Evaluating Application Resilience with XRay

Sui Chen*, Greg Bronevetsky†, Bin Li*, Marc Casas Guix§ and Lu Peng*
*Louisiana State University
†Lawrence Livermore National Lab
‡Barcelona Supercomputing Center

***Abstract*—The rising count and shrinking feature size of transistors within modern computers is making them increasingly vulnerable to various types of soft faults. This problem is especially acute in high-performance computing (HPC) systems used for scientific computing, because these systems include many thousands of compute cores and nodes, all of which may be utilized in a single large-scale run. The increasing vulnerability of HPC applications to errors induced by soft faults is motivating extensive work on techniques to make these applications more resiilent to such faults, ranging from generic techniques such as replication or checkpoint/restart to algorithm-specific error detection and tolerance techniques. Effective use of such techniques requires a detailed understanding of how a given application is affected by soft faults to ensure that (i) efforts to improve application resilience are spent in the code regions most vulnerable to faults and (ii) the appropriate resilience technique is applied to each code region. This paper presents XRay, a tool to view the application vulnerability to soft errors, and illustrates how XRay can be used in the context of a representative application. In addition to providing actionable insights into application behavior XRay automatically selects the number of fault injection experiments required to provide an informative view of application behavior, ensuring that the information is statistically well-grounded without performing unnecessary experiments.**

I. Introduction

Soft faults in processor circuits can affect applications in three major ways:

- Abnormal Termination: Program performs an erroneous action that is detected by the hardware or operating system and is aborted.
- Erroneous Result: Program runs to completion but produces erroneous results.
- Correct Result: Program runs to completion and outputs the correct result, as if no error occurred.

To ensure that computers can be used reliably in spite of soft faults it is necessary to make applications resilient to their effects. This includes the design of new resilience techniques and their effective deployment in individual applications. Since in most cases different application code regions have different vulnerability properties and require different resilience techniques the primary things that developers need to understand are (i) how the source code location where a given soft falult manifests as an error relates to the application outcome (termination, erroneous result or correct result), and (ii) whether different executions of the same code region are affected differently by errors.

The XRay tool is designed to provide this information in an accessible way. As illustrated in Figure 1, XRay performs a fault injection campaign on the target application, executing either the entire target application or an individual routine many times on the same input and injecting the output of a randomly-selected instruction within each run with a single bit flip. XRay records the dynamic index of the injected instruction (the number of instructions completed before the injected one), the instruction's location in the source code and the index of the bit in the instruction's output that is flipped. Further, it records the result of the application, whether it aborted, completed correctly or produced an erroneous output. The magnitude of any result error is computed using an application-specific error metric and also recorded. This information is then visualized to relate the fault injection information to application outcomes. XRay's fault injection campaign is controlled by a tree-based statistical regression model [1] that attempts to predict the outcome of an injected error based on the above information and keeps conducting more fault injection experiments until the model determines that no more predictive accuracy can be achieved based on these training features. Since this is the point where XRay's visualizations cannot grow more accurate with more experiments, XRay's fault injection campaign is limited to this number of experiments.

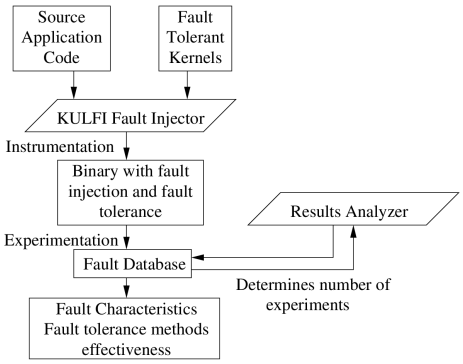

Fig. 1. Overview of the XRay framework.

Given an application that needs to be made more resilient XRay's views are used to identify the code regions that are the most vulnerable to soft faults based on the effects of faults injected during the execution of these code regions on application results. Once application developers implement resilience techniques for critical application regions XRay can be applied to the modified application to understand how the improvements affect overall application resilience and whether a given resilience technique is cost-effective.

This paper is organized as follows. Section II-A describes our fault injection infrastructure and Section II-B discusses the application used in our case study. Section III presents the functionality of XRay in the context of our representative application, showing how XRay is used to identify the application's critical routines, its visualizations of the error vulnerability of the original and resilient versions of these routines and finally, how the use of resilience affects the error vulnerability of the overall application. Section V then details our algorithm for selecting the number of fault injection experiments to use in our analysis.

II. Experiment setup

A. Fault injection experiments

XRay is based on the KULFI fault injector [4], which runs on applications compiled into the LLVM [3] bitcode. KULFI transforms the bitcode representations by wrapping instructions with code to inserting bit-flips into their outputs at runtime. KULFI can also be linked with a library to call functions within the library on every error injection to allow external tools such as XRay to record information about the injection. In the program image to be instrumented, each wrapped LLVM instruction that takes up a certain amount of space in the program image is referred to as "a static fault site"; When the program is being executed, each instance of the wrapped instructions to be executed is referred to as "a dynamic fault site". In other words, there is an 1:1 mapping between program offset and a static fault site, and a 1:1 mapping between time and dynamic fault site.

XRay analyzes an application's vulnerability to soft errors by running the target application many times and injecting an error into each run. Since our focus is on systems where multiple errors in a given application's execution are rare, exactly one error is injected into each run, with the dynamic fault site chosen uniformly at random. We use two methods to select the the bits of an instruction's output that is injected. In half of the experiments we inject into all bits of a fault site. In the other half, we inject into randomly chosen bits, one at a fault site. The purpose of each method is to capture the effect of Bit ID and time on the outcome of a fault, respectively, and provide more meaningful input for the model described in Section V. After fault injection, the application runs until it exits, at which point XRay collects the results from this pass. This procedure is repeated until the model described in Section V determines that the set of experiments is representative.

B. Target Application

We demonstrate the use of XRay on an application that uses the Alternating-Direction Method of Multipliers [5] to solve a Lasso problem, which fits a linear model $\hat{y} = b_0 + b_1 x_1 + b_2 x_2 + ... + b_p x_p$. In our experiment, the dimentionality of y and b are 40x1 and 40x500, respectively. Lasso utilizes numerical routines from the GNU Scientific Library. The amount of error in the vector output by Lasso is quantified via Root Mean Standard Deviation (RMSD) of the difference between the known correct vector and the vector produced by a given run. It is defined as $RMSD = \sqrt{\sum_{i=1}^{N} (x_i - \hat{x}_i)^2 / N}$. Smaller RMSD values correspond to smaller errors.

III. Experimental results

This section describes how XRay can be applied to an application, using Lasso as a representative example. The figures in this section show the data that would be presented to users of XRay.

A. Vulnerability of Lasso

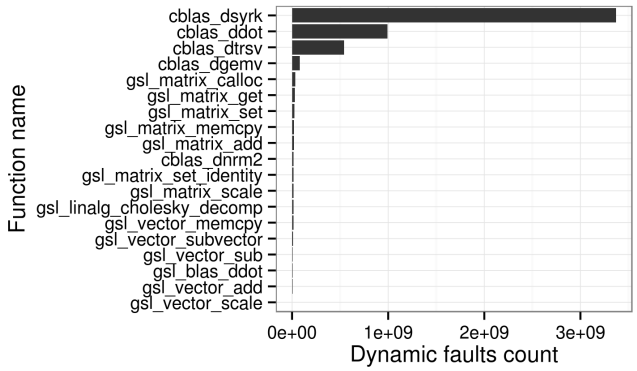

Fig. 2. Functions of lasso most dynamic fault sites belong to

Figure 2 identifies the routines that Lasso utilizes most by showing the number of fault injections that occur in different routines. The `cblas_dsyrk` routine computes the Rank-K matrix update and `cblas_dgemv` computes Matrix-Vector Multiplication (MVM). `cblas_ddot`, `cblas_dtrsv` and `cblas_dnrm2` are subroutines utilized by the Cholesky Decomposition routine.

The high-level resilience properties of Lasso are shown in Figure 3, which presents for each dynamic fault site the probability that an error injected at that site will result in Abnormal termination, Erroneous Result or Correct Result. As marked in the figure, Lasso's execution consists of three distinct program phases, each dominated by one linear algebra kernel: Rank-K update, Cholesky Decomposition and Matrix-Vector Multiplication.

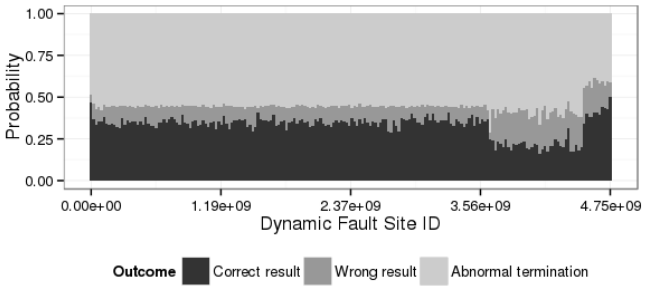

Fig. 3. Probability of outcome of errors during the lifetime of Lasso and the routine that dominates running time in each phase.

The above information identifies the dominant routines in each of Lasso's distinct program phases. A bit-flip error in each phase would cause a large fraction of the runs to terminate abnormally or complete with erroneous results. As such, making each of Lasso's dominant routines more resilient to soft errors can be expected to significantly improve the resilience of the overall application, while work on other routines is unlikely to be productive. Section III-B considers each of these three routines, detailing their fault characteristics and the impact of deploying resilience techniques on them.

B. Results on individual routines

1) Cholesky Decomposition: Figure 4 details the vulnerability properties of the Cholesky Decomposition (CD) routines, for both the original and resilient versions. The table at the top shows the probability that an injected soft error will result in an Abnormal Termination, Erroneous Result or Correct Result. The graphs below show more detail about runs with erroneous results, with the top two graphs focusing on the original version and the bottom graphs on the resilient version. The scatterplots on the left show the magnitude of errors in outputs of CD (quantified as RMSD) on the vertical axis, with the horizontal axis denoting the Dynamic Fault Site ID (i.e. injection time). The histogram on the right projects the scatter plot onto the vertical axis, showing the number of error injections (horizontal axis) that resulted in an RMSD of a given magnitude (vertical axis). The vulnerability information all routines is shown using the same format.

The original CD routine has a built-in assertion that terminates the program when the matrix is not positive-definitive. Since most injected errors cause the assertion to be violated, most runs of the original CD are terminated, while those that complete generally finish with very small errors.

Resilience is applied to CD by observing that CD decomposes matrix A into $L \cdot L^T$ where L is lower-triangular with a positive diagonal. This operation must maintain the identity $Ax = L \cdot (L^T x)$ [2], which can be computed in $O(n^2)$ time. This is significantly cheaper than the $O(n^3)$ complexity of the deterministic CD algorithm. GSL implements an iterative algorithm that runs faster than $O(n^3)$ time but our experiments show that our checker is still significantly faster.

In addition to the algorithm-specific check we also employed light-weight checkpointing based on the signal handling mechanism in by Linux. We call `sigsetjmp` before the program enters the linear algebra kernels and back up their inputs. When the program encounters an error and is about to abort, the signal handler is triggered, reverting the program state to the latest checkpoint and restores the inputs if possible. All the numerical routines employ this method.

The use of these resilience technique has a significant effect on the vulnerability of CD to injected errors. Many runs that would otherwise trigger the assertions finish with very small errors, which means the damage to the inputs by a single bit flip error is very small, and can be easily recovered through input backup. Overall, Cholesky Decomposition benefited most from the generic segmentation fault error handler.

Cholesky Decomposition

	Abnormal Termination	Erroneous Result	Correct Result
Original	26,620 (90%)	704 (2%)	2,238 (8%)
Resilient	740 (2%)	178 (1%)	27,294 (97%)

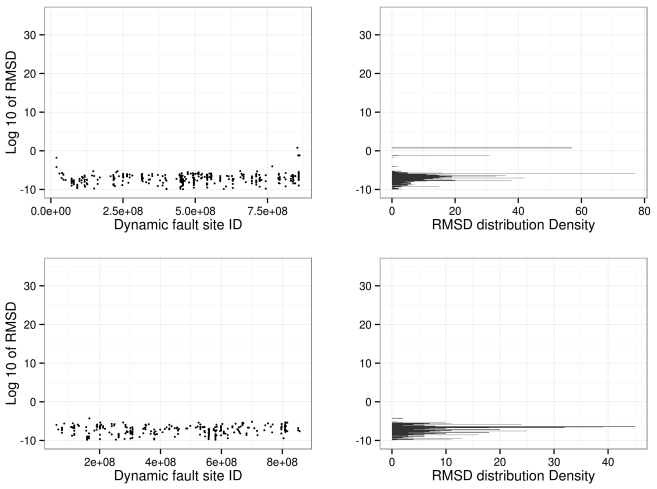

Fig. 4. Error Vulnerability of Cholesky Decomposition, original (top row) and resilient (bottom row) versions

2) Rank-K Update: Figure 5 shows the soft error vulnerability of both the original and the resilient version of Rank-K (RK) update. This algorithm computes $\alpha A \cdot A^T + \beta B$, where A and B are matrixes. Its results are checked via the identity $(A \cdot B) \cdot x = A \cdot (B \cdot x)$, where x is an error-checking vector (we use the vector of all 1s). Since checking is done using matrix-vector multiplication, which takes $O(n^2)$ time, as compared to $O(n^3)$ time for RK, this check is very efficient. Like CD, the error checker for Rank-K update causes many runs with Erroneous Results to produce Correct Results. However, 15% of the erroneous runs are not corrected, mainly due to round-off errors in the checker, since the error magnitudes are small.

3) Matrix-vector multiplication: The Matrix-vector multiplication(MVM) operation computes Ax, where A is a matrix and x is a vector. It is checked using a similar identity $(x^T A)x = x^T(Ax)$. The complexity of computing $x^T A$ is $O(n^2)$, the same as the original MVM but using additions rather than multiplications. Since in Lasso MVM is applied many times to the same matrix with different vector, the vector $x^T A$ can be reused, amortizing the cost of computing it. The effect of adding resilience of MVM is shown in Figure 6 and is very similar to RK.

C. Effectiveness of resilience methods on Lasso

Figure 8 shows the effect of adding resilience to the primary routines of Lasso on the vulnerability of Lasso to soft errors. Most of the errors in Lasso have been handled by the fault checkers, with over 94% of abnormal terminations recovered from and over 90% of the runs completing correctly. This improvent is the result of the effective protection the provided by the algorithmic checkers for the three primary

Rank-K update

	Abnormal Termination	Erroneous Result	Correct Result
Original	14,434 (50%)	12,865 (45%)	1,416(5%)
Resilient	12,927 (45%)	4,527 (15%)	11,591(40%)

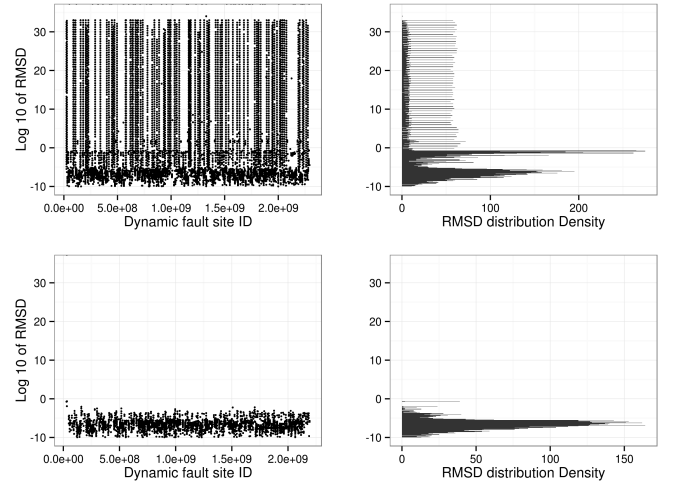

Fig. 5. Error Vulnerability of Rank-K update, original (top row) and resilient (bottom row) versions

Matrix-vector multiplication

	Abnormal Termination	Erroneous Result	Correct Result
Original	25,309(51%)	16,530(33%)	7,992(16%)
Resilient	7,875(11%)	693(1%)	60,960(88%)

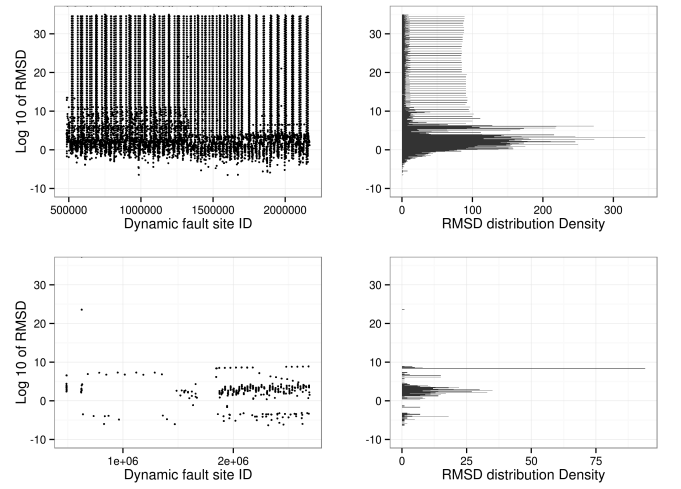

Fig. 6. Error Vulnerability of Matrix-Vector Multiplication, original (top row) and resilient (bottom row) versions

routines in Lasso (RK, MV and Cholsky Decomposition) and the signal handler's successful recovery from most abnormal terminations. This demonstrates the utility of XRay in guiding application developers towards productive use of resilience techniques in applicationa and quantifying for them both the vulnerability of a given application and the degree to which it improves when various resilience techniques are applied to it.

Lasso

	Abnormal Termination	Erroneous Result	Correct Result
Original	52746(55.54%)	11012(11.59%)	31214(32.87%)
Resilient	1424(5.91%)	839(3.48%)	21832(90.60%)

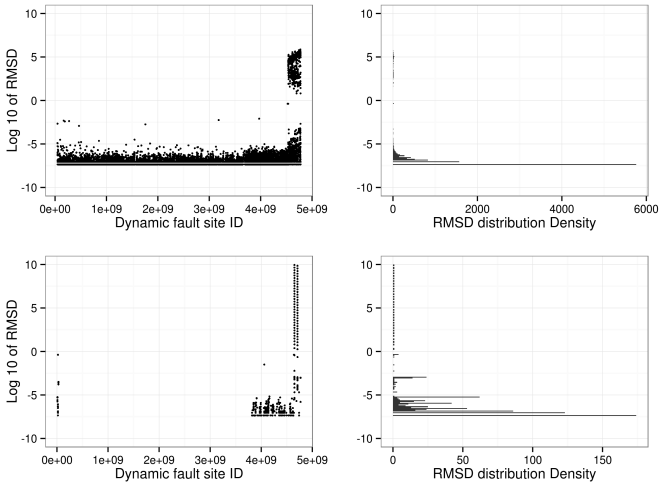

Fig. 7. Fault characteristics of Lasso, without (in row 1) and with fault tolerance (in row 2)

IV. An Intuitive Accuracy Metric

Even in the absence of hardware faults, scientific applications face many sources error, including those in input data, discretization error and modeling error. Here in this case study we relate output errors due to soft faults to measurement errors in application inputs. The purpose of doing this is 1) it shows the error sensitivity of different routines to their inputs and 2) it helps users to understand the error magnitude better by evaluating input errors that are related to a certain RMSD.

Since the input configuration space is astronomically huge, we perform random local search (by multiplying the values by random values sampled from a Gaussian distribution with mean 1 and a standard deviation varying from 1e9 to 1) upon several input configurations. By performing enough experiments (50 in this case for mean RMSD) we would be able to 1) tell how much a program/routine is likely to magnify errors and 2) how the characteristics of a program and the routines it consists are related.

As is shown in the figures, there exists a clear linear relationship between the magnitude of output error, measured in RMSD, and input error, which depends on the SD of the noise added to it for MM, RK and MV routines. Input errors greater than 1e-09 always triggered the assertion failures inside the CD routine and their results are not shown. A similar linear relationship is also observable from results from LASSO, as LASSO spends a considerable amount of CPU time on RK routine.

V. Determining Number of Experiments

The effect of errors on applications is an inherently complex process and it is difficult to determine the number of fault

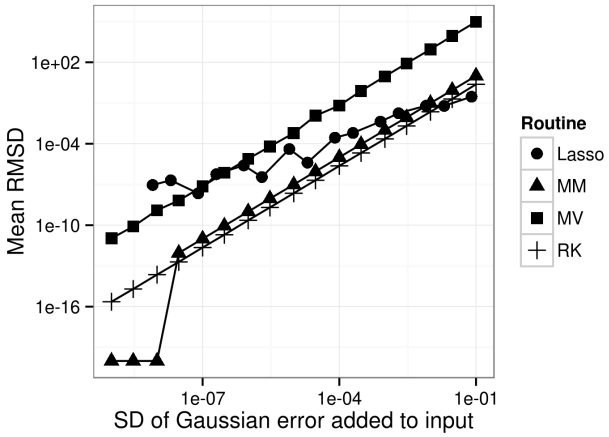

Fig. 8. Mean RMSD caused by adding Gaussian error to inputs

injection experiments needed to adequately characterize it. The visualizations provided by XRay are a specific choice for how such effects should be characterized: for each dynamic fault site classify by Abnormal Termination, Erroneous Results, Correct Results and when the outcome is Erroneous Results, compute the magnitude of the error. This section shows how XRay identifies the number of experiments needed for this particular characterization.

XRay quantifies the amount of relevant information contained in a set of fault injection experiments by modeling its visualizations in terms of a statistical model that takes the available information about a given error injection (e.g. scatterplot x-axis) and predicts the outcome of the error on the application (e.g. scatterplot y-axis). The model uses as input (i) ID of the dynamic fault site, (ii) ID of the static fault site, (iii) index of the flipped bit in the injected instruction's output. It then categorizes runs into the classes, "Abnormal Termination", "Erroneous Result" and "Correct Result". Finally, for the "Erroneous Result" runs it predicts the RMSD of the result error. Its structure is illustrated in Figure 9.

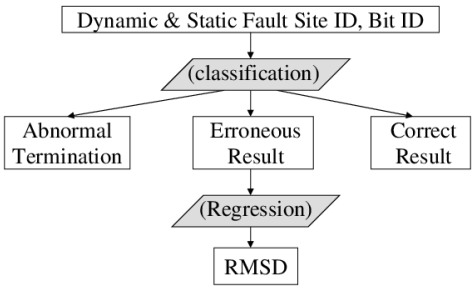

Fig. 9. Structure of the XRay evaluation models. Shaded procedures are where the tree model is applied.

The model's accuracy is evaluated using two metrics:

- 1st-level categorization model: mis-classification rate. Since we have 3 categories, a random guess would result in an error rate of 66.7%. With the knowledge of the training set, the tree model should produce a much smaller misclassification rate.
- 2nd-level regression model: R-Square, defined as $1 - \sum_{i=1}^{N}(\hat{y}_i - y_i)^2 / \sum_{i=1}^{N}(y_i - \bar{y})^2$, which describes how much of the variance in the data the model is able to capture. (The R-square is not applicable to the 1st-level classification.)

XRay selects the number of experiments incrementally, by performing more and more experiments and observing the effect of the additional training data on the accuracy of the model. For a given sample XRay performs a two-fold cross-validation for the model (train on half the data then predict for the other, and vice versa) to obtain the misclassification rate and R-square. When XRay finds the sample size where the accuracy of the model stops improving with increasing sample size, XRay stops the fault injection campaign since this number of samples is sufficient to characterize the relationship between the injection properties and application outcomes considered by XRay. Additional improvements in accuracy can only come from adding more features into the analysis, not by running more experiments.

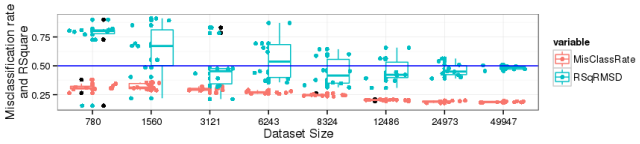

Fig. 10. Trend of R-square and misclassification rate as dataset size grows. (A random guess = misclassification rate of 66.7%)

Figure 10 illustrates the procedure using experiments on the Matrix Vector Multiplication routine, executed on 500x500 matrixes. As the number of fault injection experiments increases we see that both the misclassification rate and R-square drop steadily until they stabilize at a sample size of 49947 experiments. As the data shows this sample size is sufficient for the purposes of XRay's visualization and is much smaller than the $\sim 1e9$ experiments required to fully explore the experimental space. This is the sample size chosen for this routine and XRay employs the same procedure for all routines and applications.

VI. Conclusion

We present XRay, a tool that supports application developers in making applications resilient to errors induced by soft faults. XRay conducts fault injection campains and visually presents the relationship between the time of a fault and its effect on application results. This view enables developers to focus their efforts on the routines that are most critical to the application's overall resilience. Further, XRay uses statistical estimation to identify the number of fault injection experiments needed to produce accurate visualizations, giving developers confidence that the visualizations are a valid description of application behavior.

We demonstrated the use of XRay for the Lasso linear algebra application. This demonstration showed that

- Algorithm-specific error checkers are effective at detecting erroneous application results, as illustrated in our experiments with MVM and Rank-K update.
- By recovering from assertion failures and backing up data, we can salvage many runs of Cholesky Decomposition that would otherwise fail.
- Protection of the key routines identified by XRay significantly improves the resilience of Lasso to soft faults.

ACKNOWLEDGMENT

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 glad to be involved in KULFI and contribute our improvements back to it.

REFERENCES

[1] J. Elith and J. Leathwick. Boosted regression trees for ecological modeling.

[2] K.-H. Huang and J. A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. In *International Conference on Dependable Systems and Networks (DSN)*, pages 161–170, 2010.

[3] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[4] V. C. Sharma, A. Haran, Z. Rakamarić, and G. Gopalakrishnan. Towards formal approaches to system resilience. In *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2013. to appear.

[5] R. Tibshirani. http://statweb.stanford.edu/tibs/lasso.html.