prevented any down time when implemented with a live NetApp storage system.

Keywords: Storage response time · Storage filer · Random forest regression · Time-sharing · Storage load controller

1 Introduction

Large scale systems like the Grid Computing environment [7] has tremendous computing power. Storage technology has been evolving continuously to keep up with the growth in computing power. Even though storage technology has advanced considerably, it still forms the bottleneck in delivering high performance in a distributed environment. As more applications are becoming data intensive, there is continuous growth in data access intensity, imposing greater load on the storage systems. For instance, CERN lab generates around 1 Petabyte (PB) of data per day [3] with peak load reaching up to 10 Gigabytes (GB) per second. This generates huge I/O operations capable of clogging the I/O bandwidth resulting in unresponsiveness and sometimes leading to job failure [5,9].

Grid Computing Environment can handle different type of jobs with different run lengths. With storage being an important resource in Grid Computing system,

it's important to consider the performance of the storage system while scheduling jobs. Storage as a resource has found limited consideration while scheduling a job. Past research was focused on decoupling problems of storage and the performance of a scheduler. Finding the right balance between controlling the load on the filer and maintain good performance of the scheduler is attempted here. Not much of research has gone into controlling I/O load through a scheduler that is actually controlling the sequence of jobs responsible for I/O load generation. Meta-Schedulers are used in large scale Grid computing environments to manage millions of jobs. A meta-scheduler driven storage load controller that maintains the performance at optimal levels and enhancing the productivity of the meta-scheduler and Grid environment on the whole is achieved in the paper. This can eliminate the need to have a storage oriented load balancing mechanism.

A filer is a specialized file server capable of providing fast data access. A modern storage filer supports various specialized hardware to handle high I/O load. Performance of a storage filer depends on a number of parameters. Monitoring a single filer performance parameter like CPU or memory utilization or I/O latency on a filer with multiple individual components is not the wisest approach to find the response state. Identifying the different combination of the parameters that are ideally suited for accurately deciding the response state is a difficult task. Machine learning approach to identify and predict current response state and forecast future response was proven to be successful in [6]. We build on the ideas presented in [6] and provide a simpler solution to identify the current state of the filer and its individual logical entity called volume. The time required by the filer (in its current state) to write benchmark units of data on an individual volume is predicted. We also extend the idea to provide a simplistic solution to forecast the future response time of the volumes in the next t minutes. If the forecasted response time is beyond a threshold $r_{threshold}$, informed time-sharing of I/O intensive jobs is enforced to minimize the load on the individual volumes that are being accessed by jobs.

As the set of candidate jobs for time-sharing is derived from the information provided by the meta-scheduler the need to have a storage oriented load balancing mechanism is eliminated. Total completion time and Total Flow time are couple of key criterion used to measure the success of a schedule [1]. Completion time is defined as the difference between job processing start time and job finish time. Total completion time is measured as the sum of completion times of individual jobs submitted to the Grid environment in a schedule. We define Flow completion time as the difference between the job submit time and the job finish time. Flow completion time includes the time spent by the job waiting for a slot to be scheduled plus the completion time.

Next section talks about the past literature followed by an overview and analysis of the proposed Storage Load Controller in Sect. 3. Modeling and implementation details are discussed in Sect. 4. Model is validated and the results are presented in Sect. 5 followed by the conclusion in Sect. 6.

2 Literature Survey

There is considerable amount of research in the field of storage load balancing in the past [8,12]. But the action of load balancing is derived based on individual or a small subset of storage performance parameters. Data migration based on deterministic analysis of the individual filer performance parameters is one of the popular techniques in load balancing [12,13,15] The performance impact on the storage system during the process of load balancing is not negligible. The need for migration which has been used in traditional approaches of load balancing is completely eliminated in this paper.

NetApp is a popular storage vendor which provides a solution to control the load on a storage filer [5,9]. The process of identifying the filer performance parameters to be monitored and defining the threshold for individual filer performance parameters in [5,9] are left to the administrators. This can be a big problem as the parameter subset and their thresholds vary based on the filer configuration and workload generating the I/O. It also has a hierarchical method of deciding the state of each volume (logical entity of a filer). The process of manually identifying parameters and their associated threshold values is eliminated by using Random Forest Regression with windowing scheme is proposed in this paper. Existing NetApp Solution forces all the jobs accessing a loaded volume to be in *PEND*ing state till the filer parameters go below the thresholds defined by the administrator. This is detrimental to the overall compute efficiency as short jobs are bound to suffer in that scenario. Time-sharing among I/O intensive jobs accessing the filer with high load may help short jobs to get access to the filer for short periods of time. This allows shorter jobs to complete their task instead of starving for I/O bandwidth. [11,19,20] propose data-aware schedulers with emphasis on the data requirement of the jobs. Knowing data requirements of the job is a difficult task and it depends on the information provided by user while launching jobs. Estimating the size of the output generated by the job is not easy. [11] also assumes that the data request sequence is known to the scheduler which is not always true. The proposed solution makes no such assumptions.

Our past work in [6] uses Random Forest, a decision tree based algorithm to identify multiple rules that help the framework to decide the current state of the filer. This offers a major advantage compared to [5,9] and helps in developing an efficient self-learning system. [6] forecasts the future response class by forecasting the individual parameters of a filer and these forecasts are provided to the classifier. The idea of forecasting individual parameters lead to higher error rates as forecasting hundreds of filer parameters comes with individual error rates. The cumulatively error rate of forecasting hundreds of parameters will decrease the overall accuracy of response class. Random Forest Regression [14] is used in the current work instead of Random Forest Classification [2] to get the instantaneous response time of the filer. The predicted response time trends are treated as time series data to forecast the future response times using Auto Regression model. This modification proves to be 7 %

more accurate over a forecasting period of 10 min compared to the forecasting proposed in [6].

Knowing the state of the filer can be used to efficiently share I/O bandwidth between multiple jobs accessing the same volume. Controlling the load on the filer can help in achieving faster completion time leading to earlier execution starts for other jobs. [4] illustrates how time sharing has been useful in enhancing the throughput of the system.

3 Storage Load Controller

Storage Load Controller is an independent component developed to control load and optimize the completion times for all I/O jobs. Load Sharing Facility (LSF) [17] is a popular, commercially available meta-scheduler from International Business Machines (IBM), widely used in various industries, especially in Electronic Design and Automation (EDA) industry. We integrate the Storage Load Controller with LSF meta-scheduler for Grid Computing Environment. Three important states of a job handled by LSF are PEND (new job waiting for a slot/server to begin execution), USUSP (job suspended by user or forced preemption) and RUN state. Storage Load Controller has the capability to interact with filers, compute servers and meta-schedulers. It also has the ability to suspend and resume jobs managed by the meta-scheduler. Suspending a job on LSF will put the processes of the job to USUSP state but the slot occupied by the job is retained. No pending job in the meta-scheduler will be able to get access to a slot occupied by a suspended job. Hence, the Load Controller has to effectively time-share the I/O jobs and move them between USUSP and RUN state to ensure early finish of jobs.

The core idea of Storage Load Controller is to monitor, predict and forecast the load on individual volumes of the filer. When the load on an individual volume is above the threshold defined by the Grid administrator, all the I/O intensive jobs accessing the volume except the oldest job are forced to share I/O bandwidth in a Round Robin fashion. In the proposed scheme none of the CPU intensive jobs are involved in time-sharing. The idea here is to allow the oldest I/O intensive job that entered the Grid setup to run to completion without being hampered by load generated by other jobs. Time-sharing also helps the completion of I/O intensive jobs that have enough compute resource but their progress is hampered by slow I/O response.

We focus our analysis on the time required to process a job at a load of η as defined in Eq. (1) where r_{normal} is the response time at normal load or no load on the filer. The worst case scenario for a Grid setup is when all jobs experience high I/O load when they enter the system. We assume that the jobs follow Poisson's arrival with ρ being the utilization factor of Poisson's distribution. We also assume a hyper-exponential service time distribution. Let n_i be the number of jobs required to breach the threshold factor η for i^{th} volume, vol_i. It is assumed that till a point where a volume has $(n_i - 1)$ jobs accessing it, the load on the

filer will be under control. The worst case processing time T_q of job q at load η is given by Eq. (2).

$$\eta = \frac{r_{threshold}}{r_{normal}} \tag{1}$$

$$T_q = \sum_{r<q} T_r + Remaining\ Processing\ time\ of\ q. \tag{2}$$

The amount of time a job spends in a Round Robin scheme and the amount of time a job gets uninterrupted storage access are given by $\sum_{r<q} T_r$ and $T_q - \sum_{r<q} T_r$ respectively. As soon as all the *'r'* older jobs complete their execution, job_q will get continuous I/O access till completion without being suspended. The fact that one job will go to completion under controlled filer load enhances the completion time for that individual job. [18,21] uses Pollaczek-Khinchin (P-K) Formula to show that expected time for completion of jobs in FCFS scheme is directly proportional to the variability of the service times of the jobs. When coefficient of variability of service time is much greater than 1, Round Robin implementation is superior [18]. Large-scale Grid environments are known to have large variance in service time. We do not use classical Round Robin algorithm because there will be loss of throughput and underutilization of storage resource due to which at least one job will get uninterrupted access.

When the number of I/O jobs accessing a volume are less than n_i, the response time is less than $r_{threshold}$. Hence Storage Load Controller will work like FCFS as the response time is maintained below $r_{threshold}$. If the number of I/O jobs accessing a volume are greater than or equal to n_i, Storage Load Controller will share the I/O subsystem among n - *1* jobs. Let δ be the length of the time slice and $N(\delta)$ be the number of time slices required for job_q to complete in a classical Round Robin Scheduling scheme. Let $l = \sum_{r<q} T_r$ which is the time spent by job_q in the Round Robin scheme of the proposed Storage Load Controller. We calculate the remaining processing time for a job after it becomes the oldest job as $R_q = (N(\delta) * \delta) - l$. If Round Robin scheme was continued after R_q duration, then the number of time slices remaining is given by Eq. (3).

$$R_{RR_q} = \frac{(N(\delta) * \delta) - l}{(1 - \rho)} \tag{3}$$

$$((N(\delta) * \delta) - l) < \frac{(N(\delta) * \delta) - l}{(1 - \rho)} \tag{4}$$

It is clear that $R_q < R_{RR_q}$. Equation (4) holds true whenever there are n_i or more jobs accessing a volume which has necessitated the need for time sharing. As soon as the load factor is below η Eq. (4) fails and hence FCFS is enforced by the Storage Load Controller. The number of I/O intensive jobs required to load a volume depends on multiple factors. n_i will gradually decrease as more volumes are loaded for the same filer. This is because, all physical entities of a filer are shared among the logical entities (vol_i). Hence, $n_i > \ldots > n_j$ for $vol_i \ldots vol_j$, where vol_i is the first volume loaded.

4 Modeling and Implementation

4.1 Data Extractor

The data extractor is responsible for collecting live storage filer performance parameters at a pre-defined interval *'d'*, which will be used for response prediction. Data extractor is also responsible to collect information about all jobs that are executing in the Grid Environment. A job can launch multiple processes and each process can access multiple files. It's important to identify candidate jobs for time-sharing that are actually involved in I/O to cause prolonged load on the filer. A naive way of achieving this is to keep track of all the file descriptors opened by the job and its processes along with the size and time stamps. This will be used to identify the jobs, which are most probable candidates for generating the load on the filer and hence be a candidate to share the I/O bandwidth. A job involved in continuous change in the file size or continuous modification of large files can be a typical candidate. Data extractor communicates directly with meta-schedulers like LSF to gather job related information and store the same in a structured form called the *jobDetails* structure. The key information collected are: *jobID*, *serverName*, *jobStartTime*, *allFileDetails* → [*processIDs*, *fileDescriptors*, *filerName*, *volumeName*, *size*, *accessTime*, *modifyTime*].

allFileDetails is used to store multiple file descriptors which are actively being accessed by the job on the filer. There is no overhead generated by data extractor in this process as all the necessary information except file statistics are already available with LSF. Filer performance parameters are also collected by Data Extractor. Each request to fetch a set of filer performance parameter (V) provides a set of volume parameters (V_{vol_i}) and system (network, protocol, other subsystem and statistics) related parameters (V_{sys}). Volume parameters will be unique to each volume but rest of the system parameters will have the same impact on the load of a filer. Hence, we decompose the data collected into volume parameters and non-volume system parameters. Each row in the data set will be of the form:

$$D_j = V_{sys} + V_{vol_i}$$

The above equation has essentially created x data instances for x volumes through a single data request from the filer as a result of decomposition.

4.2 Response Forecaster

Forecasting the response time is important as rescheduling is based on the forecast of response states for each filer/volume. Identifying the continuous high load period is important for time-sharing among I/O intensive jobs to be effective. The entire data collection and data aggregation phase is explained in great detail in [6]. I/O load is generated through real world or synthetic workload and parallely traning data is collected for a prolonged period of time. Training data essentially contains the set of filer performance parameters while writing benchmark data. When the I/O load is being generated time taken to write benchmark data and the performance parameters recorded during this write operation

forms the dataset. Training data is pre-processed to identify the filer parameters having high correlation with the response times. All parameters above a *correlation_threshold* will be used as features in Random Forest Regression to build a model with smallest Out Of Bag (OOB) error rate [14]. Once the model is successfully generated in the training phase, every new data instance generated in a live filer will be sent as input to the Random Forest Regression model. The model will predict the possible response time, r_{out}.

$$\beta = ws.\frac{60}{d} \tag{5}$$

$$\phi(\beta) = \begin{cases} -\left\lfloor \frac{\beta}{3} \right\rfloor & \text{if } r_{out} < r_{threshold} \\ 1 & \text{if } r_{out} > r_{threshold} \end{cases} \tag{6}$$

A high response time threshold, $r_{threshold}$, will be used to predict a positive value indicating high response time and a negative penalty if low response is predicted as shown by the penalty function, $\phi(\beta)$. The values returned from the penalty function, $\phi(\beta)$ is stored in a separate array for each volume, which represents the window (w) with window size β. ws is the number of minutes to be considered by the moving window. Any negative class prediction will have its impact for 33 % of β instances as indicated by the penalty function. The value of 33 % was arrived purely on the basis of experience and there is no theoretical explanation for the same. The decay function defined by $\psi(i)$ makes sure that the most recent filer load is given higher preference. With every new instance, the older negative predictions decay by a factor of 1. It is assured that the negative prediction loses its weight after $\frac{\beta}{3}$ data instances. A decision on the overall state of the volume can be concluded using the function (8).

$$\psi(i) = \begin{cases} \psi(i-1) + 1 & \text{if } \psi(i-1) < 0 \\ \psi(i-1) & \text{otherwise} \end{cases} \tag{7}$$

$$S(\alpha, \beta) = \begin{cases} 1 & \text{if } \frac{\sum_{i=n}^{n-\beta} w_i}{\beta} \geq \alpha \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

$S(\alpha, \beta)$ evaluates to see if at least α of the past β data instances of a volume has high load predicted by $\phi(\beta)$. If the function $S(\alpha, \beta)$ returns a 1, the algorithm concludes that the volume is loaded for the past ws minutes. Response time (r_{out}) predicted by the model is treated as a time series data. Every minute, a short term response time forecast is carried out using *AR(q)* autoregression model. The duration of the forecast t, should be of the order of few minutes as this will determine the number of minutes a suspended job will wait before a rescheduling cycle (suspension/resumption of jobs) might restart. One of the criteria used by the meta-scheduler to choose the best machine available to run a job is the load on each server for 15 min (r15m). The windowing scheme helps to replicate the

same for *ws* minutes by considering the current response behavior along with the recent response trend to conclude the load on the filer.

4.3 Job State Handler

Job state handler manages the state changes of all the jobs that have started their execution and might be causing the load on the individual volumes of a filer. A virtual queue is created for each individual volume of a filer to provide a time-sharing system for all jobs accessing a loaded volume/filer. Each job will have multiple processes associated with it and any of these processes can be involved in some I/O operation. *'jobDetail'* structure is extracted from Data Extractor to build a structure called *'runList'*. A *'runList'* structure per volume helps maintain details of all jobs actively accessing or generating load on the volume. *runList* is a hashtable data structure with filerName:volumeName as key and a reference to stack data structure as the value. Each element of the stack points to *jobDetail* structure. This stack maintains the list of jobs that are in RUN state and accessing the volume to which the *runList* is associated. Stack data structure is used to pick the most recent job (execution start time) as a possible candidate for suspension.

List of suspended jobs are maintained in a separate hash called *suspendList*. The *suspendList* structure is similar to *runList* except for the fact that a Queue data structure is maintained for each filerName:volumeName key. Using queue data structure facilitates implementing different SLA based priority schemes for resuming jobs. The criterion for selecting the jobs for time-sharing by Storage Load Controller can vary based on the understanding of the workload and the Grid setup. Possible checks to identify if the running job is a candidate for rescheduling are: (1) If the file was accessed or modified in the last 60 s to ensure that we don't choose a job for suspension that is not generating any I/O on the volume. (2) A check on the size of the file to ensure that short jobs accessing files of size less than ω can be ignored. (3) The number of open files is also important (Load can also be generated when millions of small files are accessed creating huge number of metadata operations [10]). (4) If the file size is not varying or not being accessed continuously, then jobs associated with such files are ignored. Every volume is monitored and $S(\alpha, \beta)$ is evaluated at the beginning of a rescheduling cycle. If $S(\alpha, \beta)$ returns 1 for any volume, then a job is popped from the stack in *runlist*, pushed onto the queue of the *suspendList* and a suspension request for this job is sent to the meta-scheduler. This ensures that the oldest job that started accessing the volume will never be suspended. If $S(\alpha, \beta)$ returns 0 for any volume, then a job from *suspendList* is pulled and pushed onto the stack of the *runlist* and resume request is sent to the meta-scheduler. It must be noted that, suspension is called only if there is more than one job associated with each volume.

If any job job_q is suspended for accessing volume vol_i, then no job related to vol_i will be resumed in the same rescheduling cycle. It helps to see the effect of suspending job_q. Once the rescheduling cycle is completed, sleep signal is issued for a time interval identified by $\Psi(r_{forecast_{ij}})$.

$$\Psi(r_{forecast_{ij}}) = \begin{cases} t & \text{if } \exists r_{forecast_{ij}} \geq r_{threshold}, \forall i, j \\ 1 & \text{otherwise} \end{cases} \quad (9)$$

If the forecast for any vol_i is high for next t minutes, then the rescheduling cycle will start after t minutes. If the forecast indicates that the volume load for any volume is going to reduce over a period of next t minutes, then the rescheduling cycle will start after 1 min. $\Psi(r_{forecast_{ij}})$ makes sure that the suspended jobs don't spend their time in suspended state even after the load on the volume has come down.

5 Results

Storage Load Controller was developed using C programming language for job state handler, PERL for Data Extractor and R statistical package for Response Forecaster. The test setup includes a NetApp ONTAP 8 filer with Dual 2.8 GHz Intel P4 Xeon MP processor, 2 MB L3 cache, 8 GB of main memory and 512 MB NVRAM. Three servers with quad core processors and 4 GB of RAM as execution hosts. One single core machine is used to run the proposed Storage Load Controller. LSF version 9.1 was used as the meta-scheduler with its default FCFS scheduling algorithm enabled. A new queue was defined to test Backfilling algorithm. Total of 12 slots were configured with each core taking up one slot. We present the results obtained by executing two different workloads with/without Storage Load Controller in action. DS1, DS2 and DS3 are the identifiers for the datasets used in our experiments. DS1 consists of 6 jobs having 2 CPU intensive jobs and 4 I/O intensive jobs. Table 1 gives details of the type of jobs in each dataset. The ratio of CPU intensive versus I/O intensive jobs was chosen based on the trend observed in EDA industry workload. I/O intensive jobs had a combination of EDA workload (20 jobs), metadata intensive jobs generated using PostMark (12 jobs) [10], I/O benchmark applications like IOZONE (16 jobs) [16]. DS2 follows Poisson's arrival distribution with hyper-exponential service time distribution. DS3 was generated by using DS2 as the population and random sampling was carried out with no bias towards any type of job.

Different types of I/O operations were tested specially using the industry workload and IOZONE tool. PostMark was used to test the behaviour of the storage filer towards jobs working with 10,000 to 10 million short files of less than 256 KB. The amount of data by non-PostMark jobs ranged from 10 GB to 100 GB each. Storage Load Controller is combined with FCFS and Backfilling separately and we compare the results in Table 1. The values of various parameters set for the experiments carried out are: $d \rightarrow 10$ s; $r_{threshold} \rightarrow 60$ s; $\beta \rightarrow 18$; $ws \rightarrow 3$ min; $\alpha \rightarrow 0.75$; $t \rightarrow 3$ min; $p \rightarrow 1$ GB; $\omega \rightarrow 4000$ MB; The key parameters that have a major impact on the success of Storage Load Controller are the values of $r_{threshold}$ and t. Setting a low value for $r_{threshold}$ and t will force frequent job suspensions which is detrimental to the performance of the Storage Load Controller. DS2 considered in the current experiment were run with four different values of t. For $t = 1$ and 5 min, Storage Load Controller performed

Table 1. Performance Measure (unit - hh:mm:ss)

Dataset ID	Size	CPU Jobs	I/O Jobs	Prediction (%) for t mins	**Total Completion Time**		**Total Flow Time**	
					FCFS	**Controller**	**FCFS**	**Controller**
DS1	6	2	4	94.2 %	15:35:27	12:51:16	15:35:34	12:51:23
DS2	152	104	48	92.0 %	256:06:14	235:57:28	298:28:12	274:24:14
DS3	21	13	8	93.4 %	63:26:42	57:58:39	67:11:32	61:03:05
					Backfill	**Controller**	**Backfill**	**Controller**
DS2	152	104	48	91.1 %	314:12:18	295:16:17	345:33:11	323:02:10
DS3	21	13	8	92.3 %	65:12:12	60:14:55	69:03:22	62:33:54

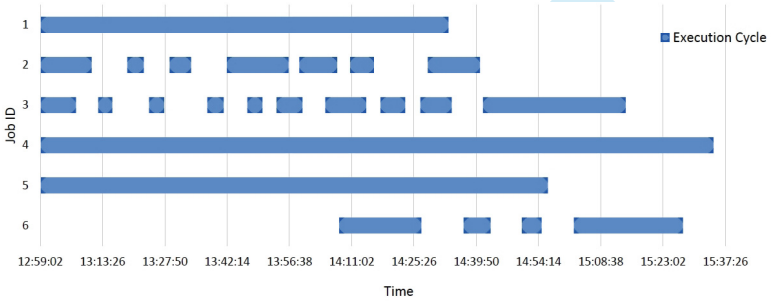

Fig. 1. Gantt chart for time-sharing process

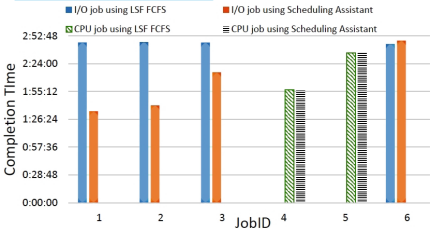

Fig. 2. Completion Time for LSF FCFS with/without Storage Load Controller

worse than LSF scheduling by 2.4 % and 6.4 % respectively. For $t = 2$ and 3 min, Storage Load Controller performed better than LSF scheduling by 5.7 % and 7.8 % respectively. t should be set to high values (2 to 5 min) when I/O intensive jobs are expected to have a runtime of few hours. Short I/O intensive jobs might run to completion with limited effect on the completion time of the job.

To show the importance of time-sharing, a Gantt chart is presented in Fig. 1. It shows the suspension and resume cycles of individual jobs which share I/O bandwidth among themselves. Jobs 4 and 5 are CPU intensive jobs, which do not access any files on the filer. Jobs 1–3 and 6 are I/O intensive jobs, involved in read, write, reread and rewrite operations of various file sizes ranging from 10 GB to 100 GB. Job 1 is the first I/O intensive job to get a slot and it executes

to completion because start time is used as the measure of priority. Other jobs are time shared till they become the oldest running jobs accessing the volume on the filer. Other forms of priority can also be used to change the suspension order based on the SLA that needs to be implemented. Figure 2 shows the completion time comparison of FCFS with/without Storage Load Controller. It is evident that Storage Load Controller affects no CPU intensive job but completion of I/O intensive jobs has improved.

Table 1 shows the performance comparison in terms of Total Completion Time and Total Flow Time for FCFS, Backfill, FCFS with Load Controller and Backfill with Load Controller. Early completion of I/O jobs has a direct effect on the total flow time of all jobs. Storage Load Controller is able to complete the tasks much earlier than FCFS/Backfill scheme. As more I/O jobs complete early, it enables early start for other waiting jobs. For DS2, more than 20+ h of compute time is gained by Storage Load Controller when compared to FCFS and an average gain of 25 min per I/O intensive job. 18+ h of compute time gain with an average gain of 22 min was observed for the same dataset was observed for Backfilling algorithm with Storage Load Controller. Storage Load Controller doesn't have any dependency on the order of job submission. As the framework tries to control the load generated by running jobs, we continue to see improvements in the job completion times and flow times. Since we have specific checks in Job State Handler to omit CPU intensive and small I/O jobs, the system is able to take all CPU intensive jobs to completion without affecting other I/O intensive jobs.

6 Conclusion

The major objective of maintaining the load on the storage system is achieved through Storage Load Controller. Response time to write benchmark data is kept under $r_{threshold}$ throughout the schedule. This leads to faster overall completion times and hence the faster flow times. Storage Load Controller being at a level above the meta-scheduler can be integrated with any meta-scheduler that is using a variety of algorithms. Our work presents the results for FCFS and Backfill but the same can be extended to any scheduling algorithm. Order of job suspension can be improved for Backfilling as the runtime is provided in advance which will be part of our future work. The Suspend and Resume order in the Job State Handler module can be changed by implementing various priority schemes which gives the framework a better chance to succeed in various environments.

References

1. Avrahami, N., Azar, Y.: Minimizing total flow time and total completion time with immediate dispatching. Algorithmica **47**(3), 253–268 (2007)
2. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
3. CERN: European Laboratory for Particle Physics (2014). http://home.web.cern.ch/about/computing. Accessed 30 September 2014

4. Chen, J., Zhou, B.B., Wang, C., Lu, P., Wang, P., Zomaya, A.: Throughput enhancement through selective time sharing and dynamic grouping. In: 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS), pp. 1183–1192 (2013)
5. Choudhury, B.R.: IBM Platform Load Sharing Facility (LSF) Integration with Netapp Storage. Technical report (2013)
6. Dheenadayalan, K., Muralidhara, V., Datla, P., Srinivasaraghavan, G., Shah, M.: Premonition of storage response class using skyline ranked ensemble method. In: 2014 21st International Conference on High Performance Computing (HiPC), pp. 1–10, December 2014
7. Foster, I., Kesselman, C.: The grid: blueprint for a new computing infrastructure. Morgan Kaufmann Publishers Inc., San Francisco (1999)
8. Gulati, A., Kumar, C., Ahmad, I., Kumar, K.: BASIL: Automated IO load balancing across storage devices. In: Proceedings of the 8th USENIX Conference on File and Storage Technologies, FAST 2010, p. 13. USENIX Association, Berkeley (2010)
9. Hameed, S.: Integrating lsf storage-aware plug-in with operations manager. Technical report (2011)
10. Katcher, J.: PostMark: A New File System Benchmark. Technical report (1997)
11. Kosar, T.: A new paradigm in data intensive computing: stork and the data-aware schedulers. In: 2006 IEEE Challenges of Large Applications in Distributed Environments, pp. 5–12 (2006)
12. Kunkle, D., Schindler, J.: A load balancing framework for clustered storage systems. In: Sadayappan, P., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2008. LNCS, vol. 5374, pp. 57–72. Springer, Heidelberg (2008)
13. Liang, H., Faner, M., Ming, H.: A dynamic load balancing system based on data migration. In: Proceedings of the 8th International Conference on Computer Supported Cooperative Work in Design, vol. 1, pp. 493–499, May 2004
14. Liaw, A., Wiener, M.: Classification and regression by randomforest. R News **2**(3), 18–22 (2002)
15. Mondal, A., Goda, K., Kitsuregawa, M.: Effective load-balancing via migration and replication in spatial grids. In: Mařík, V., Štěpánková, O., Retschitzegger, W. (eds.) DEXA 2003. LNCS, vol. 2736, pp. 202–211. Springer, Heidelberg (2003)
16. Norcott, W., Capps, D.: IOzone file system benchmark. Technical report (2006)
17. Quintero, D., Denham, S., Garcia da Silva, R., Ortiz, A., Guedes Pinto, A., Sasaki, A., Tucker, R., Wong, J., Ramos, E.: IBM Platform Computing Solutions (IBM Redbooks). IBM Press (2012)
18. Thompson, S., Lipsky, L., Tasneem, S., Zhang, F.: Analysis of round-robin implementations of processor sharing, including overhead. In: Eighth IEEE International Symposium on Network Computing and Applications, NCA 2009, pp. 60–65 (2009)
19. Venkataraman, S., Panda, A., Ananthanarayanan, G., Franklin, M.J., Stoica, I.: The power of choice in data-aware cluster scheduling. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI 2014, pp. 301–316. USENIX Association, Berkeley (2014)
20. Wei, X., Li, W.W., Tatebe, O., Xu, G., Hu, L., Ju, J.: Implementing data aware scheduling in Gfarm(r) using LSF^{TM} scheduler plugin mechanism. In: Arabnia, H.R., Ni, J. (eds.) GCA, pp. 3–10. CSREA Press (2005)
21. Zhang, F., Tasneem, S., Lipsky, L., Thompson, S.: Analysis of round-robin variants: favoring newly arrived jobs. In: Proceedings of the 2009 Spring Simulation Multiconference, SpringSim 2009 (2009)