Fault Management in Map-Reduce through Early Detection of Anomalous Nodes

Selvi Kadirvel, Jeffrey Ho and José A. B. Fortes
University of Florida
Email: selvik@ufl.edu, jho@cise.ufl.edu, fortes@ufl.edu

***Abstract*—Map-Reduce frameworks such as Hadoop have built-in fault-tolerance mechanisms that allow jobs to run to completion even in the presence of certain faults. However, these jobs can experience severe performance penalties under faulty conditions. In this paper, we present Fault-Managed Map-Reduce (FMR) which augments Hadoop with the functionality to mitigate job execution time penalties. FMR uses an anomaly detection algorithm based on sparse coding to anticipate a faulty slave node. This proposed technique has the following key advantages: (1) model training uses only normal-class data, (2) time taken for prediction is less than a second, and (3) confidence estimates are produced along with the anomaly prediction. FMR uses the result of anomaly detection to invoke a closed-loop recovery action, namely dynamic resource scaling. A scaling heuristic is proposed to determine the extent of scaling necessary to reduce impending performance penalty. FMR facilitates practical adoption by being implemented as a set of libraries and scripts that require no changes to the underlying source code of Hadoop. A set of realistic Map-Reduce applications were studied through a few thousand job executions on a 72-node Hadoop testbed. Detailed empirical evaluation shows that FMR successfully mitigates performance penalties from 119% down to 14%, averaged across experiments.**

I. Introduction

Innovations in infrastructure, middleware and applications have made "*big data*" analytics possible and economically viable in a wide range of fields such as bioinformatics, data mining, web indexing, document classification, recommendation systems, etc. The Map-Reduce (MR) programming paradigm [17] along with the free and widely supported open-source implementation, Hadoop [1], has become a popular choice for incorporating data analytics in industry, government as well as academic domains. One of the important benefits of this choice is that job parallelization, data distribution and fault-tolerance are facilitated and provided by the framework itself.

Enterprise data centers, in-house clusters and cloud computing environments (that host Map-Reduce platforms) experience many faults and failures as shown in recent studies by [31], [33], [32] and [3]. The causes for these faults include scale, heterogeneity, geographical distribution, configuration management over a large set of inter-dependent services and human error as illustrated in [9]. These faults adversely affect applications running in these environments resulting in job performance degradations, failed jobs, increased costs for users and loss of revenue for the provider when Service Level Objectives are violated. Wang et al. in [38], show through simulation studies that a single node fault can result in up to 139% performance slowdown in Map-Reduce. Dinu et al. in [18] record penalties of up to 350% in job run time due to TaskTracker failures. Ananthanarayanan et al. in [7], show that job completion times in Dryad (an implementation of the Map-Reduce paradigm) can be inflated by 34% because of outliers and that faster completion times (by reducing the effect of outliers) provide a competitive advantage to service providers.

These performance variabilities and penalties make it challenging to use Map-Reduce where response time is important, such as in user-facing social networking applications at Facebook [10], user-customization applications at LinkedIn [6] and user click-stream processing, web-index generation and advertisement selection applications at Microsoft [22].

Fault-managed Map-Reduce (FMR), presented in this paper, aims at mitigating these performance penalties experienced by Map-Reduce jobs. FMR uses a Monitor-Analyse-Plan-Execute (MAPE) control loop to provide an online, on-demand and closed-loop solution to fault management. In FMR, faults are anticipated through the detection of anomalous conditions that are indicative of an impending fault [31] [23].

For anomaly detection in this context, we propose the use of a simple machine-learning technique based on sparse coding. This technique satisfies the following two requirements: (1) model training using only *normal-class* data (as opposed to the use of both normal-class and anomaly-class data) and (2) fast prediction time. Normal class data captures run time behavior of a job that has not experienced a performance fault. The need for training using only normal-class data is necessary because anomaly-class data that is representative of all (or most) possible types of faults, is difficult to obtain in a production environment. Prediction computation time using the proposed sparse-coding technique is less than a second. This allows the anomaly detection module to be incorporated in an online fashion within the MAPE loop for handling faults during job execution. In addition to these essential requirements, sparse coding based anomaly detection has two other benefits. The sparse coding model is deployed locally on each slave node and does not need to communicate or synchronize with models on other nodes to make a prediction. This makes FMR applicable to both homogeneous and heterogeneous Map-Reduce environments. The time taken to train a sparse coding model is in the order of a few seconds. This makes it possible to quickly create models for a new Map-Reduce application and also to quickly re-train models when system characteristics change. Map-Reduce applications need to be instrumented to emit heart beats, which are further processed to construct feature vectors that then serve as input to the anomaly detection module.

After an anomaly is detected, FMR uses dynamic resource scaling to reduce the performance penalty due to an impending fault. A scaling heuristic is used to determine the extent of scaling necessary. This heuristic uses performance prediction models derived from our previous work [27] to estimate Map-Reduce job execution times both in fault-free and fault-present conditions. The cost due to increased execution time is compared with the cost for additional resources and then a suitable scaling decision is taken.

FMR leverages built-in features of Hadoop in order to implement its control loop. This includes features such as (1) the provision for seamless dynamic addition of slave nodes to an executing job, (2) blacklisting of slave nodes to stop assignment of new tasks to a slave node, and (3) the node health script feature to periodically monitor a user-defined set of conditions on the slave. FMR has been designed to require no changes to the underlying Hadoop code base, thereby facilitating practical adoption.

The increasing prevalence of Map-Reduce applications along with increasingly fault-prone, large-scale computing environments, makes FMR a timely and critical component to improve Map-Reduce performance in the presence of faults. The main contributions of this paper in the context of the proposed FMR are as follows:

(1) Fault anticipation and early detection through a sparse-coding based anomaly detection method. The proposed technique has a high true positive rate of 0.95 and a high true negative rate of 0.93 averaged across experiments. Additionally, it provides the benefits of short training and testing times, requiring only normal-class data for training.

(2) A closed-loop, online dynamic resource scaling approach to reduce fault-induced performance penalties. Observed performance penalties (without FMR) range between 18% up to 210%. Using FMR, penalties were brought down to values ranging between 5% to 46%. FMR has been thoroughly evaluated using a few thousand experiments on a 72-node in-house cluster. Injected faults include CPU, memory and disk hog processes as well as node crashes. Benchmark applications from the domains of text mining and machine-learning were used for the evaluation of FMR.

Other building blocks that enable FMR were proposed in our prior works: (1) a comparative evaluation of regression based machine-learning techniques for predicting the performance of Map-Reduce jobs [27] and (2) a study of the effect of various types of faults on a MapReduce job (motivating the need for FMR) and the feasibility of resource scaling to improve performance of an executing Map-Reduce job [26].

In Section II, background to the problem and related work are discussed. In Section III, the Fault-managed Map-Reduce approach is introduced. In Section IV, implementation details of FMR are described. Section V consists of experimental validation of FMR and a discussion of the results. Conclusions are presented in Section VI.

II. Background and Related Work

In this section, we summarize Map-Reduce research related to fault management and bring out the need for FMR.

Effect of faults in Hadoop: Dinu et. al [18] evaluate the behavior of Hadoop in the presence of fail-stop faults of an entire compute node as well as Hadoop components such as the TaskTracker and DataNode daemons. The authors show that TaskTracker failures can result in up to 350% penalty while DataNode failures can lead to 218% penalty. Wang et. al [38] present a Map-Reduce simulator, MRPerf and show that it can capture fault effects. Their simulation experiments show penalties up to 186% for various injected faults. Our work [26] illustrates the effect of various factors on performance penalty such as number of slave nodes, time of fault injection and fault-detection timeout interval.These research results along with the increasing importance of Map-Reduce motivates our goal for improving fault management in Hadoop.

Fault diagnosis: The Fingerpointing project, that includes works such as [30], [34], and [8] focuses on *fault diagnosis* in Map-Reduce environments. Our approach focuses on fault *detection* and fault *recovery* through an online, closed-loop approach. However, diagnosis is important and our choice of sparse coding for anomaly detection is motivated by the need to extend detection to diagnosis in order to facilitate more targeted recovery actions.

Fault handling: In Mantri [7], outliers in an executing Dryad Map-Reduce job are identified through the use of static thresholds determined from application history. The determination of the correct threshold to use is challenging and a pre-set threshold can often drift to become incorrect in dynamic environments. Hadoop provides a built-in feature called speculative execution in which slow tasks are chosen to be executed through duplicate task instances. The deficiency of speculative execution in heterogeneous environments has been addressed by the LATE algorithm proposed by Zaharia et. al [39]. FMR applies to both performance faults as well as performance faults that lead to crash faults. However, the latter condition cannot be handled by speculative execution and LATE and this is empirically illustrated in Section V. Speculative execution also uses resources inefficiently through the execution of many duplicate tasks (for e.g. in [39] it was observed that as many as 80% of tasks were speculatively executed). In contrast to speculative execution and LATE, which use progress-based analytical models for detecting a slow task, FMR uses decentralized and local machine-learning models on each node for detecting anomalies.

Performance prediction: Predicting the completion time of a MapReduce job is done through analytical models in [37] and through simulation models in [24]. In [11], the authors predict map and reduce task slowdown using the gradient boosted decision tree model. However, prediction is based on offline analysis. The anomaly detection method proposed in this paper can be used in an online fashion and hence enables incorporation into the MAPE control loop.

Anomaly detection: Tan et. al [35] propose anomaly pre-

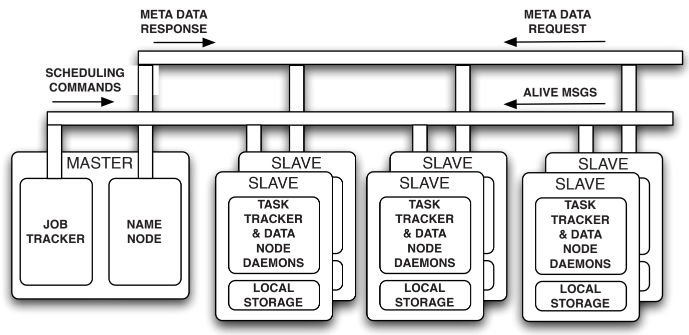
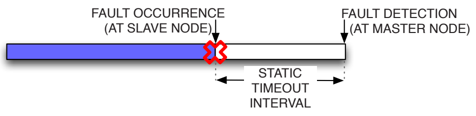
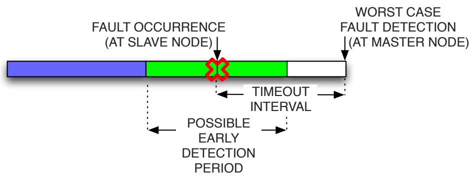


Fig. 1. Overview of Hadoop showing interactions between the JobTracker, NameNode, TaskTracker, DataNode hosted on the master and slave nodes.

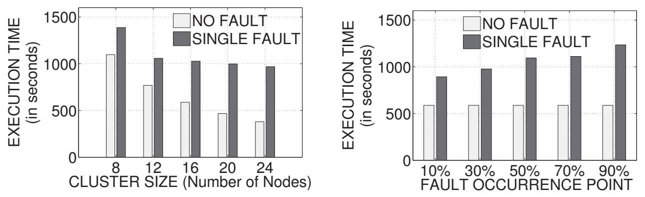

FAULT OCCURRENCE (AT SLAVE NODE)
FAULT DETECTION (AT MASTER NODE)
STATIC TIMEOUT INTERVAL

(a) Fault detection in Hadoop


FAULT OCCURRENCE (AT SLAVE NODE)
WORST CASE FAULT DETECTION (AT MASTER NODE)
TIMEOUT INTERVAL
POSSIBLE EARLY DETECTION PERIOD

(b) Improved fault detection in Hadoop with FMR

Fig. 2. The shaded (blue) region represents map and reduce tasks running on a slave. After the fault shown by a cross tasks stop running on this slave. Hadoop master detects the fault after a static timeout value. The lightly shaded (green) region introduced into the node timeline in (b) corresponds to the period leveraged by FMR for early fault detection



(a) (b)

Fig. 3. Execution time increase for Hadoop wordcount jobs of (a) different cluster sizes and (2) for node crash faults injected at different points in job progress.

vention schemes through the use of Markov chain models for general virtualized cloud computing. systems. However, it is a supervised learning technique which means that representative normal and anomaly class data is needed. [16] and [23] propose unsupervised techniques for anomaly detection. FMR's anomaly detection technique is similar in goal to these works.

Performance management: Starfish [24] and ARIA [37] propose the use of dynamic resource scaling for performance management of Hadoop jobs. The Starfish project does not handle faults and the ARIA project handles fail-stop faults. FMR's focus is on performance faults that result in degraded job execution times. AROMA [29] uses machine-learning techniques for resource allocation and configuration in Hadoop, however it does not handle performance deviations introduced by faults.

Our work is most similar to Jockey [22], in which resource allocation is used to guarantee job latencies for data parallel jobs. Jockey depends on an offline job profile simulator for completion time prediction; while FMR uses an online, machine-learning based model for prediction.

III. Design of Fault-Managed Map-Reduce

This paper focuses on the open-source Map-Reduce implementation, Hadoop [1]. Hadoop consists of the following main components: (1) *JobTracker* and *TaskTracker* daemons that manage scheduling and coordination of map and reduce tasks, and (2) *NameNode* and *DataNode* daemons that manage the Hadoop Distributed File Systems (HDFS). The JobTracker and NameNode daemons run on the Hadoop master node, while the TaskTracker and DataNode daemons run on the slave nodes. Figure 1 shows a simplified overview of Hadoop.

In a Map-Reduce job, when a node fails, all map tasks that were executed on this node (for this job) have to be re-executed on other healthy nodes. This is because map outputs are stored locally at each slave node (rather than being stored on the replicated HDFS). Map tasks whose outputs have already been read by corresponding reduce tasks need not be re-executed. The master node detects a slave node fault after a static timeout interval (as shown in Figure 2(a)) and then initiates re-execution. The performance penalty due to a single node fault is illustrated for Hadoop clusters of different sizes in Figure 3(a) and for the case when node faults occur at different points during a job's runtime in Figure 3(b). These penalties (ranging up to 155%) motivate the need for FMR.

One of the main contributors to the performance penalty experienced in the presence of faults is the timeout interval between fault occurrence and detection by the master. And therefore, in order to detect faults sooner, the key idea in FMR is the anticipation of a fault through anomaly detection. Figure 2(b) shows the period during which FMR attempts to detect faults. An anomaly refers to a condition that is indicative of an impending performance fault. In the context of this paper, a performance fault refers a Map-Reduce job's execution time exceeding a pre-specified Service Level Objective (SLO). By default, this SLO is the fault-free execution time. Several studies have shown that node crash faults are preceded by anomalous conditions [23] [31].

We use a machine-learning technique to identify whether a node is behaving in a manner that is unusual based on its own history for a specific type of application. After the detection of a node anomaly, recovery is initiated through dynamic resource scaling. Anomaly detection and dynamic resource scaling are described in the following subsections.

A. Anomaly Detection

Current anomaly detection techniques used in systems management depend on identifying various static thresholds as part of the control policy. When system metrics exceed these pre-determined thresholds, either alarms or suitable recovery actions are invoked. Although this simplifies the process of anomaly detection, these thresholds are difficult to determine and need to be customized as system conditions change.

In the context of machine-learning, anomaly detection can be viewed as a classification problem. Given a feature vector describing recent conditions on a compute node, we want to be able to predict whether or not this corresponds to an anomalous condition. In our work, an anomalous condition on a node could lead to a performance fault of the executing Map-Reduce job. Training data from compute nodes that are operating normally will be referred to as *normal-class* data; while those from a potentially faulty node will be referred to as *anomaly-class* data.

Application heart beats: The Map-Reduce application is instrumented to emit heart beats to indicate the rate of progress in processing input data. Heart beat timestamps are recorded locally on each slave node in a heart beat file. A sliding window of timestamps are processed to determine the heart beat rate. A sequence of heart beat rate values (referred to as a *heart beat wave*) captures application behavior and is used as the input feature vector to the anomaly detection module. Application heart beats have been used for autonomic management goals in [12] and [25].

An implicit requirement for the binary or multi-class formulation is the need for balanced and representative training data from each of the classes. In a production computational infrastructure, it is easy to obtain representative normal training data. However, requiring a system designer or administrator to provide sufficient and representative examples of anomalous data (such as from all possible performance anomalies) would strongly restrict the applicability of our approach. In order to overcome this limitation, we propose an anomaly detection method that can be trained using only normal-class data.

Sparse representation has received a great amount of attention in the signal processing community recently e.g., [20], [13], [21], and it readily provides a principled and flexible framework for feature-based anomaly detection needed in FMR. We note that there are several recent works in image processing and computer vision applying similar ideas to anomalous event detection (e.g., [15] [40]). Although originating from different application domains, these problems can all be considered as anomaly detection given only normal features. That is, anomalies are not explicitly defined based on input features but only relative to the training normal features, and this apparent asymmetry in training features is the main source of difficulty. Therefore, an algorithmic solution would require a suitable generative model for the normal features that can be used for identifying anomalies, and the sparse representation [20] offers such a model that is known for its simplicity, generalizability, and computational efficiency. Formally, in sparse representation, a feature (considered as a vector) $\mathbf{x}$ is represented as a linear combination of a small number of basis features chosen from a dictionary $\mathbb{D}$ (of basis features). In the following discussion, we will assume that $\mathbf{x}$ is a column vector of dimension d and the dictionary $\mathbb{D}$ is given as a $d \times l$ matrix such that $l > d$ ($\mathbb{D}$ has more columns than rows). The columns of the dictionary $\mathbb{D}$ are the basis features,

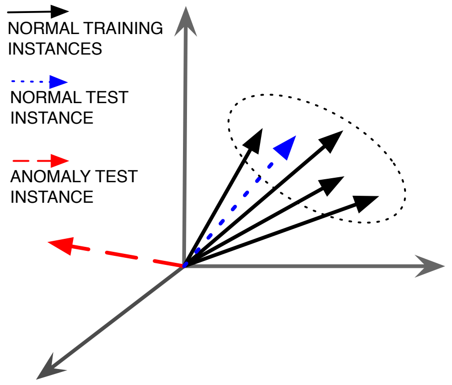


Fig. 4. A simplified diagram to illustrate the anomaly detection approach. Normal training instances (or feature vectors) are similar because they are produced by the same underlying process and hence with a high probability, lie within a confined subspace. A normal test instance is also produced by the same process and hence can be reconstructed well by other normal training instances (i.e. dictionary atoms) and as a result its sparse representation has low error values. On the other hand, an anomaly test instance is generated by a different underlying process and lies in a different subspace. Hence, when reconstructed using normal training instances, the sparse representation has larger errors.

and the main assumption in sparse signal representation is that a relevant feature $\mathbf{x}$ can be reconstructed by a small number k of columns. Mathematically, this can be written as

$$f\mathbf{x} = \mathbb{D}\,\mathbf{c}_{\mathbf{x}},$$

where $\mathbf{c}_{\mathbf{x}}$ is the sparse coefficient for $\mathbf{x}$ with respect to the dictionary $\mathbb{D}$, and all but k components of $\mathbf{c}_{\mathbf{x}}$ are zero. The integer k is the sparsity level of the feature $\mathbf{x}$ and it tells us that the feature $\mathbf{x}$ can be reconstructed by taking a linear combination of k columns of $\mathbb{D}$. In other words, $\mathbf{x}$ is in the linear span of these k columns. Given the dictionary $\mathbb{D}$, the sparsity level k is the parameter that controls the generalizability (or expressiveness) of the model. For example, when the sparsity level is set to $k = 1$, each feature vector $\mathbf{x}$ is just a column of $\mathbb{D}$ with a scaling since we are working with $\mathbf{x}$ such that $\mathbf{c}_{\mathbf{x}}$ only has one nonzero component in the equation above. For other values of k, the features are assumed to be those in the subspaces spanned by no more than k columns of $\mathbb{D}$. For this generative model, which is linear in nature, the dictionary and the sparsity level are the only two parameters used for specifying the normal features, and in particular, training of the model is exceptionally easy: simply take the normal training features as the dictionary columns. For anomaly detection, the main assumption we make in regard to the essential difference between feature vectors originating from normal operating states and anomalous states is that normal features can be sparsely approximated well using only a small number of normal features while anomaly features are expected to not enjoy this property. Therefore, given a dictionary $\mathbb{D}$ and a sparsity level k, we will consider any feature vector as a normal feature if it belongs to a subspace spanned by k columns of $\mathbb{D}$; otherwise, it will be considered as an anomaly. Figure 4 illustrates this through a simplified diagram.

More precisely, given a dictionary $\mathbb{D}$ of normal features and a sparsity level k, we expect that for a normal feature vector $\mathbf{x}$, its sparse-coding error, $\mathbf{e}$

$$\mathbf{e} = \mathbf{x} - \mathbb{D}\,\mathbf{c}_\mathbf{x},$$

should be a vector with small components and its magnitude follows some multivariate normal distribution (and the squared error norm $e = \|\mathbf{e}\|_2^2$ can be modeled by a χ-distribution $\phi_\chi(e)$). On the other hand, for an anomaly feature vector, its sparse-coding error $\mathbf{e}$ is expected to be large and its squared error norm does not follow the distribution ϕ_χ. Therefore, by estimating the background distribution $\phi_\chi(e)$ during training, the squared error norm e for an unknown feature vector $\mathbf{x}$ can be compared against ϕ_χ to determine its classification and the associated confidence level. We remark that the validity of using a sparse model for anomaly detection can only be supported empirically, and Figure 4 displays the results of several experiments that confirm our expectation that anomalous features incur large errors when sparsely coded with respect to the dictionary whose columns are normal features, providing a strong support for the sparse model. Furthermore, these results also suggest that the error $\mathbf{e}$ can be a useful feature for identifying the anomalies.

More specifically, the training component of our method consists of two steps: forming the dictionary $\mathbb{D}$ and estimating the distribution ϕ_χ. We randomly divide the training (normal) feature vectors into two groups. Feature vectors in the first group form the dictionary $\mathbb{D}$ and those in the second group are used to empirically estimate ϕ_χ and its cumulative distribution function $\mathbf{CDF}_{\phi_\chi}(e)$. The user specifies two parameters, $0 < \beta < \mathbf{fnr} < \alpha < 1$, which are used to bound the false negative rate $\mathbf{fnr}$ as follows: Let $e_n = \mathbf{CDF}^{-1}_{\phi_\chi}(1-\alpha)$ and $e_a = \mathbf{CDF}^{-1}_{\phi_\chi}(1-\beta)$. For any feature vector $\mathbf{x}$ with squared sparse-coding error norm e, it would be classified as normal if $e \le e_n$ or as an anomaly if $e \ge e_a$. For the "gray area" between e_n and e_a, we define the confidence level $\rho(e)$ of declaring $\mathbf{x}$ as an anomaly according to the formula,

$$\rho(e) = \frac{\mathbf{CDF}_{\phi_\chi}(e) - (1-\alpha)}{\alpha - \beta}.$$

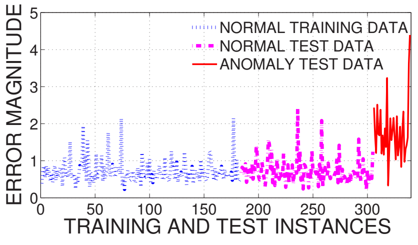
Note that $0 \le \rho(e) \le 1$ and for $\rho(e)$ to provide the confidence level, $\rho(e)$ simply scales the probability mass of ϕ_χ between e_n and e_a linearly to zero and one so that $\rho(e_n) = 0, \rho(e_a) = 1$. We also note that because we are declaring any feature vector $\mathbf{x}$ with error $e > e_a$ to be an anomaly, this gives β as a lower bound on $\mathbf{fnr}$, the false negative rate (the proportion of (training) normal feature vectors classified as anomalies). Similarly, we also have α as an upper bound for $\mathbf{fnr}$.

We remark that the key point in our method described above is the sparsity requirement, since without it, any anomaly feature vector can be approximated well using sufficiently many normal feature vectors in the dictionary $\mathbb{D}$. Only by imposing sparsity, it is then possible to use the error e as a meaningful value for classifying the feature vector $\mathbf{x}$. The sparsity requirement can further be justified using our qualitative understanding of the normal states and anomalies. In most applications, the normal features are comparatively more homogeneous than the anomalies, which due to their diverse origin and sporadic nature, are difficult to model consistently. Computationally, this homogeneity can be modeled by a dictionary $\mathbb{D}$ that captures (most of) the variability of the normal features such that each normal feature can be represented as a linear combination of only a small number (k) of basis features in $\mathbb{D}$. Therefore, this expected regularity of normal features provides the motivation and rationale for using sparse representation for their modeling. On the other hand, the heterogeneity of the anomalies precludes such modeling, and in general, an anomaly feature is not expected to be well approximated by a few basis features in $\mathbb{D}$. Therefore, using $\phi_\chi(e)$ as the background distribution, the sparse-coding error e provides a discriminative and useful quantity for classifying the feature vectors. In Figure 4, we plot these errors for three different Map-Reduce application datasets. We can observe the significant difference between errors for the normal and anomaly class instances.

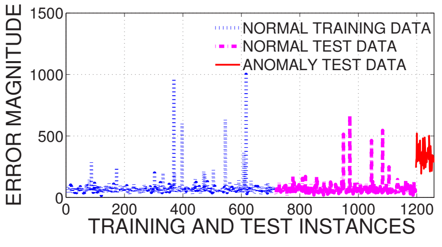
The proposed anomaly detection framework is conceptually simple and its implementation is straightforward. An important computational issue is to determine the sparse coefficients $\mathbf{c}_\mathbf{x}$ given a test feature $\mathbf{x}$. Fortunately, there are efficient algorithms such as orthogonal matching pursuit (MOD) [19] and LASSO [36] that compute sparse signal decomposition, given the signal $\mathbf{x}$ and dictionary $\mathbb{D}$. Using these efficient sparse coding algorithms, the running time of our method, both in training and testing, is fast and makes real-time anomaly detection feasible. Furthermore, the simplicity of our method allows various generalizations and extensions such as incorporating incremental updates of the dictionary $\mathbb{D}$ and background distribution ϕ_χ for anomaly detection in dynamic and complex environments, a topic we will pursue in the future.

B. Remediation through Dynamic Resource Scaling

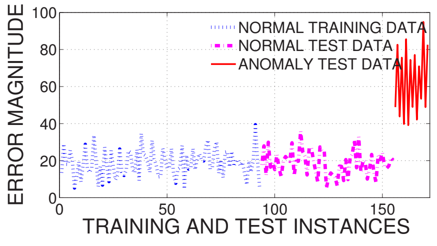
Dynamic resource scaling refers to the addition of Map-Reduce slave nodes to an executing job. This is a feasible solution to improve execution time in the presence of faults because of two reasons: (1) Hadoop allows for seamless addition of slave nodes (without restart of the master node daemons or changes to configuration files) and (2) Horizontal scaling is provided through a programming API in most virtualized and cloud environments. For a new node to be included in an executing job, TaskTracker and DataNode daemons must be started on it and the master node IP address must be provided to it. These newly started daemons will make a request for work to the master node and are then assigned data blocks to process. We note that the master node need not be aware of a slave node that may potentially be added in the future. This provides necessary flexibility to add as many nodes as needed for handling different faulty conditions. Additionally, an important design goal in our work has been to keep the underlying Hadoop framework unmodified in order to ensure that our solution can be easily adopted in practice. Dynamic resource scaling chosen as the remediation



(a) Wordcount



(b) Pi estimation



(c) Grep

Fig. 5. Errors in the sparse representation of training and test feature vectors for three Map-Reduce application datasets. Illustrates a significant difference in the magnitudes between normal class and anomaly class instances.

technique, facilitates this design goal.

The number of nodes to be added depends on the time at which the fault is detected, expected completion time of the job, the number of chunks yet to be processed and the number of nodes involved in the job. When the number of map tasks to be executed is more than the number of slave nodes available, then multiple map waves are executed. Dynamic resource scaling can help as a recovery technique for an executing job, only if at least one or more map and reduce waves are yet to be started.

The scaling heuristic that is a part of FMR needs to add sufficient number of nodes to reduce execution time penalty. Both tasks that have already completed on the faulty node and future tasks that would have executed on that node need to be executed on newly added nodes. This condition is expressed in Equation (1) where $mapProgressPercentage$ is retrieved from the Hadoop runtime using a built-in API.

$$N_{nodes\_added} = ceil\left(\frac{1}{1 - MapProgPercentage} + 1\right) \quad (1)$$

After determining the optimal number of nodes to be used for scaling (using Eq. 1), we determine the associated cost of these resources ($costOfScaling$). In order to determine whether the cost of scaling would be justified, we use Map-Reduce execution time prediction models to estimate job duration in the presence of a node fault ($execTime_{fault}$).

In our previous work [27], we have shown that Map-Reduce execution times can be estimated using machine-learning based regression models ($PerfModel$). We showed that 4 techniques, namely gaussian process regression, regression by discretization, multilayer perceptron and model trees, achieved best performance for predicting Map-reduce job execution time. Average prediction errors obtained were less than 12%. Out of these models, model trees were chosen for use in the experiments in this paper.

Using this execution time, we calculate the potential cost ($delayPenalty$) associated with exceeding the job deadline. Any user-defined cost function ($CostModel$) can be used here. The cost for the execution time penalty is compared with the cost for resource scaling, and scaling is invoked if it provides a cost benefit. This functionality of FMR is described as pseudocode in Figure 6.

```
1: $execTime_{nofault} = PerfModel(NumFaults = 0)$
2: $execTime_{fault} = PerfModel(NumFaults = 1)$
3: $delayPenalty = CostModel(execTime_{nofault}, execTime_{fault})$
4: $N_{nodes\_added} = ceil\left(\frac{1}{1-MapProgPercentage} + 1\right)$
5: $costOfScaling = nodesToScale * costPerNode$
6: if $costOfScaling < delayPenalty$ then
7:     Invoke scaling operation
8: end if
```

Fig. 6. Pseudocode of the scaling heuristic in FMR

IV. Implementation of Fault-Managed Map-Reduce

The various components of FMR that together constitute the MAPE control loop are illustrated in Figure 7 and described in this section.

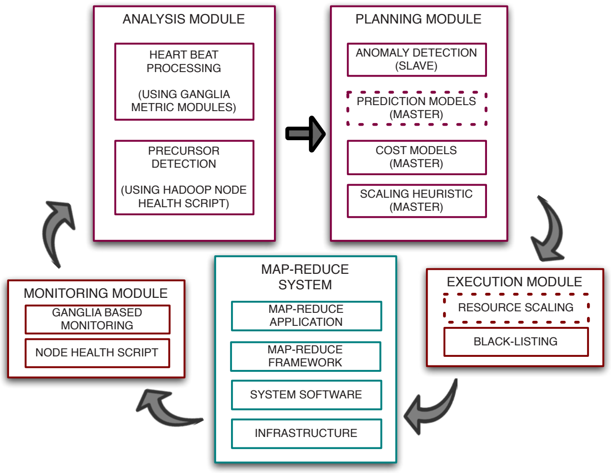


Fig. 7. Autonomic control loop of Fault-managed Map-Reduce showing a high-level overview of the monitoring, analysis, planning and execution modules. Contributions described in this paper are shown within solid-outline blocks. Contributions from prior work that are used in FMR are shown within dashed-outline blocks.

Monitoring using Ganglia: Ganglia is an open-source project [2] that provides a flexible monitoring framework for distributed systems. In FMR, customized metrics are added to Ganglia for calculating the heart beat rate and for performing anomaly detection.

Node Health Script: The node health script is a feature provided in Hadoop that allows for a pre-defined health script to be periodically executed on each slave node. As soon as a node anomaly is conveyed to the master through Ganglia, the node is *black-listed* by FMR. Black-listing immediately prevents any more tasks from being scheduled on that node. Typically, slave node faults are detected by the master only after a timeout interval. The advantage of blacklisting is that the master is made to become aware of a slave's degrading health status immediately. This is beneficial since a recovery action can be invoked without any delay. We also configure the node health script to check for other fault precursors such as task and daemon crash faults. This precursor detection functionality helps detect some crash faults that are not preceded by anomalous conditions (that could be detected by the sparse coding technique).

Anomaly Detection: The anomaly detection module is invoked at the end of each task. The input feature vector to the anomaly detection module is a heart beat wave that corresponds to the last-completed task. The sparse coding technique is implemented in Matlab and converted to a standalone executable which is then executed on the slaves using the Matlab Compiler Runtime (MCR) environment.

Recovery through Dynamic Resource Scaling: Once an anomaly is detected, the anomalous node is forcefully blacklisted. Then the scaling heuristic is executed to determine the number of nodes to be added. We use a cost model in which dollar costs are associated with different penalty ranges. Virtual machine images for the new slaves are pre-set with the master IP address. TaskTracker and DataNode daemons are started up on the new nodes, which then become a part of the executing Map-Reduce job.

V. Experimental Evaluation

Experimental Testbed: The test bed used to evaluate the FMR approach consists of 16 IBM blade servers (HS22) mounted on two different racks. Each physical node has a 8-Core Xeon 2.4 GHz CPU and 24 GB of RAM and runs CentOS 5.5 with Xen 3.4.3. The two racks are linked together by a Gigabit Ethernet network. Each physical node hosts five guest virtual machines. This guest VM (which forms a Hadoop slave node) runs Ubuntu 10.04.2 and is configured with a single core and 2Gb of RAM. Hadoop version 0.20.203 is used.

Map-Reduce applications: Applications from the Hadoop distribution and the PUMA benchmark suite [4] were used and are described below:

1) Wordcount (WC): Map outputs a *(word, 1)* key-value pair for each word in a document. Reduce combines the count for each word producing a *(word, wordcount)* pair.
2) Grep (GR): Map searches for a pattern in the input documents and produces *(pattern,1)*. Reduce combines the count for each pattern producing *(pattern, patterncount)*.
3) Pi estimation (PI): Estimates the value of Pi using quasi-Monte Carlo method.
4) Inverted index (II): Map generates the document index for each word as *(word, document index)*. Reduce combines all occurrences of a word to produce *(word, list of document indices)*.
5) Term vector per host (TV): Determines frequently occurring words in a document. Map produces *(host, termvector)* for each host. Reduce combines term vectors for each host and outputs *(host, list(termvector))*.
6) Histogram ratings (HR): Generates a histogram of movie ratings from a dataset of user reviews. Map produces *(rating, 1)* for each user review. Reduce combines the count to produce *(rating, count)*.

Input dataset: Dataset used for WC, GR, II and TV consists of books from Gutenberg [5] with size varying between 5GB to 20GB. PI does not require any input data. Input for HR is generated using scripts from PUMA.

Job duration: Performance penalties are low for long-running jobs that execute on a large number of nodes. However, long running jobs are not the common case for Hadoop as seen from two production traces that were analyzed in [14], [28]. In these studies, the average length of a job varies between few tens of seconds to few tens of minutes. The average Map-Reduce job size at Google [17] varied between 395 to 874 seconds over a period of three years between 2004 and 2007. FMR and its evaluation experiments thus focus on short jobs with runtimes ranging between 300 to 600 seconds which correspond to the majority workload in production environments.

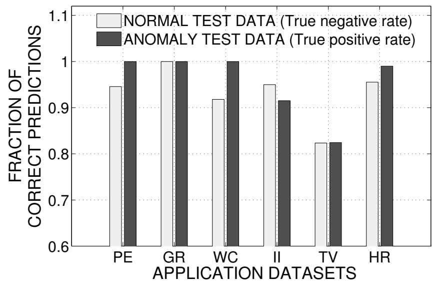
Faultload: The following fault conditions were injected into slave nodes:

1) CPU hog: A CPU-intensive sequence of matrix multiplication operations.
2) Memory hog: A sequence of memory leaks programmed into an executing matrix multiplication process.
3) Disk hog: The linux *dd* command used to copy large chunks of data between two disk partitions.
4) Node crash fault: The linux *kill* command used to terminate the TaskTracker and DataNode daemon processes running on the node.

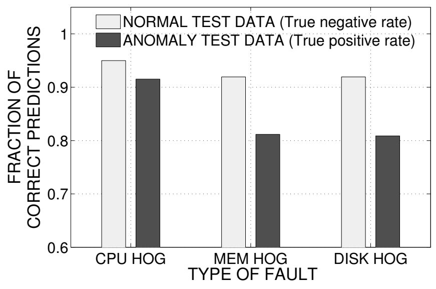
Each fault experiment consists of loading HDFS with the input, starting FMR scripts and the Map-Reduce job and then injecting faults at pre-specified time instances. Node crash faults are preceded by performance faults. After each fault experiment, HDFS is reformatted and reloaded with input data. This ensures that any non-uniformity in data distribution and replication is eliminated for each new experiment. A set of 3000 job executions were performed for validating anomaly detection, performance prediction and resource scaling components of FMR and are described in the following subsections.

A. Anomaly Detection

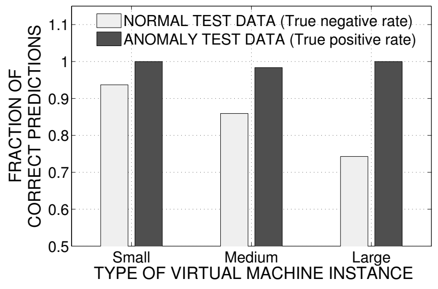
The experiment shown in Figure 8 is used to illustrate the operation of the anomaly detection module. A Wordcount Map-Reduce job is executed with a CPU hog injected into one node. We note that the anomalous slave node 'Dom-13' (in the fourth plot from the top) was correctly identified. In accordance to the goal of *early* fault detection, the fault was detected at the end of the first application heart beat wave and is marked in using an arrowhead.



(a) Different Map-Reduce applications



(b) Different types of faults



(c) Different virtual machine instances

Fig. 9. Evaluation of sparse coding based anomaly detection for (1) different Map-Reduce applications, (b) different faulty conditions, and (c) different VM instance sizes in a heterogeneous environment.

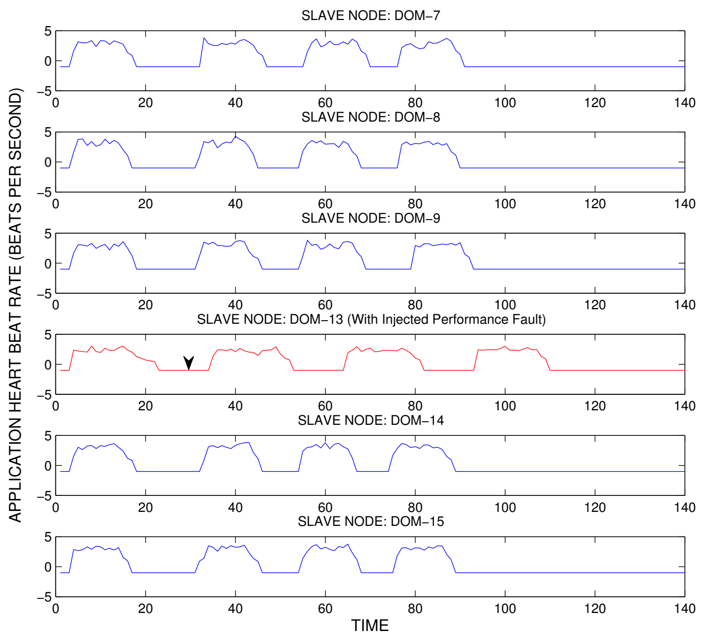
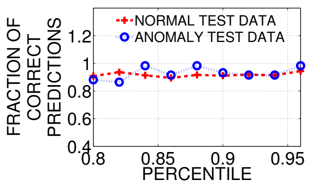


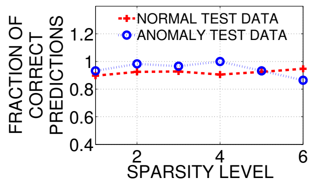
Fig. 8. Application heart beat waves for a 6-node Hadoop job in which a CPU hog process was injected into one slave node ‘Dom 13’. The anomalous node is shown in the fourth plot (from the top). An arrow head marks the time of detection of the anomaly on this node.

Figure 9(a) shows the fraction of correct predictions for the normal and anomaly test data for six datasets corresponding to six different Map-Reduce applications. The fraction of correct predictions in the anomaly dataset is the the True Positive Rate (TPR) $= \frac{TP}{FN+TP}$; while the fraction of correction predictions in the normal dataset is the True Negative Rate (TNR) $= \frac{TN}{TN+TP}$. Here TP, TN, FP, FN stand for True Positives, True Negatives, False Positives and False Negatives respectively. In this context correctly detecting an anomaly is termed a True Positive. We see that for all the datasets the TNR is greater than the theoretical bound of 0.8 that was chosen for the percentile parameter. This corresponds to a maximum False Positive Rate of 0.2.

In Figure 9(b), we plot the TPR and TNR for the inverted index application for different injected faults. In order to identify the cause for variation in performance, we compare the intensity of the effect (performance penalty) of each these faults. A CPU hog, memory hog and disk hog causes 22%, 13% and 11% increase in average execution time. The CPU



(a)



(b)

Fig. 10. Sensitivity analysis: Variation of the fraction of correct predictions for normal test data (true negative rate) and anomaly test data (true positive rate) using the sparse coding anomaly detection technique for different values of the (a) ‘Percentile’ parameter and (a) ‘Sparsity Level’ parameter. Application: Term vector per host. Injected fault: CPU hog.

hog with maximum penalty is a more severe fault and hence can be detected with better performance. The memory hog and disk hog effects are more subtle and hence appear to result in slightly lesser performance.

The time taken for prediction computation and model training is shown in Table I for 3 application datasets. We see that for the largest dataset of 1400 instances, training takes 1 sec and testing takes only 0.008 secs. This ensures minimal overhead when our anomaly detection model is used within FMR.

TABLE I
TRAINING AND TEST DURATIONS FOR ANOMALY DETECTION

Dataset	Application	Total Instances	Training Duration	Testing Duration
1	Pi Estimation	1497	1.06 secs	0.008 secs
2	Grep	202	0.13 secs	0.008 secs
3	Wordcount	400	0.22 secs	0.008 secs

Heterogeneity: The use of decentralized, local models for anomaly detection enables us to extend FMR to work in a heterogenous environment. A heterogeneous testbed was configured consisting of three different virtual machine instance types: ‘small’ VMs with 1 CPU and 2GB of RAM, ‘medium’ VMs with 2 CPUs and 4GB RAM and ‘large’ VMs with 4CPUs and 6GB of RAM. Performance of anomaly detection for each VM instance type in this environment is shown in Figure 9(c).

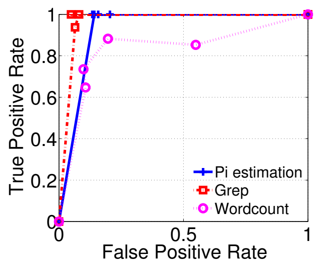
Sensitivity Analysis: In order to determine the effect of choosing different parameters, we perform a sensitivity analysis of two parameters, namely the sparsity level in Figure 10(a) and percentile value in Figure 10(b). Sparsity level is varied between 1 and 6 and the percentile parameter (which is related to $1 - \beta$) is varied between 0.8 and 0.96. The effect of these variations on TPR and TNR is plotted. We see that anomaly detection performance is quite stable within these ranges, thereby providing sufficient leeway in choosing good parameter values. We note that although anomaly data is not needed for training, it can be leveraged when available for parameter tuning.

Receiver Operating Characteristic curves: We plot ROC curves for 6 applications in Figure V-A by varying the confidence threshold between 0 and 1. All curves are close to the upper-left corner, where TPR is high and FPR is low. In addition, most of the curves provide a number of possible values of confidence threshold (i.e. points on the curve with markers) in the upper left corner region indicating that good performance is possible for many confidence threshold values.

TABLE II
COMPARISON OF ANOMALY DETECTION TECHNIQUES

Application	Multilayer Perceptron	K-means clustering	Support Vector Machines	Sparse-coding
	True positive rate / True negative rate			
PI	1.0/0.96	0.99/0.88	0.76/0.3	1.0/0.93
GR	1.0/0.94	0.7/0.31	1.0/0.65	1.0/1.0
WC	0.98/0.88	0.96/1	1.0/0.69	1.0/0.92
II	0.99/0.95	0.8/0.66	1.0/0.49	0.92/0.95
TV	0.95/0.76	0.93/0.69	0.89/0.49	0.82/0.82
HR	0.99/0.98	0.975/0.99	1.0/0.51	0.99/0.96

Comparison of anomaly detection techniques: We compare sparse coding with 3 classification techniques in Table II. Multilayer perceptron provides best performance. However, it needs anomaly data for training and takes ten to a hundred seconds for training, thus making in unsuitable for FMR. K-means clustering also needs anomaly data for training and does not achieve very good TPR and TNR values. Single-class SVM does not need anomaly data for training, making it a viable candidate. However, sparse coding models achieve much better performance for all 6 benchmark applications.



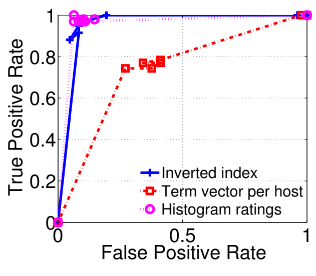


Fig. 11. ROC curves showing performance of anomaly detection as the confidence threshold is varied. Six benchmark applications are shown in 2 separate plots (a) and (b) for clarity.

B. Dynamic Resource Scaling

We first evaluate the accuracy of model tree prediction, which is a critical component of the FMR control loop. Prediction accuracy for 6 test jobs is shown in Figure 12 and was an average of 11.1%. Features used include number of

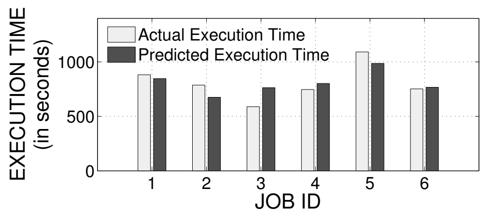


Fig. 12. Prediction accuracy of the model tree algorithm used for Map-Reduce job execution time prediction. Application: Wordcount. Jobs 1, 2, 3 have one fault injected, while jobs 4, 5 and 6 are fault-free.

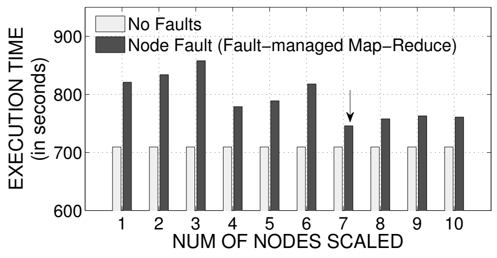


Fig. 13. Evaluation of FMR scaling heuristic. Application: Pi estimation. Fault: Node crash fault at 520 seconds. FMR scaling heuristic scales by 7 nodes (marked with an arrow in the plot)

slaves, dataset size, time of fault, number of faults, timeout and Hadoop framework configuration parameters. The model tree implementation in the Weka tool suite was used for training and testing.

Next we evaluate the scaling heuristic in Figure 13 to determine whether sufficient number of nodes are chosen for scaling. For the job shown, the heuristic scales by 7 nodes based on the map progress percentage of 0.83 in Eq (1). We manually scale by 1 to 6 nodes and 8 to 10 nodes, to determine if 7 is the right choice. We see that for $N_{nodes\_added} < 7$, penalty is $> 5\%$ and for $N_{nodes\_added} > 7$ there is no additional benefit.

The penalty reduction through dynamic resource scaling is illustrated using a *swimlane* plot in Figure 14. In this plot, each y-axis coordinate corresponds to the execution of a map task in a single map-slot. Our experiments use Hadoop's default setting of two slots per node. So a pair of consecutive lines (parallel to the x-axis) correspond to two map tasks running simultaneously on a node.

Figure 14 (a) shows a Map-Reduce job that consists of four map waves. The job did not experience any faults. In Figure 14 (b), the same job is rerun with a CPU performance fault injected into one of the nodes. The presence of a fault results in an execution time penalty of 18.5%. In the next execution of the same job, FMR scripts are enabled. Figure 14 (c) shows the addition of two nodes to the running job after detection of the anomalous node. We note that with the help of resource scaling, performance penalty is reduced to 4.6%.

In Figure 15, FMR is compared with Hadoop's built-in speculative execution. After a job begins execution, a CPU hog process is injected and is followed by a node crash fault after 30 seconds. We see that with speculative execution, penalty is not reduced. However using the FMR approach, through

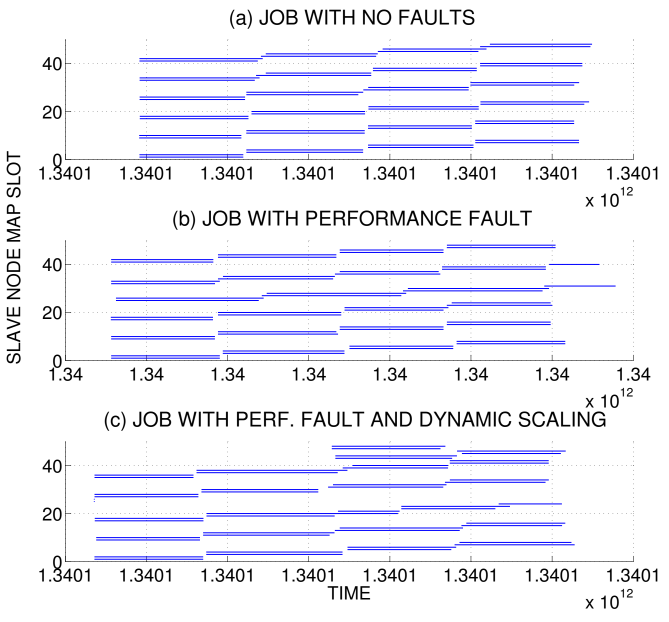


Fig. 14. Swimlane plots of three Map-Reduce jobs. (a) Job with no faults (b) Job with injected CPU performance fault (Execution time penalty of 18.5%) and (c) Job with injected performance fault and dynamic resource scaling enabled (Execution time penalty reduced to 4.6%).

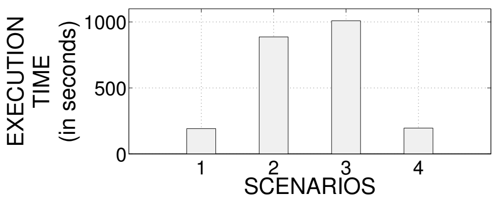


Fig. 15. Comparison of FMR with speculative execution. (1) Job with no faults (2) Job with fault and speculative execution turned off. (3) Job with fault and speculative execution turned on. (4) Job with faults managed using FMR approach. Application: Inverted Index. Fault: CPU hog process + node crash fault after 30 seconds.

scaling by 4 nodes, the penalty is decreased to $< 5\%$ of the fault-free execution time.

In the next set of experiments, node crash faults are injected into a running job executing on different numbers of nodes. As shown in Figures 16(a) to 16(f), we see that FMR consistently helps in mitigating performance penalty. FMR helped decrease performance penalty from an average of 119% to 14% across these 6 sets of experiments.

C. Discussion

Virtualized environment: FMR has been implemented on a virtualized environment which easily provides the actuators needed for dynamic resource scaling. However, a non-virtualized environment can also use FMR by provisioning extra resources that can be added to the Hadoop cluster on-demand. These extra resources can be utilized for executing preempt-able jobs during those periods when they are not utilized as part of recovery. However, a virtualized environment provides the capability to extend recovery to other actions such as migration (to handle hardware faults) and scaling up (to handle resource exhaustion faults).

Application-specific anomaly detection models: The anomaly detection model developed is specific to a Map-Reduce application because each application has different heartbeat characteristics. We believe that it is reasonable to manage application-specific models because typical Map-Reduce workloads involve the execution of the same job on gradually evolving data sets. Recent literature shows that 80% of jobs in a workload from Yahoo! were repeated at least 50 times [11]. The feature vectors needed for training the sparse coding model are derived from one application heart beat wave that corresponds to the processing of one data chunk by a map task. Hence, even a single MapReduce job can provide few tens to a few hundred feature vectors for training.

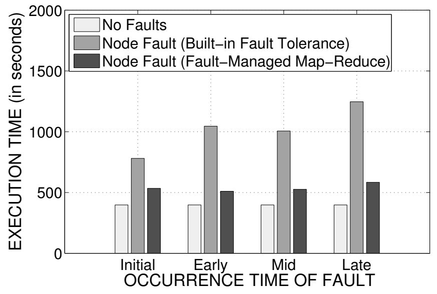
Scalability of FMR: Since anomaly detection is performed by local, decentralized models at a node, the associated overheads are local to a node. Therefore, the overhead does not increase adversely as the number of slaves is increased. The computation performed at the master (by FMR) for each slave is limited and only consists of evaluating two binary metrics for each slave (namely the presence/absence of an anomaly and the result of the node-health script). The latest version of Hadoop, called 'NextGen' Hadoop uses distributed masters and will further help reduce time taken for this evaluation. Furthermore, FMR aims at achieving soft deadlines for Map-Reduce jobs. In a typical shared Map-Reduce cluster only a subset of the jobs would have these soft-deadline requirements. Thus, FMR needs to be enabled only for these jobs, thus avoiding the need to monitor and manage all jobs.

VI. Conclusions

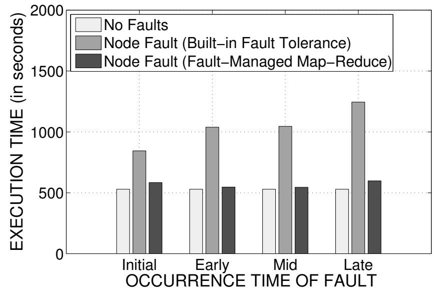
Map-Reduce has become an important platform for a variety of data processing applications. Built-in fault-tolerance mechanisms in Map-Reduce frameworks such as Hadoop, suffer from performance degradations in the presence of faults. Fault-managed Map-Reduce, proposed in this paper provides an *online*, *on-demand* and *closed-loop* solution to managing these faults. The control loop in FMR mitigates performance penalties through early detection of anomalous conditions on slave nodes. Anomaly detection is performed through a novel sparse-coding based method that achieves high true positive and true negative rates and can be trained using only normal class (or anomaly-free) data. The local, decentralized nature of the sparse-coding models ensures minimal computational overhead and enables usage in both homogeneous and heterogenous Map-Reduce environments. After an anomalous condition is detected, dynamic resource scaling, through the proposed scaling heuristic, is invoked as the recovery action. Through extensive evaluation of a variety of benchmark applications on a 72-node Hadoop cluster, we show that FMR can effectively mitigate performance penalties.

VII. Acknowledgements

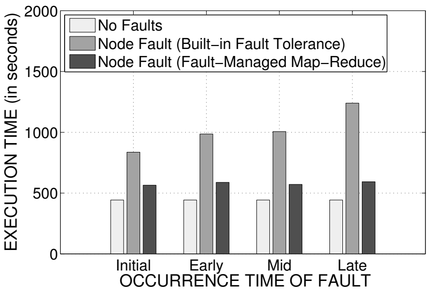
The authors would like to thank Dr. Nagarajan Kandasamy for helping shepherd our paper. This work is supported in part by the National Science Foundation grants No. OCI-0910812 and No. 0758596 and the AT&T Foundation.



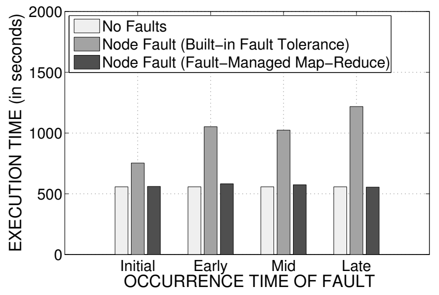
(a) 12 node cluster + 10 nodes for scaling



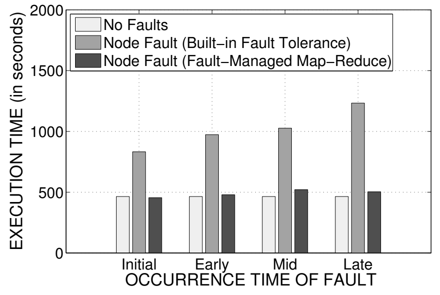
(b) 22 node cluster + 10 nodes for scaling



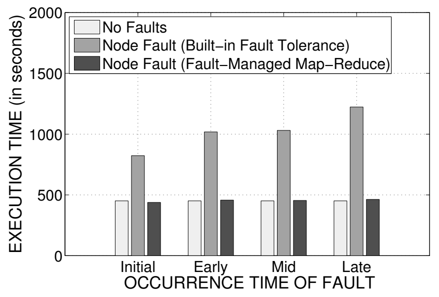
(c) 32 node cluster + 10 nodes for scaling



(d) 42 node cluster + 10 nodes for scaling



(e) 52 node cluster + 10 nodes for scaling



(f) 62 node cluster + 10 nodes for scaling

Fig. 16. Comparison of job execution times of a Pi Estimation job in the presence of a node crash fault through the use of Hadoop's built-in fault tolerance and FMR. x-axis labels 'Initial', 'Early', 'Mid' and 'Late' correspond to a node crash fault injected at 1 sec, 120 sec, 220 sec and 320 sec from job start.

REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] Ganglia Monitoring Tool. http://ganglia.sourceforge.net/.

[3] Jeff Dean. http://tinyurl.com/87kgcev.

[4] Purdue MapReduce Benchmark Suite. http://tinyurl.com/bn5gmga.

[5] Gutenberg. http://www.gutenberg.org/, 2009.

[6] Jay Kreps. http://tinyurl.com/cu24pwz, 2009.

[7] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. of OSDI*, 2010.

[8] K. Bare, S. P. Kavulya, J. Tan, X. Pan, E. Marinelli, M. Kasick, R. Gandhi, and P. Narasimhan. Asdf: an automated, online framework for diagnosing performance problems. 2010.

[9] L. A. Barroso and U. Hlzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2009.

[10] D. Borthakur and et al. Apache hadoop goes realtime at facebook. In *2011 ACM SIGMOD Intl. Conf. on Management of Data*, 2011.

[11] E. Bortnikov, A. Frank, E. Hillel, and S. Rao. Predicting Execution Bottlenecks in Map-Reduce Clusters. In *HotCloud*, 2012.

[12] E. S. Buneci and D. A. Reed. Analysis of application heartbeats: learning structural and temporal features in time series data for identification of performance problems. In *Proc. of Supercomputing*, 2008.

[13] E. Candes and T. Tao. Decoding by linear programming, 2004.

[14] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *MASCOTS*, 2011.

[15] Y. Cong, J. Yuan, and J. Liu. Sparse reconstruction cost for abnormal event detection. In *Proc. of CVPR*, 2011.

[16] X. G. Daniel Dean, Hiep Nguyen. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proc. of ICAC*, 2012.

[17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[18] F. Dinu and T. E. Ng. Understanding the effects and implications of compute node related failures in hadoop. In *Proc. of HPDC*, 2012.

[19] D. L. Donoho, Y. Tsaig, I. Drori, and J. luc Starck. Sparse solution of underdetermined linear equations by stagewise orthogonal matching pursuit. Technical report, 2006.

[20] M. Elad. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer, 2010.

[21] Y. C. Eldar and M. Mishali. Robust recovery of signals from a structured union of subspaces. *IEEE Trans. Inf. Theor.*, 55(11), 2009.

[22] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *EuroSys*, 2012.

[23] M. Gabel, A. Schuster, R.-G. Bachrach, and N. Bjorner. Latent fault detection in large scale services. In *Proc. of DSN*, 2012.

[24] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SOCC*, 2011.

[25] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application Heartbeats. In *Proc. of ICAC*, 2010.

[26] S. Kadirvel and J. Fortes. Towards self-caring mapreduce: Proactively reducing fault-induced execution-time penalties. In *HPCS*, 2011.

[27] S. Kadirvel and J. Fortes. Grey-box approach for performance prediction in map-reduce based platforms. In *Proc. of ICCCN*, 2012.

[28] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *CCGRID*, 2010.

[29] P. Lama and X. Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *ICAC*, 2012.

[30] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: BlackBox diagnosis of MapReduce systems. *SIGMETRICS Perform. Eval. Rev.*, 37(3), Jan. 2010.

[31] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. of FAST*, 2007.

[32] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance comp. systems. *Trans. on Dep. and Sec. Comp.*, 2010.

[33] B. Schroeder and G. A. Gibson. Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *Proc. of FAST*, 2007.

[34] J. Tan, X. Pan, E. Marinelli, S. Kavulya, R. Gandhi, and P. Narasimhan. Kahuna: Problem diagnosis for Mapreduce-based cloud computing environments. In *Proc. of NOMS*, 2010.

[35] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *Proc. of ICDCS*, 2012.

[36] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

[37] A. Verma, L. Cherkasova, and R. H. Campbell. Resource provisioning framework for mapreduce with performance goals. In *Middleware*, 2011.

[38] G. Wang, A. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *MASCOTS*, 2009.

[39] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proc. of OSDI*, 2008.

[40] B. Zhao, L. Fei-Fei, and E. P. Xing. Online detection of unusual events in videos via dynamic sparse coding. In *Proc. of CVPR*, 2011.