J Supercomput (2016) 72:1570–1596
DOI 10.1007/s11227-016-1682-2

# Soft error resilience in Big Data kernels through modular analysis

**Sui Chen**[1] · **Greg Bronevetsky**[2] · **Lu Peng**[1] · **Bin Li**[3] · **Xin Fu**[4]

Published online: 3 March 2016


**Abstract** The shrinking processor feature and operating voltages of processor circuits are making them increasingly vulnerable to soft faults, which calls for fault resilience techniques at both the software and hardware levels under the big data context. To assist software developers in writing fault-resilient big data applications, we propose the tool ErrorSight, which helps them to focus their efforts on code regions and data structures that are most vulnerable to soft errors, understand how numerical errors propagate through the program, and apply fault resilience techniques effectively. ErrorSight achieves this through efficient generation of error profiles leveraging the predictive power of the Boosted Regression Tree model. We use four big data kernels to illustrate the modular analysis mechanism of ErrorSight and show its usefulness in the development of numerical fault-resilience in Big Data.

**Keywords** Soft faults · High-performance computing · Numerical errors · Fault resilience · Big data

## 1 Introduction

As the feature sizes of processor circuits shrink and operate at lower voltages they become increasingly vulnerable to soft faults, which are transient interruptions in

✉ Lu Peng
lpeng@lsu.edu

1 Division of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, USA

2 Lawrence Livermore National Laboratory, Livermore, USA

3 Department of Experimental Statistics, Louisiana State University, Baton Rouge, USA

4 Department of Electrical and Computer Engineering, University of Houston, Houston, USA

the correct operation of individual gates or circuits [3]. Soft faults may be caused by a wide range of physical phenomena, including voltage variation, as well as the impact of cosmic ray-induced neutrons or alpha particles from chip packaging. The transient electric fields induced by these phenomena distort the flow of charge in transistors and temporarily corrupt the state of the circuits they use them [4]. These corruptions may propagate through processor logic to corrupt the state of the applications that execute on them, causing them to crash or silently return incorrect results. Today the error rates in DRAMs have been reported to reach 70,000 FITs (failures per billion device hours) per Mbits [18]. Further, since by the year 2020 processor feature sizes are expected to shrink to just 5–7 nm (DRAM $\frac{1}{2}$ pitch), which is just 10–14 silicon atoms (5 Å per atom) across, soft faults are expected to become a critical problem in these designs and the software that runs on them. This makes it important for software developers to design ways to enable system and application software to survive the impact of soft faults with minimal cost in power use and performance.

Prior work in various application domains has demonstrated that it is possible to build hardware and software that are resilient to soft faults. They range from mechanisms that are generic and expensive (e.g. replication) to application-specific techniques that have a low runtime overhead but require significant development effort [9,11]. For general techniques, the user has to decide smartly when and where to deploy them effectively at a reasonable cost. For algorithm-specific techniques, the user needs to gain a thorough understanding of the algorithm in question to develop fault resilience techniques. Both would require significant amount of efforts.

This paper focuses on the problem of reducing the amount of work needed for application developers to design and deploy resilience techniques by presenting a support tool named ErrorSight for this task. Similar to performance analysis tools that quantify the resource utilization of various application regions, ErrorSight helps developers understand the impact of soft faults on their application state and how their impacts flow through application logic.

Developers can use this information to:

- Focus their efforts on code regions and data structures the errors in which have the most significant impact on application results, and
- Understand how the errors propagate as the program runs,

and produce fault-resilient software more efficiently.

ErrorSight begins by running fault injection plans, where the program is executed a large number of times (Table 5) with one error modelling physical soft faults injected into its program state each time. The errors are injected into registers and can propagate through expressions and memory operations as the program continues. By observing the flow of these errors through application state and their impact on application output, ErrorSight creates a profile of the errors that have the most significant impact on application output and how they propagate to the output. As the data begin to accumulate, a statistical model is trained with the data in hand, which is then cross-validated. This model captures the error propagation patterns and can predict the magnitude of errors down through the propagation chain, thereby saving the cost of fault injection experiments needed for an accurate characterization of the impact of

soft errors on this program. The results are then presented to the user in an intuitive way, informing the user of the necessary changes in the software needed to improve fault resilience.

While fault injection is used ubiquitously to quantify application resilience properties, ErrorSight incorporates novel capabilities specifically designed to improve developers' ability to make their software resilient in addition to evaluate resilience after the fact. First, ErrorSight quantifies the impact of errors on application state in terms of high-level concepts using developer-specified distance metrics to measure the difference between a given data structure in a fault injected run and the same data structure at the same execution point in a reference application run. For example, errors in numerical vectors may be quantified using the root mean square deviation metric, while errors in strings may be measured using the edit distance metric. This enables developers to reason about the impact of errors at the same level of abstraction they use as part of their regular development efforts, which improves their productivity. Second, ErrorSight tracks the propagation of errors through each fault injected execution to make it possible for developers to query where the errors that most critically affect application outputs originate from, and how they flow through application logic. This enables developers to design resilience techniques that detect the most critical error types (1) soon after they occur, (2) at application locations that are highly sensitive to errors (e.g. control logic), or (3) at application locations where errors may be easily identified (e.g. where critical errors induce usually large values in some application variable). Finally, ErrorSight statistically models the propagation of errors from the inputs to the outputs of individual code regions to (1) enable application developers to understand application resilience properties in a modular fashion (e.g. important for library writers and developers of large applications) and (2) reduce the number of fault injection experiments needed to comprehensively analyze an applications resilience properties. To ensure that developers can make well-grounded conclusions based on these models ErrorSight reports confidence intervals for all model predictions.

Overall, the main contribution in this paper are:

- We developed the tool ErrorSight, which provides useful guidance for the user in writing fault-resilient software,
- We developed an algorithm that can substantially reduce the cost of fault injection experiments,
- We observed three kinds of error propagation patterns, namely "maintaining", "shrinking" and "magnifying", and
- We demonstrated the usage of ErrorSight and showed how to apply fault resilience to one big data kernel.

The paper is organized in a way that follows the workflow of ErrorSight. Sect. 3 describes the design, from the error model to error propagation and the error characterization algorithm. Section 4 discusses the Big Data kernels and the driver programs used in this paper and gives an analytical analysis of the error propagation patterns that will be corroborated with results in Sect. 5. We complete the paper by showing how to add fault resilience using ErrorSight in Sect. 5.5.

## 2 Related works

ErrorSight complements the broad range of existing work done by software resilience community. It can take advantage of existing fault injection tools such as NFTAPE [19] and KULFI [1], as well as recent approaches such as Relyzer [12] that leverage redundancy in the way different errors propagate to reduce the number of fault injections needed to comprehensively understand the impact of errors on applications.

ErrorSight supports developer efforts to design and deploy resilience mechanisms. This includes the use of generic mechanisms such as redundancy [6] and OS segmentation violation detection, as well as application-level techniques [8,9], both of which require tools to quantify the flow of errors though key application sub-routines (e.g. GMRES solver for Elliot et al. [9] and LU factorization for Du et al. [8]).

Finally, ErrorSight can be incorporated into emerging resilience-aware programming models such as Containment Domains [7], which enables application developers to organize their resilience mechanisms hierarchically. In this context ErrorSight can serve the same role as debuggers and performance analyzers do in traditional programming models.

## 3 Design of ErrorSight

ErrorSight performs fault injection campaigns, tracks the execution of a program and log its program states, builds a non-parametric tree-based predictive model named Boosted Regression Tree to predict the error propagation in the program, and then obtains the error characteristics of an application at the source code level. This section introduces each of these steps in order, as is illustrated in Fig. 1.

### 3.1 Error model

We use the KULFI [1] error injection framework throughout the experiments, which is based on LLVM [13]. It uses a static single assignment (SSA) compilation strategy

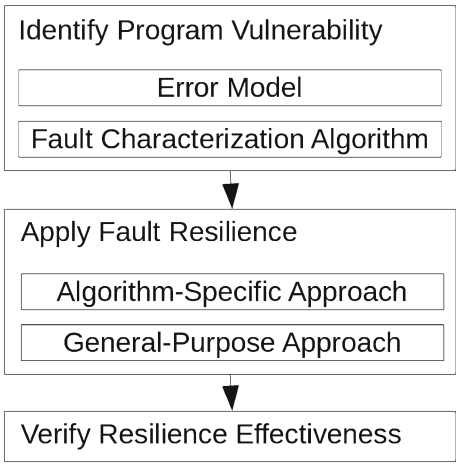



**Fig. 1** Workflow of ErrorSight

**Table 1** Example of static fault site to source code mapping

| Source Code | ID | Instruction (Static Fault Site) |
|---|---|---|
| `120: for (i = 0; i < nlocal verts; ++i) pred[i] = -1` | 4 | `%51 = load i64* %i, align 8` |
| | 5 | `%52 = load i64** %3, align 8` |
| | 6 | `%53 = getelementptr inbounds i64 %52, i64 %51` |
| | 7 | `store i64 -1, i64* %53, align 8` |

**Table 2** Example trace of K-means. A bitflip is injected at the 10054576'th dynamic fault site at iteration 2 and propagates through iteration 6

| Run ID | Dynamic FSID | Bit ID | Static FSID | Is Init | Num iter | Error metric |
|---|---|---|---|---|---|---|
| 3 | 10054576 | 1 | 78 | 1 | 2 | −4.667642 |
| 4 | 10054576 | 1 | 78 | 0 | 3 | −6.372848 |
| 5 | 10054576 | 1 | 78 | 0 | 4 | −7.477853 |
| 6 | 10054576 | 1 | 78 | 0 | 5 | −8.271486 |
| 7 | 10054576 | 1 | 78 | 0 | 6 | −9.056564 |

which is capable of supporting arbitrary programming languages. The source code is compiled into LLVM byte code representing the LLVM instruction set. As a result, there is a one-to-many mapping between the entities in the source code (statements, expressions) and the instructions, as is illustrated in Table 1.

The SSA semantics determine that instructions producing outputs write to at most one register. We consider bit flips in these output registers and define one instruction in the program image to be a *static fault site*. A dynamic instance of a static fault site is defined as a *dynamic fault site*. There is a one-to-many mapping between static and dynamic fault sites.

For each run in the fault injection campaign, a bit in a dynamic fault site is chosen for fault injection. We only inject one bit flip per run because multi-bit flip events are relatively rare.

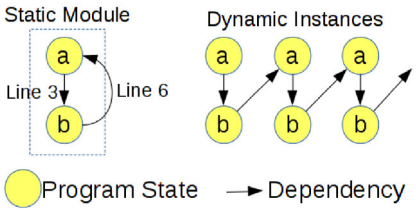
To quantify and track the propagation of errors, the intermediate results and data structures are compared with those of a fault-free run. For the applications in this paper, error metrics defined in Table 4 are measured for the entities of interest at run time. Table 2 is an example trace form the K-means program.

### 3.2 Modular analysis of a program

The modular analysis of a program is the theoretical basis the regression model is built upon. A program consists of "entities" including values and expressions, operated on by "modules" which read and store data and perform computation. A modular structure of an application is a graph consisting of nodes representing "entities" and arcs representing "routines." The graph may be considered a coarse version of the data

**Fig. 2** Example of a module

```
1   for (int iter=0; iter<10; iter++) {
2      for (int i=0; i<10; i++) {
3         b[i] = a[i] + a[i+1] + f(b[i]);
4      }
5      for (int i=0; i<10; i++) {
6         a[i] = b[i];
7      }
8   }
```




dependency graphs generated by a compiler. To put this into perspective, consider the program in Fig. 2.

We consider the loop body to be a module consisting of two entities, arrays `a` and `b`. In Line 3 `b` is updated using values of `a`. This line corresponds to the dependency arc flowing from `a` to `b` in Fig. 2. Similarly Line 6 corresponds to the arc flowing from `b` to `a`. The dependency graph may be unrolled with entities `a` and `b` duplicated for each iteration. In the unrolled form, self-loops in the graph are to be replaced by edges between incarnations of the nodes in different iterations.

The effect the arcs have on the errors are captured by the regression model, described in Sect. 3.5.

### 3.3 Error propagation

On the microscopic level, error propagation refers to the incident of an output affected by bit flip is used as the input of other instructions.

On a higher level, we consider the following two cases in the model used in ErrorSight:

- Propagation between entities: an error originates from a bit flip, and then propagates from one entity to another entity following the arcs between them.
- Propagation between time steps: propagation between entities repeats as the main loop in the program advances. Example is the propagation from `b` to the `a` in the next iteration on Line 6 in Fig. 2.

The two types of propagation are complementary which can be used to model most data flow found in iterative applications. By including relevant inputs/outputs of modules and the time step into the set of dependent variables, we can build predictive models to predict either type of error propagation. We consider two types of models, named Model 1 and Model 2, as described in Fig. 3.

The details of the models are described in Table 3. Both models are realized using the Boosted Regression Tree model described in Sect. 3.5.

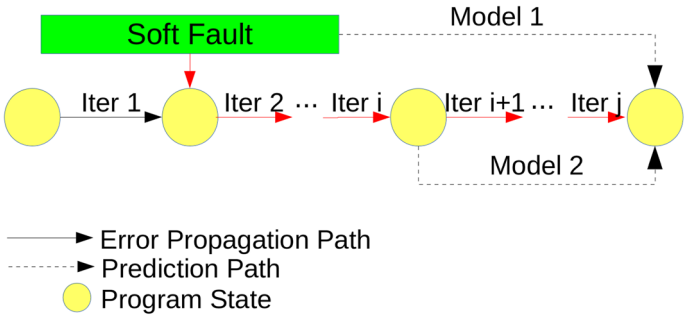



**Fig. 3** Prediction models on the error propagation path

**Table 3** Details of Model 1 and Model 2

| Model | Input | Output | Algorithms |
|---|---|---|---|
| #1 | Static/Dynamic FSID, Bit ID | Error Metric | Regression Tree |
| #2 | Error Metric at iteration $i$ | Error Metric at iteration $j$ | Linear Regression, Segmented Linear Regression or Regression Tree |

The models correspond to two cases in which we need to characterize the error of an application at iteration $j$, which may be affected by a bit flip that occurred in iterations 0 through $i$, with different input to the model in each case:

- Model 1 relates the information of a bit flip to the error in the program state. It requires the program be run to iteration $j$. This corresponds to Algorithm 1.
- Model 2 relates the error in program states in iteration $i$ and iteration $j$. It needs to observe the program states at iterations $i$ and $j$. This step is involved in Algorithm 2.

For both models, the characterization output is an aggregate of error metrics, in the form of histograms which conveniently represents the fault characteristics of the whole application. Figure 4 shows how the distribution of the error metric RMSD (Root Mean Square Distance) in the page weights of Pagerank (described in Sect. 4) changes between iterations 4 and 14. We use the Earth Mover Distance [17] (EMD) to quantify the difference between two histograms. As the name suggests, a greater distance means a greater difference in the probability masses. In Sect. 5.3 it will be used to quantify the goodness of prediction.

The two models are used to construct fault characterization algorithms.

### 3.4 Fault characterization algorithms

ErrorSight uses an efficient inter-iteration algorithm for fault characterization. It is based on the baseline fault characterization algorithm.

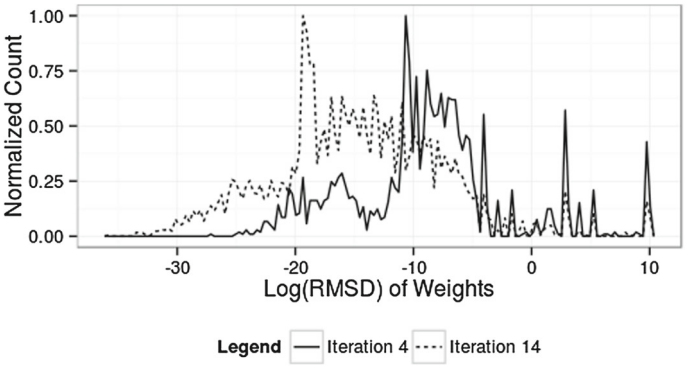



**Fig. 4** Histogram of root mean square distances in the page weights in PageRank in iterations 4 and 14

### *3.4.1 Baseline fault characterization algorithm*

**Algorithm 1**: Baseline Fault Characterization Algorithm

**Input**: Program $p$
**Result**: Empirical Error distribution at the end of iteration N

1 $rsq_{prev} \leftarrow 0$;
2 $rsq \leftarrow 0$;
3 $NF \leftarrow Number\ of\ dynamic\ fault\ sites$;
4 $num\_inj \leftarrow 1000$;
5 $errors \leftarrow \emptyset$;
6 **while** $rsq - rsq_{prev} > \epsilon$ **do**
7  **for** $fsid in (0 to NF step \frac{NF}{num\_inj})$ **do**
8  $p_{fsid} \leftarrow p\ with\ bit\ flip\ at\ dynamic\ fault\ site\ \#fsid$;
9  run $p_{fsid}$ until completion ;
10  $errors \leftarrow errors \cup error(p_{fsid})$;
11  **end**
12  $RT \leftarrow Model1(errors)$;
13  $rsq_{prev} \leftarrow rsq$;
14  $rsq \leftarrow CrossValidate(RT, errors)$ ;
15  $num\_inj \leftarrow 2 \cdot num\_inj$;
16 **end**
17 **return** $(errors)$

The baseline fault injection algorithm is listed in Algorithm 1. This algorithm incrementally increases the number of fault injection experiments until the *R*-Squared value measured from the validation step (Line 14) suggests the sample size is large enough for an accurate model. Every fault injection run has to be executed to completion in order to obtain the errors (Line 9). The number of dynamic fault sites is not directly related to the number of experiments needed.

Cross-validation is achieved by splitting the collected errors into a training set used to train the model and a test set use to evaluate the *R*-squared value. The *R*-squared

value quantifies how much the model can explain the uncertainty of the real underlying distribution of errors.

Most of the cost in Algorithm 1 is incurred by Line 9 (running program to completion after fault injection), which Algorithm 2 seeks to improve.

### *3.4.2 The efficient algorithm*

**Algorithm 2**: Inter-Iteration Efficient Characterization Algorithm

**Input**: Program $p$; Fault site counts at iteration $i$, $NF_i$ $(i \in (0, 1, 2, ..., N))$
**Result**: Error distribution at the end of iteration N

1 $NF \leftarrow Number\ of\ dynamic\ fault\ sites$;
2 $errors_1, errors_2, ..., errors_N \leftarrow \emptyset$;
3 $errors \leftarrow \emptyset$;
4 **for** $i = 0, 1, ..., N$ **do**
5  $num\_inj \leftarrow 100$; $rsq_{prev} \leftarrow 0$; $rsq \leftarrow 0$; $runs_i \leftarrow \emptyset$;
6  **while** $rsq - rsq_{prev} > \epsilon$ **do**
7  **for** $fsid = (NF_{iter}\ to\ NF_{iter+1}\ step\ \frac{NF_{iter+1} - NF_{iter}}{num\_inj})$ **do**
8  $p_{fsid} \leftarrow Program\ with\ bit\ flip\ inserted\ at\ fsid$ ;
9  $runs_i \leftarrow runs_i \cup p_{fsid}$ run $p_{fsid}$ until iteration $i$;
10  $errors_i \leftarrow errors_i \cup error(p_{fsid})$;
11  **end**
12  $m1 \leftarrow Model1(errors_i)$;
13  $rsq_{prev} \leftarrow rsq$ ;
14  $rsq \leftarrow CrossValidate(m1, errors_i)$;
15  $num\_inj \leftarrow 2 \cdot num\_inj$;
16  **end**
17  $rsq2_{prev} \leftarrow 0$;
18  $rsq2 \leftarrow 0$;
19  $n2 \leftarrow 1$;
20  **while** $rsq2 - rsq_{prev} > \epsilon$ **do**
21  $newruns \leftarrow sample(runs_i, 2 \cdot n2)$;
22  $subset \leftarrow sample(newruns, n2)$;
23  $test \leftarrow newruns \setminus subset$; $m2 \leftarrow Model2(error(subset))$;
24  run $test$ to completion;
25  $rsq2_{prev} \leftarrow rsq2$;
26  $rsq2 \leftarrow Validate(m2, test)$;
27  $n2 \leftarrow 2 \cdot n2$;
28  **end**
29  $errors \leftarrow errors \cup errors_i$;
30 **end**
31 **return** $(errors)$

In Algorithm 2, errors will be collected for every iteration (Line 2), just like in Algorithm 1. Instead of running until completion, we only selectively run the program until the end of the iteration where the error gets injected.

We confirm the number of experiments with the same validation procedure and incremental increase of sample size (Lines 13 through 13) as in Algorithm 1. A subset of the instances will be run to completion to build Model 2 (Line 21). Another subset will be run to completion to serve as the validation set validation set (Line 22). The

same validation procedure is also applied to Model 2 (Lines 25 through 27) and the subset of instances run to completion is incrementally increased. When the trained Model 2 has become accurate enough we project all the errors to the end of the program. By doing this we save the cost of having to run the rest of the instances until completion. The same procedure is repeated for all the iterations to obtain the fault characteristics of the program.

### 3.5 Boosted regression trees

We propose to use the Boosted Regression Tree method for predicting the distribution of errors at the output of modules and the propagation of modules. Being an aggregate technique that aims to providing good prediction quality by combining the predictive power of numerous weaker predictors, the Boosted Regression Tree is based on the classic Classification And Regression Tree (CART) [5] and Boosting, which builds and combines a collection of trees by penalizing erroneous predictions and preserving correct predictions.

CART is a recursive binary partitioning algorithm and is an alternative to traditional parametric models for regression problems. The term "binary" indicates it has the power to split the input space into two regions and models the response by a constant for each region. The region may be further subdivided to give a better fit of the input space. To illustrate with Fig. 5, the data set with the Dynamic Fault Site ID as the

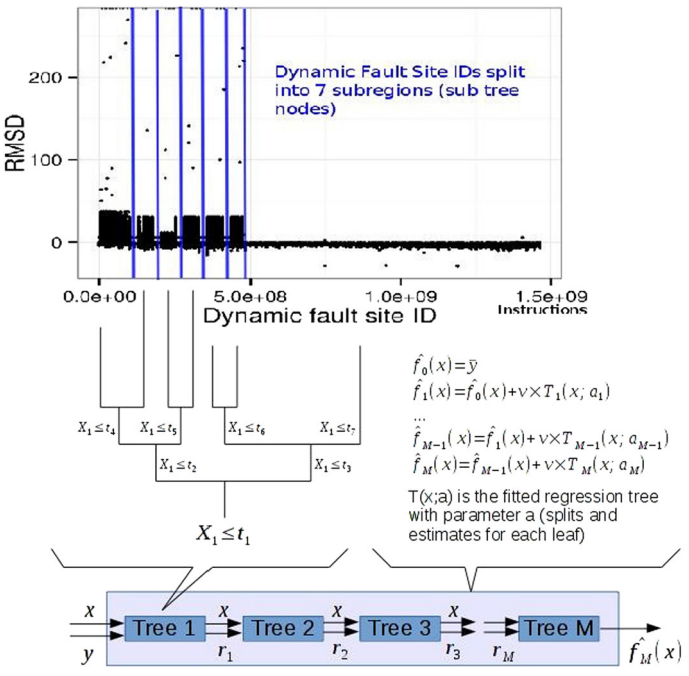



**Fig. 5** Example of one iteration of the BRT training process

independent variable and the error metric in question (RMSD is used as an example) as response may be divided into 7 non-overlapping regions by a decision tree.

The detailed BRT algorithm used in our paper is described in Algorithm 3. In the algorithm, $I(\cdot)$ is an indicator function which returns 1 if the condition is satisfied or otherwise 0. The $v$, named as "shrinkage parameter", controls the learning rate of the BRT. In this study we use the value of 0.1 which results in faster learning speed and better prediction accuracy.

From the user's point of view, the BRT is capable of capturing complex, multi-variate functions without the knowledge of the underlying distributions. Such knowledge is not required by BRT. Also, BRT is unaffected by outliers.

The BRT is able to determine the relative importance of variables. The importance is measured based on the number of times a variable is selected for splitting, weighed by the squared improvement to the model as a result of each split, and then average over all trees. A higher number indicates greater importance.

In this paper, we use the BRT to predict the propagation of errors after a certain number of time steps. The process is described in Sect. 3.3.

## 4 The big data kernels

In this section we describe the big data kernels and do a simple analysis of the error propagation characteristics based on our understanding of the underlying algorithms. The patterns are captured by the statistical model described in Sect. 3.5.

**Algorithm 3**: BRT algorithm used in this paper

Initialize $\hat{f}_0(x_i) = \bar{y}$, where $\bar{y}$ is the average for $\{y_i\}$ ;
**for** $m = 1, 2, ..., M$ **do**
    Compute the current residuals $r_{im} = y_i - \hat{f}_{m-1}(x_i), i = 1, ..., n$;
    Partition the input space into H disjoint regions $\{R_{hm}\}_{h=1}^{H}$ based on $\{r_{im}, x_i\}_{i=1}^{n}$;
    For each region, compute the constant fit $\gamma_{hm} = argmin_{\gamma} \sum (r_{im} - \gamma)^2$;
    Update the fitted model $\hat{f}_m(x) = \hat{f}_{m-1}(x) + v \times \gamma_{hm} I(x \in R_h m)$;
**end**

For the benchmarks, The PageRank and Breadth-First-Search (BFS) algorithms are Big Data algorithms by themselves; the classic K-means algorithm may serve as an unsupervised clustering algorithm on its own and can also serve as a preprocessing step in more complex learning tasks; the Stable Fluid Solver is based on linear solvers that are also used in a variety of programs.

### 4.1 PageRank

#### *4.1.1 Error propagation through the Pagerank loop*

The PageRank algorithm computes the importance for each webpage in a network, which is expressed as a graph. The PageRank loop can be expressed as a linear system

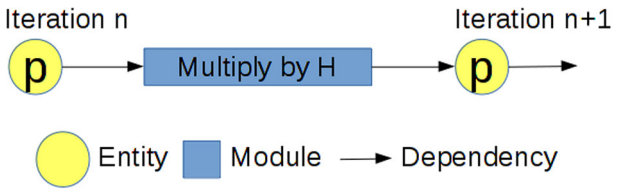



**Fig. 6** Modular structure of Pagerank

$I = GI$. The $G$ matrix is the "Google Matrix" which is derived from the graph topology. The $I$ vector is the importance ranking vector which the algorithm tries to find out.

The algorithm used is a modified Power Method [2], which computes $I \leftarrow GI$ in every iteration.

The algorithm has the following desirable properties:

- As the algorithm makes progress the $I$ will always converge.
- $I$ converges to a value independent of the initial value of $I$.
- Information of the graph will not get lost (i.e. $I$ will not be a zero vector.)

The convergence property can be explained with eigenvalues. Assume a vector $I_0$ can be expressed as the sum of the eigenvectors of $G$, that is,

$$I_0 = c_1 v_1 + c_2 v_2 + \cdots + c_N v_N$$

Applying the definition of eigenvectors ($Gv_n = \lambda_n v_n$), we have:

$$I_k = G^k I_0 = c_1 \lambda_1^k v_1 + c_2 \lambda_2^k v_2 + \cdots + c_N \lambda_N^k v_N$$

Note that the eigenvalues $\lambda_n$ are sorted by their magnitudes in descending order. One characteristic of the Google Matrix is $|\lambda_1| = 1$ and $|\lambda_2| = 0.85$ and the magnitude of all other eigenvalues are smaller than 0.85. This means $I_k$ converges to $c_1 v_1$. After normalization, it becomes $v_1$.

When an error is injected it would only affect the convergence speed of the algorithm rather than the destination of convergence, unless $I$ or the graph data is corrupted. The modular structure of PageRank is shown in Fig. 6.

### 4.2 K-means

The K-means is an unsupervised and iterative clustering algorithm. In this paper we used the K-means implementation from [14]. The algorithm finds the K clusters by minimizing the sum of intra-cluster distance $S = \sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||^2$. The algorithm consists of a loop of `recomputation` and `reassignment` routines which update the cluster centers ($\mu_i$) and cluster memberships ($S_i$), as is shown in Fig. 7.

We quantify the correctness of two clustering results with the quantity Error Factor. Given two clustering results $S_1$ and $S_2$, the Error Factor is defined as $EF = 1 - \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} [[c_1(i) = c_1(j)] = [c_2(i) = c_2(j)]]/k(k+1)$, where $c_1(x)$ and $c_2(x)$ denote

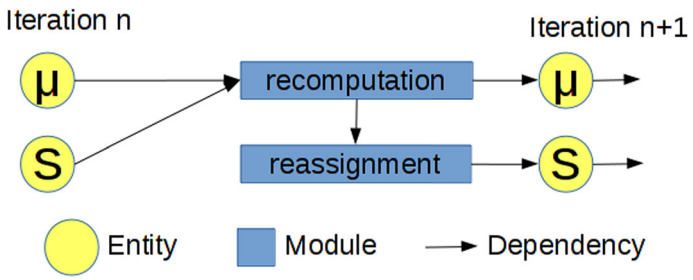



**Fig. 7** Modular structure of K-means

the cluster $x$ belongs to under clustering $S_1$ and $S_2$. The $[\cdot]$ is a boolean function which evaluates to 1 when the condition is satisfied and 0 otherwise. The nominator traverses through all pairs. If the two pairs belong to the same cluster in both $S_1$ and $S_2$, it is incremented by 1. The denominator is the total number of pairs. If $S_1$ and $S_2$ are identical clusters, EF will be zero. Note that the clusters need only contain the same data points but not the same cluster ID. For example, cluster IDs [1,1,2,2] and [2,2,1,1] assigned to four data points are identical because the first two points belong to the same cluster and so do the last two points.

With Error Factor, we can quantitatively compare the results from two runs. We also have the foundations to analyze the correlation between the error in the cluster centers and the Error Factor.

#### *4.2.1 Error propagation through the reassignment step*

In the `reassignment` step, data points are assigned to clusters whose centers are the closest as measured by Euclidean distance. Assume one cluster center is perturbed by a small amount $\mathbf{e}$ towards the direction perpendicular to one of the boundaries of its Voronoi cell, that boundary would move by an amount of $\frac{1}{2}\mathbf{e}$, because the boundary is the perpendicular bisector of the line segment connecting to the centers of the adjacent cells.

Assuming the input dimensionality is $N$, the moving of the boundary sweeps through an volume in the $(N-1)$ dimensional space of $L|\mathbf{e}|$, where L is the area/length of the boundary.

Assume the data points are evenly distributed in the regions with a probability $p$, the swept volume contains $Lp|\mathbf{e}|$ data points. The cluster membership of these points will be altered. This would cause the nominator in the Error Factor to decrease by $Lp|\mathbf{e}|(n-1)$, which is a linear function of $|\mathbf{e}|$. Thus we expect EF to be the linear function of the square root of the L2 norm when the error is small.

### 4.3 Stable fluid simulation

We implemented a 2D fluid simulation program based on the three algorithms (Jacobi, Gauss–Seidel and Conjugate Gradient) described in [10]. The solvers update the elements of a grid repeatedly by solving the Navier–Stokes (NS) equations

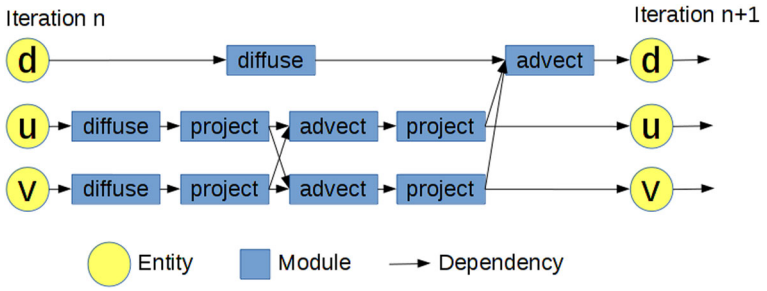



**Fig. 8** Modular structure of stable fluid simulation

$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)u + v\nabla^2 u + f$ and $\frac{\partial d}{\partial t} = -(u \cdot \nabla)d + k\nabla^2\rho + S$, where $\mathbf{u}$ represents the velocity field, and the $d$ represents the density field. Since we simulate fluid in 2 dimensions, $u$ may be written as $(u, v)$ where $u$ and $v$ represent the velocity along the X and Y axis respectively. In this paper, we consider them two entities because each of them goes through the routines listed below.

The Fluid Simulation program operates on the discretized form of the NS equation. It consists the following routines as illustrated in Fig. 8:

- `Diffuse` which solves the first term in the NS equation. It solves a sparse linear system with elements scattering on a band spanning the main diagonal line. All elements except the ones on the band are zeroes. This routine is applied on both the density ($d$) and velocity ($u$ and $v$) fields.
- `Advect` which moves the density through a static velocity field and solves the second term in the NS equation.
- `Project` which subtracts the gradient field from the current velocities. It solves another sparse linear system which is similar to the one in `Diffuse`.

For the `Diffuse` and `Project` routines, one of the Jacobi, Gauss-Seidel and Conjugate Gradient solvers may be used.

The solutions produced by the Jacobi and Gauss-Seidel solvers are nearly identical. The solution produced by the Conjugate Gradient is slightly different, with a L2-norm of around 1e−07. This will affect the characteristics of the initial errors.

We start by discussing the `Advect` routine because it is a good example of how errors can propagate between entities.

#### *4.3.1 Error propagation through the advect routine*

The `Advect` routine propagates the errors from the $u$ and $v$ arrays into the $d$ array and exhibits an easily understandable error propagation pattern. This is because of the this routine computes for each cell the density mass which ends up at each of them at the end of a time step.

For example, the center of the cell (10, 10) is (9.5, 9.5). Assume the velocity field at this cell is (1, 1) and we use a time step of 1. The `Advect` routine traces the center backwards to $(9.5, 9.5) - (1, 1) \cdot 1 = (8.5, 8.5)$, adds up the density at the cells surrounding this point ((8, 8), (8, 9), (9, 8), (9, 9)) weighted by their distance to (8.5, 8.5). If the velocity at this field contains an error $\mathbf{e}$, the back-tracked point would

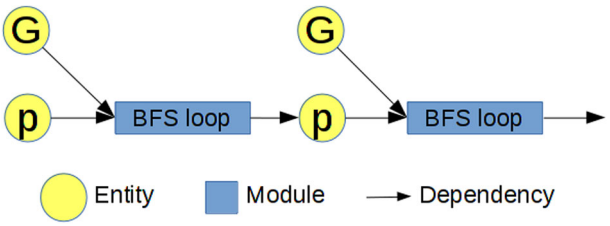



**Fig. 9** Modular structure of BFS

have become $(9.5, 9.5) - ((1, 1) + \mathbf{e}) \cdot 1$. It can be seen that if the magnitude of the error $|\mathbf{e}|$ is small it will only affect the weights of the cells surrounding the source cells. Since the weights are a linear function of $|\mathbf{e}|$, we can expect the error in $\mathbf{d}$ to be a linear function of $|\mathbf{e}|$ as well.

If $|\mathbf{e}|$ is larger it will alter the source cells or even make them go out-of-boundary. In this case the error would not be linear to $|\mathbf{e}|$. Depending on the way boundary conditions are enforced, the erroneous subscripts may be clamped at the boundary of the field.

#### *4.3.2 Error propagation through the linear solvers*

We can view the linear solvers in the `Diffuse` and `Project` routines as solving the equation $Ax_i = x_{i-1}$, where $x$ could be substituted with $d$, $u$ or $v$ and A is the sparse linear system. When an error $e$ is added to the input $x_{i-1}$ we are essentially solving $A(x_i + e_i) = x_{i-1}$. This means the system has become the sum of two systems, whose starting value at time step $i - 1$ are $x_{i-1}$ and $e$.

The characteristics of the linear solver is not relevant to how $e_i$ would change unless it is smaller than the precision bound of the solver. The characteristics of the solver does affect the initial error $e$, if the bit flip occurs during its execution.

### 4.4 Breadth first search (BFS)

The BFS program is a reference implementation of the Graph500 benchmark [15]. It is divided into two phases. In the first phase the program generates a graph, and in the second phase a series of bread-first-search from randomly-chosen starting nodes (the roots) are performed on the graph. The second phase is completed by building the BFS tree, which is represented with a precedence list. In each iteration of the tree-building process, the "frontier" of the current precedence list (p) is being pushed forward using the topological structure of the graph (G). The modular structure is shown in Fig. 9. When the BFS tree is completed, each node will be assigned a level, which is the distance from the root of the tree.

### 4.5 Error metric

We list the error metrics used in the entities of the four Big Data kernels in Table 4. The metrics are computed from the most relevant variables in each of the programs.

**Table 4** Error metrics used for the programs (the root mean square deviation, RMSD is by definition the L2-Norm)

| Program | Error metric (s) |
|---|---|
| Fluid | L2 Norm of the error in the density field ($d$) |
| K-means | L2 Norm of the cluster centers vector; Error Factor of membership |
| PageRank | L2 Norm of the page weights |
| BFS | Proportion of nodes being assigned a wrong level |

**Table 5** Program inputs and number of iterations of the main loop

| Program | Input | Iterations | No. experiments |
|---|---|---|---|
| Fluid | A 50 × 50 grid initialized with a simple pattern | 10 | 24,374 |
| K-means | Dimension-reduced data containing 1797 hand-written characters | 15 | 75,075 |
| PageRank | Amazon web dump containing 65,536 nodes [20] | 14 | 15,057 |
| BFS | Randomly-generated graph with 8192 nodes and 10,650 edges | 7 | 47,984 |

## 5 Experimental results

### 5.1 Input configuration and input generation

The inputs to the Big Data kernels and the number of iterations of the main loop of respective programs utilizing the kernels are listed in Table 5.

### 5.2 Propagation of errors

This section discusses how the error metrics change as the errors propagate during run time.

#### *5.2.1 PageRank*

Figure 10 shows the traces of a subset of injected errors in the PageRank program. Most errors monotonously decrease in a stable way as the iteration count increases. In comparison, although using a similar linear-algebraic algorithm, the Fluid Simulation program tends to see error metrics that preserve their magnitudes without either magnifying or dampening.

#### *5.2.2 K-means*

Figure 11 shows the traces of a subset of injected errors in the K-means program. The age of the error is mapped onto the $X$ axis. The Error Factor is mapped to the $Y$ axis.

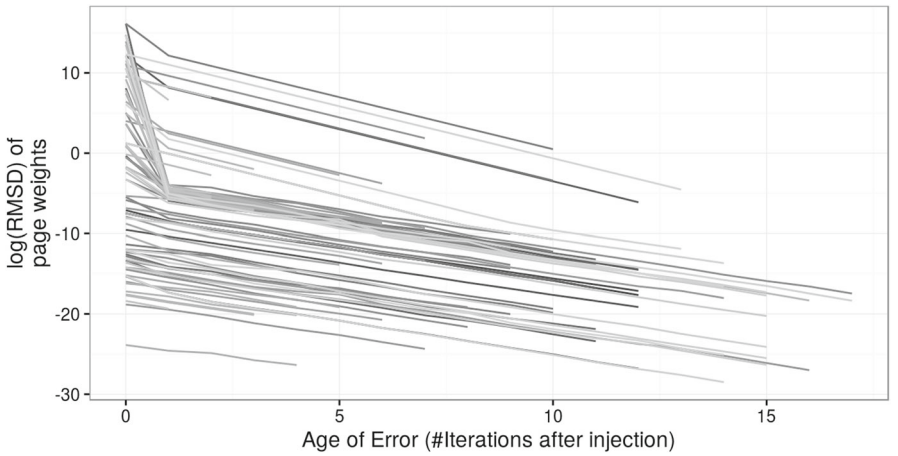



**Fig. 10** Traces of a subset of injected errors in PageRank. *X* axis denotes the "age of a bit flip error" (number of iterations after error injection). *Y* axis denotes the error metric. *Different shades of grey* represent different runs


**Fig. 11** Traces of a subset of errors injected in K-means. *Different shades of grey* represent different runs

It can be seen that the propagation pattern is not uniform; some of the corrupted runs would re-converge to the correct run in a short number of iterations but some could not.

### *5.2.3 Fluid simulation*

Figure 12 shows the trace of a subset of errors induced by bit flips in the Fluid Simulation program. The age of the error, which is the number of time steps passed since error injection, is mapped onto the *X* axis. The errors are injected at random positions, which could be in any iteration. The RMSD in the *d* field is mapped to the *Y* axis.

From the figure it can be seen that the error magnitudes tend to change gradually as time step advances. The magnitude also tends to stabilize. The trend at which the

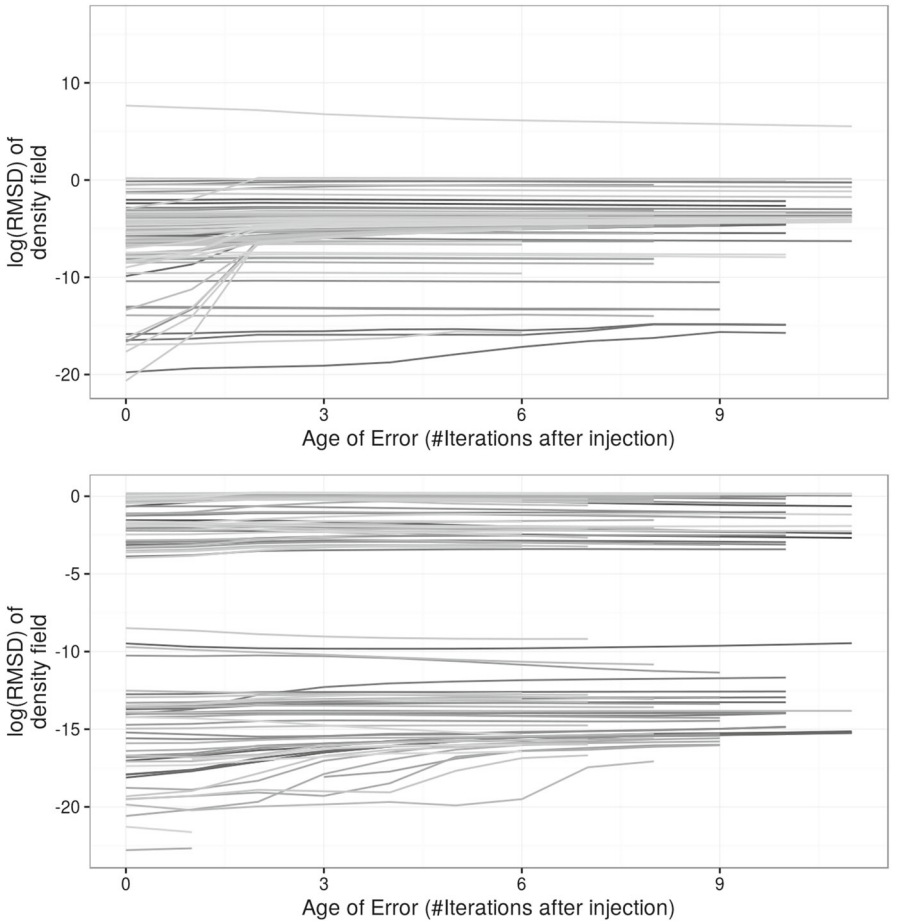



**Fig. 12** Traces of a subset of errors injected in fluid simulation with the conjugate gradient (CG) solver (*top*) and the Gauss-Seidel (*GS*) solver (*bottom*). *Different shades of grey* represent different runs

magnitudes change is dependent on the initial magnitudes. To illustrate, the initial magnitude of CG are mainly distributed between $[10^{-7.5}, 10^{0}]$, which is different from that of GS, $[10^{-20}, 10^{0}]$. The final magnitudes are also different.

### *5.2.4 Breadth first search (BFS)*

Figure 13 shows the trace of 100 errors induced by bit flips in the BFS program. The age of the error (number of iterations after injection) is mapped to the $X$ axis. The proportion of nodes that would receive a wrong level based on the intermediate BFS search tree at individual time steps are mapped to the $Y$ axis.

As we can see from the figure, most bit-flip induced errors in BFS monotonously increase. In some cases, the result would become completely incorrect due to critical data structure being corrupt.


**Fig. 13** Traces of a subset of errors injected in Breadth First Search. *Different shades of grey* represent different runs

### 5.3 Model training and accuracy

This section discusses the accuracy of Model 1 and Model 2 described in Sect. 3.3. For Model 1 we quantify how much it is able to model the relationship between the dynamic fault site information to the distribution of errors, namely how a bit-flip propagates to program states. For Model 2 we quantify how much it is able to model the propagation of errors between time steps.

Accuracy for both models is quantified by comparing against ground truth. We compute the earth mover distance (EMD) between the predictions and the actual RMSD at the end iteration, denoted $EMD_1$. We also compute the earth mover distance (EMD) between the RMSD distribution at the beginning iteration and the end iterations, denoted $EMD_2$. We compute the ratio $\frac{EMD_1}{EMD_2}$. $EMD_2$ Thus, a *smaller* the ratio means a more accurate prediction.

The beginning and ending iteration numbers are (4, 14) for PageRank, (1, 15) for K-means, and (4, 10) for Fluid Simulation.

For each application, we vary the proportion of the examples used for training and see how the prediction quality varies. We pick the traces by their unique combination of fault injection parameters (DynamicFSID, BitID) into the training and test set. The Static Fault Site ID (StaticFSID) is implied by DynamicFSID so it doesn't need be included.

We measure the prediction error on the entire data set. That means the training set and the prediction output from the test set together make up the error distribution at the ending iteration.

#### *5.3.1 Pagerank*

Due to the simplicity in the error propagation patterns, a segmented linear regression model is enough for capturing the error propagation pattern of PageRank, as shown in Fig. 14.

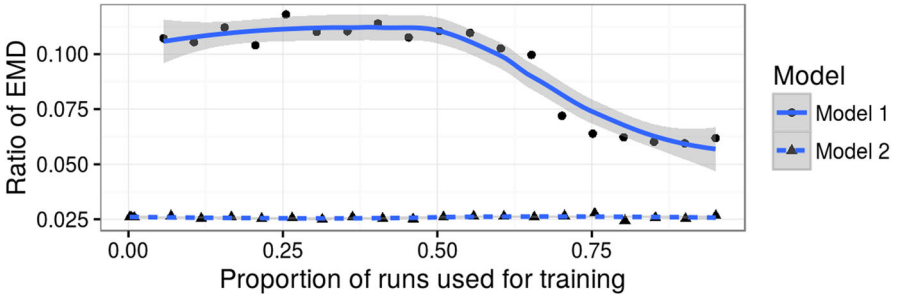



**Fig. 14** For Pagerank, Model 1 needs 75 % of the input data for training to reach the maximum predictive power. Model 2 needs only a few data points to reach the maximum predictive power

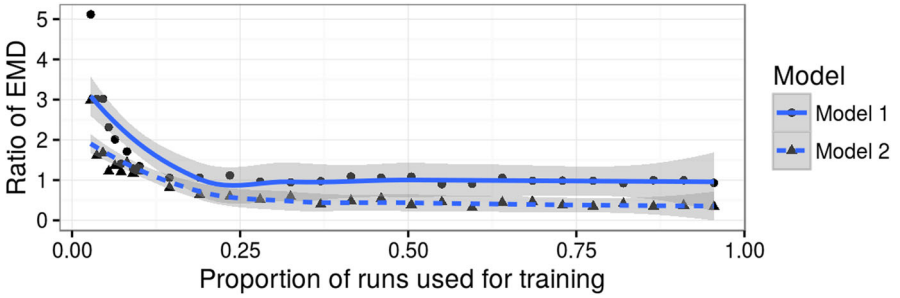



**Fig. 15** For K-means, both Model 1 and Model 2 need 25 % of the input data for training to reach maximum predictive power

### *5.3.2 K-means*

We had to use a regression tree to capture the error propagation pattern of K-means, because there is one segment in the range of the input RMSD that does not have a one-to-one mapping. The correctness improves as training set size increases, as shown in Fig. 15.

### *5.3.3 Fluid simulation*

A segmented linear regression model is used for Fluid as is shown in Fig. 16 because the RMSD propagation pattern is simple. Most of the changes in the RMSD are in predictable directions. From the results we see the performance of Model 2 is very stable. Even with very few training examples, Model 2 is able to capture the change of the RMSD changes.

### *5.3.4 Breadth first search (BFS)*

The regression tree is used to capture the error propagation pattern of BFS because of the non-linear pattern, as is shown in Fig. 17. It is worth noting that Model 2 suffers from over-fitting when the proportion of data used for training is high.

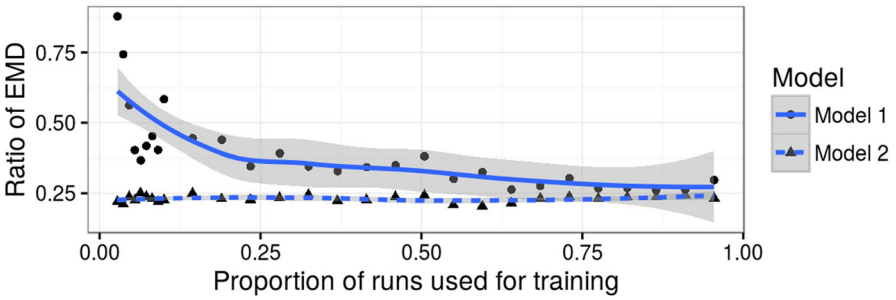



**Fig. 16** For Fluid Simulation, Model 1 needs 25 % of the input data for training to reach the maximum predictive power. Model 2 needs only a few data points to reach the maximum predictive power

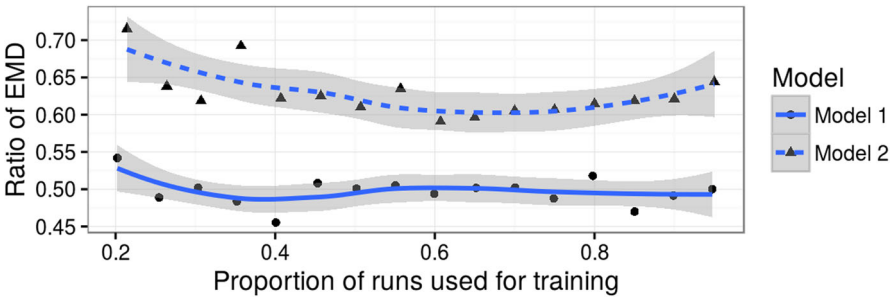



**Fig. 17** For BFS, Model 1 needs about 60 % of the input data for training to reach the maximum predictive power. Model 2 needs about 50 % of the input for training to reach the maximum predictive power

**Table 6** Variable relevance in Model 1

| Program | DynamicFSID | StaticFSID | BitID | NumIter |
|---|---|---|---|---|
| Fluid | 0.523348 | 0.316858 | 0.118150 | 0.041644 |
| K-means | 0.016321 | 0.691838 | 0.290197 | 0.001644 |
| Pagerank | 0.445098 | 0.327972 | 0.129911 | 0.097019 |
| BFS | 0.940505 | 0 | 5.949493 | 0 |

## 5.4 Factors affecting model accuracy

### *5.4.1 Model 1*

The independent variables of Model 1 are listed in Table 3. However, not all of the variables are equally relevant to the final RMSD.

As Table 6 shows, the most relevant variable for Fluid and Pagerank is DynamicFSID, and for K-means, the most relevant variables are StaticFSID and BitID.

DynamicFSID being irrelevant in K-means suggests the shape of the error trajectories is not affected by which iterations are being injected errors. In other words, it is uncertain whether the error would be dampened or amplified across iterations.

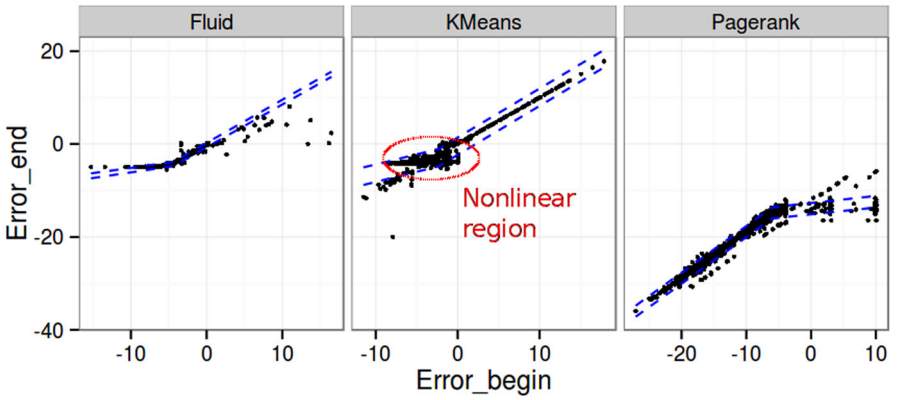



**Fig. 18** Errors in program variables at the beginning and ending iterations (*X* and *Y* axis). *Dashed lines* are prediction intervals of segmented linear models

In contrast, the patterns in Fluid and Pagerank are more stable, as can be seen from Figs. 10, 11 and 12.

For BFS, DynamicFSID is most relevant, followed by BitID. StaticFSID and NumIter are completely irrelevant. The reason is because error injected into all but only a few of the static faults are masked and will not result in any observable error in program states.

### *5.4.2 Model 2*

Model 2 takes the error distribution at the beginning and end iterations. As a result, the relationship between the errors, visualized in Fig. 18, determines the model's prediction quality.

Visually, there is linear correlation between the errors: greater errors at the beginning iteration means greater errors at the ending iteration. The only exception is when the error at the beginning iteration is small enough, the output error would be constant in Fluid. Same for Pagerank if the error at the beginning iteration is large enough. For these cases segmented linear regression would be enough for capturing the shapes. To fix the effects caused by outliers, we have removed the top and bottom 5 % of the input data.

However, there exists a non-linear region in K-means which affects the predictive power of the segmented linear regression. The region is highlighted in Fig. 18. One *X* co-ordinate in this region may correspond to two *Y* axis, which forces the predictive interval to become larger and results in greater error in the predicted errors. To fix this we decided to use the regression tree, which is more complex than line segments and can better capture the shape by further subdividing the input data set.

Since there exists multiple program variables, we should find the ones that most accurately capture the error propagation patterns with the best accuracy. Actually, certain variable combinations may make prediction more difficult. Figure 19 shows the choices that are not desirable for building Model 2.

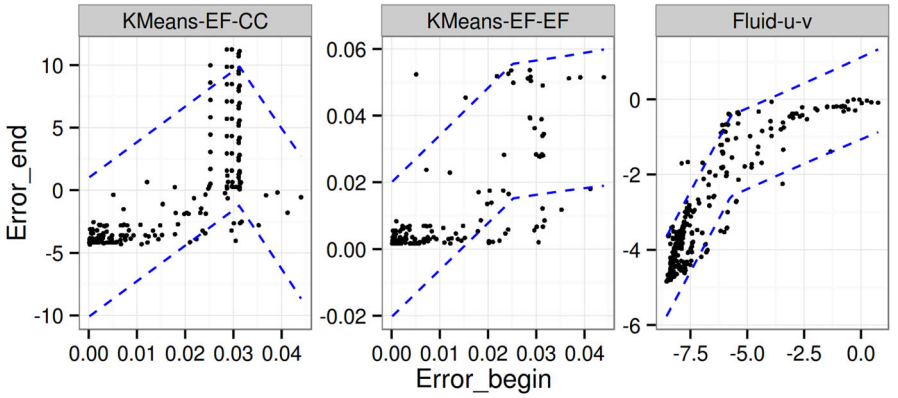



**Fig. 19** Undesirable choices of variables for Model 2

**Table 7** Cost to characterize the effect of soft faults on a program

| Program | Iter | Fraction (%) | Algorithm 1 Cost | Algorithm 2 Cost | Saving (%) |
|---|---|---|---|---|---|
| Fluid | 10 | 1 | 100 | 55.45 | 45.6 |
| K-means | 15 | 25 | 225 | 146.25 | 35.0 |
| Pagerank | 14 | 1 | 196 | 105.91 | 45.9 |
| BFS | 7 | 50 | 49 | 42.00 | 14.3 |

### *5.4.3 Cost saved by the inter-iteration efficient fault characterization algorithm*

Consider performing $NF$ fault injection experiments into a program that runs for $N$ iterations. Algorithm 1 runs all instances to completion, and the cost measured in number of program iterations is $NF \cdot N$. Algorithm 2 picks a fraction from each iteration and run to completion, and the cost measured in number of program iterations is $\sum_{i=1}^{i=N} i + (N - i) \cdot \mu$, where $\mu$ is the ratio between instances in an iteration run to completion and the total number of instances with faults injected at that iteration.

With results from Sect. 5.3 we set $\mu$ to 0.01 for Fluid and Pagerank and 0.25 for K-means. By plugging in the numbers we could obtain the costs in Table 7.

## 5.5 Applying fault resilience techniques

With the results obtained in Sect. 5, ErrorSight produces the error profile of a program and shows the expected error metric caused by a bit flip on the instructions that correspond to eachsource line. With this information, the developer can use to decide how to apply fault resilience techniques. In this table, column mean error (ME) shows the expected error that would appear in the final output if a bit flip is injected into a dynamic instruction that corresponds to this line of source code. The column Probability (P) shows the probability that a dynamic fault site belongs to this line. The column Product (Prod) is a product of ME and P. Intuitively, the sum of all the entries

**Table 8** Static fault site to source code mapping of Breadth First Search

| Line no. | Source code | Mean error (ME) | Probability (P) | Product (prod) | Mean error FT | Probability FT | Product FT |
|---|---|---|---|---|---|---|---|
| 120 | `for(i=0; i<nlocalverts; ++i) pred[i]=-1` | 2.41e−1 | 1.23e−1 | 2.98e−2 | 1.22e−4 | 5.26e−2 | 6.42e−6 |
| 192 | `for(i=0; i<oldg_count; i++) {` | 1.06e−1 | 7.10e−2 | 7.51e−3 | 1.06e−1 | 1.0e−3 | 1.06e−4 |
| 200 | `for(j=g->rowstarts[ VERTEX_LOCAL(oldg[i])]; j<j_end; j++) {` | 6.59e−3 | 3.32e−1 | 2.19e−3 | 3.63e−3 | 4.80e−1 | 1.74e−3 |
| 208 | `if(!TEST_VISITED(tgt)) {` | 4.08e−3 | 3.67e−1 | 1.5e−3 | 3.32e−3 | 3.09e−1 | 1.03e−3 |
| 209 | `SET_VISITED(tgt);` | 1.43e−3 | 6.53e−2 | 9.35e−5 | 1.06v3 | 4.60v2 | 4.90v5 |
| | (Other) | N/A | N/A | 2.45e−4 | N/A | N/A | 5.18e−4 |
| | (Sum) | | | 4.13e−2 | | | 3.44e−3 |

The expected error in the final output resulted from a random bit-flip error is 4.13e−2 and 3.44e−3 in the original and fault-tolerant versions, respectively

```
Original Line 120

for (i = 0; i < nlocalverts; ++i)
   pred[i] = -1;
```

```
Fault-Resilient Line 120

int64_t* p1 = pred, *p2 = p1, *p3 = p2;
int64_t* end1 = &(pred[nlocalverts]),
         *end2=end1, *end3=end2;
while (p1 < end1 || p2 < end2 || p3 < end3) {
    if ( !(p1 == p2 && p1 == p3) ) {
        if (p1 == p2) { p3 = p2; }
        else if(p2 == p3) { p1 = p2; }
        else if(p1 == p3) { p2 = p3; }
        else { goto retry; }
    } else { *p1 = -1; }
    p1++; p2++; p3++;
    if ( !(end1 == end2 && end2 == end3) ) {
        if (end1 == end2) { end3 = end1; }
        else if (end2 == end3) { end1 = end2; }
        else if (end3 == end1) { end2 = end3; }
        else { goto retry; }
    }
}
```

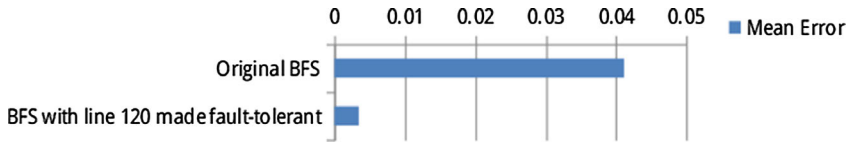



**Fig. 20** Triplication fault resilience mechanism used on Line 120 and the resultant change in the mean error of the entire BFS program

in the Prod column is the weighted sum of the ME column, which is the expected error in the final output of the program should a bit flip occurs randomly during its run time. Columns without and with "FT" represent the metrics from the original and the fault-tolerant versions of the program.

We choose to make the source code lines that are most vulnerable to the Breadth First Search (BFS) shown in Table 8. In this Table the greatest value in the Prod column belongs to Line 120 of the source code of BFS. This means that this line is the most significant contributing factor to the overall resilience of the program.

We manually triplicated the pointer dereferencing and value assignment operations in the loop, and performed a Byzantine error check [16] before incrementing the loop index and writing to the `pred` array, namely, if one replica of a pointer is corrupt, the other two are used to correct it, and if two or more replicas are corrupt, the loop is restarted from the beginning. This effectively reduced the occurrence of out-of-loop-boundary errors and the assignment of wrong values.

Figure 20 shows the fault-resilience source code and the resultant change in the mean error of the entire program after fault resilience is applied to Line 120. The mean error of the application is reduced by a magnitude, from 4.13e−2 to 3.44e−3. Table 8 indicates the Mean Error resulting from the fault-resilient version of Line 120 has been reduced from 2.98e−2 to 6.42e−6. After this, Line 120 is no longer the main contributor of errors in this BFS. In addition, the modification does not introduce significant overhead because Line 120 was not a hotspot in the original program.

## 6 Summary

In this paper we have proposed ErrorSight, a tool aimed at helping the developers to write fault-resilient programs. We demonstrated with four Big Data kernels that it can efficiently capture the error propagation patterns that a human developer can analytically obtain, and use the patterns to construct a predictive model to save the error characterization cost, and showing the application developer which part of the source code is the most significant vulnerable part of a numerical program. With this information, the developer can then apply fault resilience mechanisms to the program and significantly improve its resilience under a faulty environment.

**Acknowledgments** We are grateful to Vishal Sharma and Arvind Haran, the authors of the original KULFI and for granting us permission to modify it for our experiment purposes. We are also appreciative of the opportunity to be involved in and contribute to KULFI.

## References

1. Kulfi fault injector (2014). https://github.com/quadpixels/kulfi. Accessed 4 Mar 2015
2. Austin D (2010) How google finds your needle in the web's haystack. http://www.ams.org/samplings/feature-column/fcarc-pagerank, http://www.ams.org/samplings/feature-column/fcarc-pagerank
3. Austin T (1999) Diva: a reliable substrate for deep submicron microarchitecture design. In: Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO 1999)
4. Baumann RC (2005) Radiation-induced soft errors in advanced semiconductor technologies. In: IEEE Transactions on Device and Materials Reliability, vol 5
5. Breiman L, Friedman J, Stone CJ, Olshen RA (1984) Classification and regression trees. CRC Press, USA
6. Cappello F, Geist A, Gropp B, Kale S, Kramer B, Snir M (2009) Toward exascale resilience. In: International Journal of High Performance Computing Applications
7. Chung J, Lee I, Sullivan M, Ryoo JH, Kim DW, Yoon DH, Kaplan L, Erez M (2012) Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC12)
8. Du P, Luszczek P, Dongarra J (2012) High performance dense linear system solver with resilience to multiple soft errors. Procedia Comput Sci 9:216–225
9. Elliott J, Hoemmen M, Mueller F (2014) Evaluating the impact of SDC on the GMRES Iterative Solver. In: Proceedings of the 28th International Parallel and Distributed Processing Symposium (IPDPS 2014)
10. Goncalo Amador AG (2009) Linear solvers for stable fluids: GPU vs CPU. In: 17th Encontro Portugues de Computacao Grafica (EPCG'09)
11. Huang KH, Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. In: IEEE Transactions on Computers, vol C-33
12. Kumar S, Hari S, Adve SV, Naeimi H, Ramachandran P (2012) Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In: Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)
13. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis and transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004). San Jose
14. Liao WK (2013) Parallel K-means data clustering. http://users.eecs.northwestern.edu/wkliao/Kmeans/
15. Murphy RC, Wheeler KB, Barrett BW, Ang JA (2010) Introducing the Graph 500. Cray Users Group (CUG)
16. Nanya T, Goosen H (1989) The Byzantine hardware fault model. IEEE Trans Comput Aided Design Integr Circuits Syst 8:1226–1231

17. Rubner Y, Tomasi C, Guibas L (1998) A metric for distributions with applications to image databases. In: Proceedings of the Sixth International Conference on Computer Vision (ICCV 1998), pp 59–66
18. Schroeder B, Pinheiro E, Weber WD (2009) DRAM errors in the wild: a large-scale field study. In: Proceedings of SIGMETRICS
19. Stott DT, Floering B, Burke D, Kalbarczyk Z, Iyer RK (2000) NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. In: Proceedings of the 2000 IEEE International Computer Performance and Dependability Symposium (IPDS 2000)
20. Wang L, Zhan J, Luo C, Zhu Y, Yang Q, He Y, Gao W, Jia Z, Shi Y, Zhang S, Zheng C, Lu G, Zhan K, Li X, Qiu B (2014) BigDataBench: a big data benchmark suite from internet services. In: Proceedings of the 20th International Symposium on High-Performance Computer Architecture (HPCA 2014)