Self-tuning Filers — Overload Prediction and Preventive Tuning Using Pruned Random Forest

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

International Institute of Information Technology, Bangalore, India
d.kumar@iiitb.org

Abstract. The holy-grail of large complex storage systems in enterprises today is for these systems to be self-governing. We propose a self-tuning scheme for large storage *filers*, on which very little work has been done in the past. Our system uses the performance counters generated by a filer to assess its health in real-time and modify the workload and/or tune the system parameters for optimizing the operational metrics. We use a Pruned Random Forest based solution to predict overload in real-time — the model is run on every snapshot of counter values. Large number of trees in a random forest model has an immediate adverse effect on the time to take a decision. A large random forest is therefore not viable in a real-time scenario. Our solution uses a pruned random forest that performs as well as the original forest. A saliency analysis is carried out to identify components of the system that require tuning in case an overload situation is predicted. This allows us to initiate some 'action' on the bottleneck components. The 'action' we have explored in our experiments is 'throttling' the bottleneck component to prevent overload situations.

Keywords: Random forest · Pruning · Feature ordering · Storage · Filer · Self-tuning · Storage load

1 Introduction

Large-scale cluster-based storage systems [9,23,25] form an important part of any distributed infrastructure. This is true, because the data stored in these systems is often critical and these account for a large chunk of the overall infrastructure acquisition and maintenance cost. These systems typically take care of almost all the data storage needs of an enterprise and are expected to deliver consistent I/O performance across varied workload demands by a wide spectrum of data intensive applications. Traditionally these systems have been tuned based on workload characteristics that are assumed to be known in advance and predictable. Unfortunately, such assumptions do not hold in large organizations. We therefore focus on using the live system counters that are

generated routinely in most storage systems and use them effectively to periodically evaluate and suggest reconfiguration. Counters in general can provide information on the performance of the system at various levels. These counters are critical for an accurate analysis of system behavior and tuning the system if required.

Research on self-organizing storage in the past [8] has largely focused on scaling storage systems dynamically in response to workloads. However, online optimization or self-tuning of storage systems based on system counter data has received little attention. A reasonably generic way to make a complex system self-sustaining in terms of performance and making it relatively immune to changing workload patterns is to enable dynamic reconfiguration/tuning based on the performance counters collected from within the system. Large scale filers are an assembly of a number of *filer components*/storage objects (physical, logical, protocol, software components). Some key components are buffer, cache, NVRAM, cpu, volume, aggregate, disk etc. Complex storage system architectures can include tens to hundreds of such interconnected components working together to provide high-levels of performance expected from such systems. Each component typically generates tens to hundreds of performance counters, each measuring different aspect of performance of the component. The total number of performance counters/features, across all components could be more than 100,000. When collected at intervals of 5–10 s, this counter data forms a rich source of information for system diagnostics and tuning.

Our contributions in this paper is to develop a self-tuning storage system. Towards this, we assume that the granularity at which the system takes actions during self-tuning is at a component-level. We also assume that components can be throttled, isolated, etc. towards improving the overall performance characteristics of the system as a whole. The primary objective in our experiments was to minimize the duration of overload without impacting the live workloads. Another key contribution is the use of component saliency analysis, enabling self-tuning system to identify the bottleneck components that lead to performance degradation.

The rest of this paper is organized as follows. The next section briefly presents the past literature followed by Sect. 3, which gives an overview of our system and introduces the key ideas in our self-tuning solution. Section 4 describes in detail the system we have implemented. We conclude in Sect. 6 with some possible extensions of our work and a discussion of some of broader challenges in building a self-tuning system of the kind we have envisaged in this paper.

2 Related Work

Use of learning algorithms for performance modeling [7,13] and failure predictions at a disk-level [11,16] has been researched for more than two decades. With the evolution of I/O intensive scientific applications and the prevalence of data redundancy and replication schemes, focus has gradually moved towards large scale storage systems. Research related to large scale storage systems [4,19] has

gained importance in the recent years due the complexity of such systems increasing exponentially in the recent past and the associated difficulty in maintenance and management of such systems.

Self-* Storage systems was proposed in [8], where Self-* stands for self-configuring, self-organizing, self-tuning, self-healing and self-managing. Though, this is an ideal scenario, the work was carried out from a storage administrators' point of view. Architectural advancements in the past decade have ensured that cluster-based storage systems can provide a self-organizing system [23] to some extent but aspects such as self-tuning and self-healing continue to be rapidly evolving areas of research. Our current work is aimed at addressing the possibility of self-tuning using learning algorithms. Analysis of storage load in the past was restricted to latency analysis. More recently, there has been progress in measurement of the end-to-end load of the storage system at the client level [4]. The load is measured as a function of standard benchmark operation that makes it unique and user/administrator friendly for load analysis.

3 System Overview

Datasets used in our experiments are performance counters collected from storage system by periodically polling for and extracting counters related to all the active components. Every counter set (a snapshot of all the counter values collected together) is labeled with one of the three broad categories, viz., ZEROLOAD (**Z**), NORMAL (**N**), *OVERLOAD* (**O**). The labels are typically derived through thresholds implied by formal service-level-agreements (SLAs) on these systems. SLAs for storage systems include the total up-time of the system, time taken for read/write operations, number of retries and so on. Thresholds are defined to help maintain the agreed levels of latencies/up-time and a potential breach of the same is proactively avoided through actions taken by the proposed self-tuning system. The data collection in our experiments is similar to that collected by [4] but for real world application I/O patterns. The number of counters is typically too large for use in a model that will operate in real-time. We use a couple of simple strategies to bring the number of counters that will be considered for the final model down to a few hundreds. We use the terms counter and feature (of the current system state) interchangeably in the rest of this paper. A fraction (30%) of the dataset collected is held-out as a test set that we use for the final model selection based on a weighted class loss function.

Any model that predicts the state of the system based on performance counter data in real time (the predictions are useful only if they are actionable in real-time) needs to be fast, interpretable and accurate, in that order of priority. It needs to be fast because decisions using the model have to be taken in real time for them to be useful. Interpretability is important for being able to accommodate manual/automated interventions as required. Accuracy in this case may not be as important as the other two since we believe that some intervention is better than no intervention at all. The system performance cannot be significantly worse even if the model is not terribly accurate.

We propose a Pruned Random Forest (PRF) based solution augmented with a way to accommodate weighted classes. Our solution is built around a random forest classifier that will classify a performance counter vector into one of three classes — **Z**, **N**, **O**. Note that we assume the performance counter vector is a complete representation of the 'current' state of the system as a whole. The random forest constructed from the subset of counters is pruned based on a weighted *Matthews Correlation Coefficienct* (MCC) [20] ranking of the trees in the random forest. Optimal pruning is hard in general and pruning is probably not even worth it in most applications. However, several recent papers [5,6,15,21,22,24] have shown the effectiveness of pruning using heuristics. It can drastically cut down the number of trees in the random forest without compromising on the accuracy/generalizability of the model. In many cases, the size of the random forest is down to nearly one tenth of its original size while improving the accuracy in most cases by explicitly preserving or enhancing the strength and diversity of the ensemble [5]. The most recent attempt in this area [5] makes pruning very attractive because it effectively eliminates the need to specify any limits on the size of the pruned ensemble. It makes pruning almost independent of the original (pre-pruning) size of the ensemble.

Pruning is important to ensure the first two criteria we laid out earlier — speed and interpretability. Unfortunately, random forests with hundreds of trees and several hundreds of features on which each random tree in the forest has been built can be very expensive in their decision making. Each decision requires the new input to be pushed through all the trees in forest before bagging them together. Hence, it is not quite suited for high throughput online scenarios like that of online tuning of large scale filers where prediction and real-time analysis are necessary for tuning the storage system. Cutting the size of the random forest by a factor of 10 directly cuts down on the decision making time by a factor of 10. Our algorithm generates several candidate pruned forests by varying a couple of hyperparameters. The candidate forests are evaluated based on a loss function that is sensitive to the fact that we are more concerned about misclassifications related to an **O**verload situation — either a normal load being classified as **O**verload or a genuine **O**verload situation not being recognized. We then annotate each node in every decision tree of the pruned random forest with exactly one system component label. A node is annotated with the component from which the feature (counter value) being tested at the node has originated. The final pruned, annotated random forest is the predictive model deployed in the system for self-tuning. The schematic in Fig. 1 illustrates the entire proposed system architecture.

While in deployment, the self-tuning system monitors a stream of sets of counter values. These counters are the same as those used during training and are collected at frequent (5–10 s) intervals. The predictive model is run on each snapshot of counter values in real-time. If the model predicts that it is potentially an overload situation, it carries out a *component saliency analysis* to determine, which component(s) is(are) the 'bottleneck(s)' leading to the predicted overload situation. The internal annotated structure of the pruned random forest is used

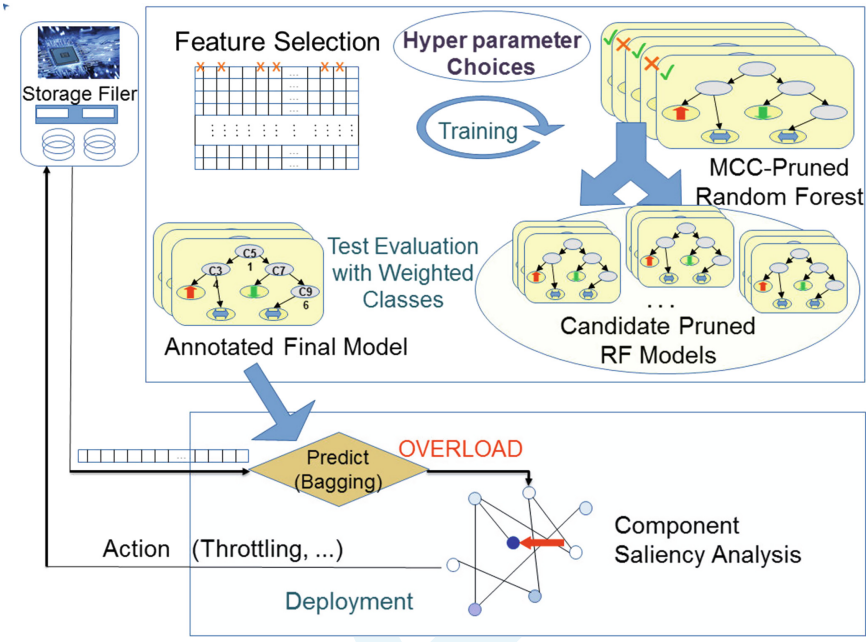

Fig. 1. System architecture

to arrive at a 'diagnosis' regarding the most likely components that need some action in case the system suffers from an overload situation. Experimental results reported in this paper are based on an implementation of self-tuning through throttling of I/O on specific components. Our experimental results adequately demonstrate the usefulness of our proposed scheme as a way to implement self-tuning on live storage systems. Short term reconfigurations such as throttling for short periods enables minimal impact on live workloads and reduce **O**verload scenarios.

4 Implementation Details

4.1 Feature Selection from Counter Data

Importance of good feature selection, both as a dimensionality reduction tool and as a way to improving accuracy (by removing features that are redundant and potentially misleading), in the design of machine learning algorithms is well known [1,18]. In our context, it was crucial to bring the dimensionality down significantly for filer dataset with more than 100,000 features. Even after the routine pre-processing task of removing columns with zero variance the time taken to generate a random forest model for our dataset was in the order of days on a standard high memory machine. We use two custom feature selection steps in our scheme. Suppose there are d features and the 0^{th} 'feature' is the

class label. Let's denote the correlation between the i^{th} and the j^{th} feature in the given dataset $\mathcal{D}$ as $\rho_{\mathcal{D}}(i,j)$ and set of all features to be retained as $\mathcal{F}$, where $\mathcal{F}$ is initialized to the set of all non-zero variance columns.

1. **Class Correlation Test:** Remove features that do not correlate well with the target variable (class label).

$$\mathcal{F} \leftarrow \{i \mid i \in \mathcal{F} \wedge |\rho_{\mathcal{D}}(i,0)| > \tau_c\} \tag{1}$$

   The number of features considered during training decreases as we increase the value of τ_c.
2. **Inter-Correlation Test:** If two features exhibit a strong correlation among themselves, remove one of them. Let

$$\overline{\rho_{\mathcal{D}}}(i) = \frac{\sum_{j \in \mathcal{F}} \rho_{\mathcal{D}}(i,j)}{d}$$

   denote the average correlation between feature i and the other features in the dataset. We update the feature set to

$$\mathcal{F} \leftarrow \{i \mid (i,j \in \mathcal{F}) \wedge (|\rho_{\mathcal{D}}(i,j)| \geq \tau_f) \wedge (\overline{\rho_{\mathcal{D}}}(i) \leq \overline{\rho_{\mathcal{D}}}(j))\} \tag{2}$$

   The idea is to retain the feature that correlates less with the other features, from a pair that has a high correlation [3,10,12]. This test enhances diversity among the features considered for the random forest construction. The number of features considered during training increases as we increase the value of τ_f.

4.2 Random Forest Pruning

We follow the MCC based pruning proposed in [5]. MCC, as a choice of performance measure is known to be an unbiased estimate of accuracy of a model [20]. One of the internal estimates known to act as a cross-validation step during forest construction is the Out Of Bag (OOB) error measure [2]. It helps in avoiding the need to maintain an exclusive hold-out set for testing or ever carry out explicit cross validation. MCC for each tree is defined on the OOB set for that tree. For a multiclass scenario like ours, the MCC is computed as the weighted sum of the class-MCCs computed for each class. Class-MCC is computed by taking the class against the rest, and computing the class-MCC as if it were a binary classification problem. Denoting the full random forest produced on the selected subset of features be $\mathcal{T}$, let class-MCC calculated in this manner for the k^{th} tree $T_k \in \mathcal{T}$ and class c be denoted $m_c^{(k)}$. Also, let the fraction of the OOB set for T_k that belongs to class c be denoted $w_c^{(k)}$. The weighted MCC $m^{(k)}$ for T_k is then defined as

$$m^{(k)} = \sum_{c \in C} m_c^{(k)} \times w_c^{(k)}$$

It was shown empirically in [5] that removing trees whose weighted MCC is below the 80^{th} percentile often works well. For our purposes we use a derived limit on the number of trees in the model to retain as many trees in the order of their weighted MCC values.

4.3 Hyperparameter Tuning for Weighted Classes

Our algorithm introduces two hyperparameters τ_c, τ_f in Eqs. 1 and 2. The effect of τ_c on the overall training time is the most pronounced. Notice that the inter-correlation test requires a pair-wise test on all features. Having a very low value of τ_c can therefore make the inter-correlation test unviable. Similarly, having a large value of τ_f can significantly increase the number of features on which the random forest needs to be built. We set a lower bound τ_c^* on $\tau_c \geq \tau_c^*$ and an upper bound τ_f^* on $\tau_f \leq \tau_f^*$ primarily from training time considerations. We set an upper limit ϵ^* on the OOB error that any of the models can commit. Another important performance criterion is the maximum time available to take a decision when the model is deployed. This time is directly dependent on the number of conditions that need to be checked on any test input for classifying it using a random forest. A rough estimate of this is clearly the product of the average depth of a tree $d_{\mathcal{T}}$ in the forest and the number of the trees $n_{\mathcal{T}}$ in the forest. We therefore require that $d_{\mathcal{T}} \times n_{\mathcal{T}} \leq \Delta$ for some constant Δ derived from the maximum time available for an online decision. We calculate the average depth of the random forest $\mathcal{T}$ consisting of trees $\{T_1, \ldots, T_{n_{\mathcal{T}}}\}$ as

$$d_i = \frac{\sum_{j=1}^{l_i} d_{ij}}{l_i}, \quad d_{\mathcal{T}} = \frac{\sum_{i=1}^{n_{\mathcal{T}}} d_i}{n_{\mathcal{T}}}$$

where d_i is the average depth of the tree T_i with l_i leaves, the depth of leaf j being d_{ij}. The pseudocode for the complete hyperparameter tuning scheme is shown in Algorithm 1. It generates a series of candidate models for evaluation.

Algorithm 1. Hyperparameter Tuning

Fix values for τ_c^*, τ_f^*, ϵ^* and Δ.
for $\tau_c = \tau_c^* \ldots 1$ **step** 0.1 **do**
  for $\tau_f = \tau_f^* \ldots 0$ **step** -0.1 **do**
    # Remove features with low class-correlation
    $\mathcal{F} \leftarrow \{i \mid i \in \mathcal{F} \wedge \rho_{\mathcal{D}}(i, 0) < \tau_c\}$
    # Remove features with high inter-correlation
    $\mathcal{F} \leftarrow \{i \mid (i, j \in \mathcal{F}) \wedge (|\rho_{\mathcal{D}}(i, j)| \geq \tau_f) \wedge (\overline{\rho_{\mathcal{D}}}(i) \leq \overline{\rho_{\mathcal{D}}}(j))\}$
    # Continue only if feature set is different from previous iteration
    # Build random forest
    $\mathcal{T} \leftarrow$ `randomForest`$(\mathcal{D}, \mathcal{F})$; $\epsilon_{\mathcal{T}} \leftarrow$ OOB Error of $\mathcal{T}$; $d_{\mathcal{T}} \leftarrow$ Average Depth of $\mathcal{T}$
    if $\epsilon_{\mathcal{T}} > \epsilon^*$ **then**
      break ▷ $\epsilon_{\mathcal{T}}$ is expected to increase for any smaller τ_f
    else
      $\mathcal{T} \leftarrow \{T_i \mid r(T_i) \leq \Delta / d_{\mathcal{T}}\}$ ▷ $r(T_i)$ = weighted MCC rank of $T_i \in \mathcal{T}$
      Output $\mathcal{T}$ as a candidate model
    end if
  end for
  if $\tau_f = \tau_f^*$ **then**
    break ▷ (τ_c, τ_f^*) pair failed. So $\epsilon_{\mathcal{T}}$ is expected to increase for any larger τ_c
    # No more candidate forests get generated
  end if
end for

Table 1. Confusion and loss matrices

Confusion Matrix

	Z	**N**	**O**
Z	c_{zz}	c_{zn}	c_{zo}
N	c_{nz}	c_{nn}	c_{no}
O	c_{oz}	c_{on}	c_{oo}

Loss Matrix

	Z	**N**	**O**
Z	0	1	λ
N	1	0	λ
O	λ	λ	0

The models generated by Algorithm 1 are evaluated on a separate (hold-out) test set using a weighted class loss function.

4.4 Weighted Class Loss Function Evaluation

We are concerned primarily about accurate predictions of **O**verload situations. It is important to control both false-positives and false-negatives in this case — false positives cause unnecessary 'tuning' actions that can degrade the system performance, false negatives make the self tuning system ineffective. This does not distort the overall performance of the random forests because the MCC based pruning carried out prior to the test evaluation ensures overfitting for the **O**verload class is avoided. Each candidate model is evaluated against the hold-out test set. The confusion matrix is computed for the model on the test set. Table 1 shows a representative confusion matrix and the loss matrix, where $\lambda > 1$ is a fixed constant. The loss matrix ensures that any mis-classification involving **O** is penalized much more than those not involving **O**. We can use an appropriately large value of λ to ensure that models performing better over **O** are preferred. The final model picked is the one that minimizes

$$c_{zn} + c_{nz} + \lambda(c_{oz} + c_{on} + c_{zo} + c_{no})$$

4.5 Component Saliency Analysis

The final model used for prediction is annotated by associating each node of the final random forest with the component to which the feature being tested at the node belongs. The idea is that an overload situation arising due to a component would invariably be indicated by counters belonging to that component. So one would expect that in case it is an overload situation, a number of counters from the component 'culprit' would have figured in the paths in a number of trees that the current test counter vector 'passed through' during the decision making. The key components responsible would often be represented in these decision paths. Notice that the number of components is of the same order as the number of trees in the forest — both are around 100. We therefore zero in on the salient component that could be 'responsible' for the overload situation as follows. Let the final model be the forest $\{T_1, \ldots, T_n\}$. Given a test counter vector $\mathbf{c}$, let v_i be a binary variable representing the verdict of $T_i(\mathbf{c})$. T_i on $\mathbf{c} \implies v_i$ is 1 if and only if T_i concludes that $\mathbf{c}$ indicates an **O**verload situation. Let $P_i(\mathbf{c})$ denote the path taken by $\mathbf{c}$ in T_i and S_i, a vector of length equal to the number of

components in the system, for each $1 \leq i \leq n$. The pseudocode for the models' response for every new counter vector is shown in Algorithm 2.

Algorithm 2. Deployment Scenario — Component Saliency Analysis

New counter vector $\mathbf{c}$, model $\{T_1, \ldots, T_n\}$.
for $i = 1, \ldots, n$ **do**
  $v_i = T_i(\mathbf{c})$
  if $v_i = 1$ **then**
    for $x \in P_i(\mathbf{c})$ **do**
      $S_{ij} += 1$ if $x \in$ Component j
    end for
  end if
  if $\sum_{i=1}^{n} v_i > \frac{n}{2}$ **then** ▷ Bagging
    $S^* = \sum_{v_i=1} S_i$
    # return the (index of) most salient component for tuning action
    return $\arg\max_j S_j^*$ ▷ S_j^* is the j^{th} component of S^*
  end if
end for

5 Experiment Setup

We conducted our experiments on a NetApp Cluster storage system with workloads similar to those observed in the real world. These workloads were generated using the 'Standard Performance Evaluation Corporations' (SPEC) Solution File Server (SFS) tool, designed by a consortium of storage vendors to evaluate the performance of different storage systems for real-world workload patterns. The key characteristics of the four different datasets used in our experiments are shown in Table 2. During offline testing, each dataset was split with 70% used as training data and the rest used for testing. Random Forest pruning and the annotations required for our algorithm was implemented by patching the *randomForest* package in *R* [14]. The size of the random forests generated for each

Table 2. Training workload/dataset summary

Dataset	Size	Instances	Clients	Workload description
TXN	3.1 Gb	7,715	43	Online Txn Data (financial, telecom,...)
SVD	2.4 Gb	5,767	20	Streaming Video Data
BLD	3.0 Gb	7,489	39	Software Build — meta-data operations, file reads, source compilation and binary data generation
MIX	2.4 Gb	6,003	102	Mixture (of TXN, SVD and BLD) Workload

Table 3. Summary of implementation results for filer data. ($\%Overload^*$ – Percentage improvement in overload classification after pruning)

	$\lvert\mathcal{F}\rvert$	$min(t_c, t_f)$	$max(t_c, t_f)$	PRF	CRF	$d_{\mathcal{T}}$	$candidates$	k_{PRF}	$\%Overload^*$
TXN	177	$(0.01, 0.65)$	$(0.21, 0.9)$	25.42	25.52	15	7	88	0
SVD	101	$(0.1, 0.70)$	$(0.5, 0.9)$	3.29	3.35	9	9	153	0.65
BLD	221	$(0.01, 0.7)$	$(0.23, 0.9)$	16.55	16.93	12	5	107	0
MIX	12	$(0.1, 0.75)$	$(0.5, 0.9)$	2.27	2.47	8	6	185	0.9

(τ_c, τ_f) combination was 500, before they were pruned to the required size. Based on the filer configuration, $\Delta = 2,000$ ms was set to identify the size of the candidate pruned forest.

Discussion: The implementation results are summarized in Table 3. The table summarizes the hyperparameter ranges that were searched $\{min(t_c, t_f), max(t_c, t_f)\}$, the number of features that were retained ($\lvert\mathcal{F}\rvert$), number of forests produced as candidates (k_{PRF}) and the accuracy of the pruned (PRF) and unpruned (CRF) models. Improvements observed for **O**verload class due to pruning is also presented as $\%Overload^*$. $d_{\mathcal{T}}$ and $candidates$ column in Table 3 indicate the average depth of trees and the number of alternated forests available for consideration. Hyperparameter tuning is illustrated in plots of the accuracy of the model against τ_f threshold for every fixed value of τ_c. The plots are shown in Fig. 2. Highlighted rectangular regions in the plot indicates the choices of τ_f and τ_c satisfying $\epsilon_{\mathcal{T}} \leq \epsilon^*$. The best candidate within the rectangular region is identified using the Loss matrix with $\lambda = 8$.

5.1 Online Self-tuning

Many possible configuration changes are available for administrators to optimize/tune/balance load on NetApp filer. Choosing the most appropriate configuration change depends on identifying the bottleneck components. Throttling the load on storage objects such as volumes is one such option [17]. Throttling helps in deterring load generated by users using a volume. This will lower the load affecting throughput of storage for a small set of users but a gain can be observed by rest of the users serviced by other volumes. Component saliency analysis helps in identifying if any single volume is accommodating huge I/O forming the bottleneck. Self-tuning of storage system was implemented on a live filer and tested on workloads for a period of 4 h. Throttling was initiated when **O**verload is predicted for 5 consecutive counter snapshots. It was enabled for a short interval (5 min). Also, throttling was carried out by restricting the I/O bandwidth of the volume to the average I/O load observed on the volume in the past 15 min. Figure 3 shows the Gantt chart of the various Overload durations encountered during the online testing phase. The figure clearly shows a significant reduction (40%) in the occurrence of **O**verload scenarios at the cost of loss

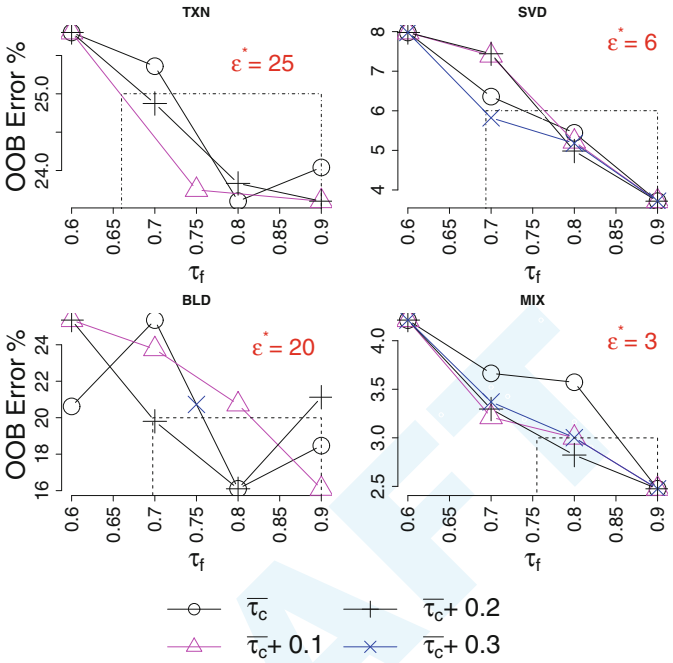

Fig. 2. Effect of (τ_c, τ_f) on accuracy

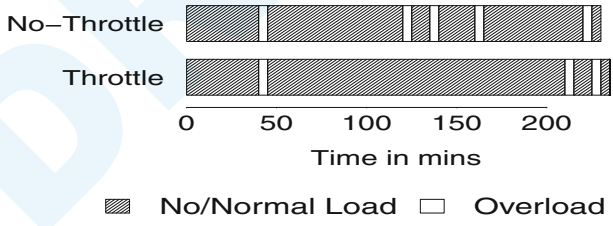

Fig. 3. Effect of self-tuning on load of a filer

of throughput of 1.33%. Self-tuning, when implemented using unpruned random forest will lead to 3 to 5 times increase in computation cost along with loss in prediction accuracy.

6 Conclusions and Future Work

Storage system counter data is very useful in identifying its health and the same has been demonstrated for different industry workload patterns. With appropriate modifications (like pruning) to standard machine learning algorithms such as random forests, it is possible to predict the impending health of the system in real-time and initiate corrective actions as a response to prevent unfavourable

events such as Overload. For a limited scenario, we have demonstrated the effectiveness of such a self-tuning scheme. We believe this can be extended to numerous other scenarios. The success of such self-tuning primarily depends on the quality of the predictions by machine learning model followed by bottleneck component identification. Experiments with (i) adaptation of other learning algorithms to real-time scenarios, (ii) more sophisticated real-time bottleneck diagnosis algorithms and (iii) corrective actions using the large array of tuning options provided by the system vendor, other than simple throttling, are possible extensions to the ideas explored in this paper.

References

1. Almuallim, H., Dietterich, T.G.: Learning boolean concepts in the presence of many irrelevant features. Artif. Intell. **69**(1–2), 279–305 (1994)
2. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
3. Contributions, M.K.: caret: Classification and Regression Training, r package version 5.15-044 (2012)
4. Dheenadayalan, K., Muralidhara, V.N., 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
5. Dheenadayalan, K., Srinivasaraghavan, G., Muralidhara, V.N.: Pruning a random forest by learning a learning algorithm. MLDM 2016. LNCS (LNAI), vol. 9729, pp. 516–529. Springer, Cham (2016).
6. Fawagreh, K., Gaber, M.M., Elyan, E.: On extreme pruning of random forest ensembles for real-time predictive applications. CoRR abs/1503.04996 (2015)
7. Ganapathi, A.S.: Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning. Ph.D. thesis, EECS Department, University of California, Berkeley, December 2009
8. Ganger, G.R., Strunk, J.D., Klosterman, A.J.: Self-*storage: Brick-based storage with automated administration. Technical report, Carnegie Mellon University, School of Computer Science, Technical report (2003)
9. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, pp. 29–43. ACM (2003)
10. Hall, M.A.: Correlation-based feature selection for discrete and numeric class machine learning. In: Proceedings of the Seventeenth International Conference on Machine Learning, pp. 359–366. Morgan Kaufmann Publishers Inc. (2000)
11. Hamerly, G., Elkan, C.: Bayesian approaches to failure prediction for disk drives, pp. 202–209. Morgan Kaufmann Publishers Inc. (2001)
12. Kohavi, R., John, G.H.: Wrappers for feature subset selection. Artif. Intell. **97**(1–2), 273–324 (1997)
13. Lee, E.K.: Performance Modeling and Analysis of Disk Arrays. Ph.D. thesis, EECS Department, University of California, Berkeley, August 1993
14. Liaw, A., Wiener, M.: Classification and regression by randomforest. R News **2**(3), 18–22 (2002)

15. Martinez-Munoz, G., Hernandez-Lobato, D., Suarez, A.: An analysis of ensemble pruning techniques based on ordered aggregation. IEEE Trans. Patt. Anal. Mach. Intell. **31**(2), 245–259 (2009)
16. Murray, J.F., Hughes, G.F., Kreutz-Delgado, K.: Machine learning methods for predicting failures in hard drives: a multiple-instance application. J. Mach. Learn. Res. **6**, 783–816 (2005)
17. NetApp Inc.: Managing workload performance by using storage qos. https://library.netapp.com/ecmdocs/ECMP1196798/html/GUID-660A6C00-6D7E-4EE5-B97E-9D33C0B706B5.html
18. Opitz, D.W.: Feature selection for ensembles. In: Proceedings of the Sixteenth National Conference on Artificial Intelligence, pp. 379–384. American Association for Artificial Intelligence (1999)
19. Pollack, K.T., Uttamchandani, S.M.: Genesis: a scalable self-evolving performance management framework for storage systems. In: 26th IEEE International Conference on Distributed Computing Systems, p. 33 (2006)
20. Powers, D.M.W.: Evaluation: from precision, recall and f-measure to roc., informedness, markedness & correlation. J. Mach. Learn. Technol. **2**(1), 37–63 (2011)
21. Schwing, A.G., Zach, C., Zheng, Y., Pollefeys, M.: Adaptive random forest - how many "experts" to ask before making a decision? In: Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition, pp. 1377–1384. IEEE Computer Society (2011)
22. Tamon, C., Xiang, J.: On the boosting pruning problem. In: López de Mántaras, R., Plaza, E. (eds.) ECML 2000. LNCS (LNAI), vol. 1810, pp. 404–412. Springer, Heidelberg (2000).
23. Tang, H., Gulbeden, A., Zhou, J., Strathearn, W., Yang, T., Chu, L.: A self-organizing storage cluster for parallel data-intensive applications. In: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, p. 52. IEEE Computer Society (2004)
24. Tsoumakas, G., Partalas, I., Vlahavas, I.: An ensemble pruning primer. In: Okun, O., Valentini, G. (eds.) Applications of Supervised and Unsupervised Ensemble Methods. SCI, vol. 245, pp. 1–13. Springer, Heidelberg (2009).
25. Zhu, Y., Jiang, H., Wang, J., Xian, F.: Hba: distributed metadata management for large cluster based storage systems. IEEE Trans. Parallel Distrib. Syst. **19**(6), 750–763 (2008)