Fault Modeling of Extreme Scale Applications using Machine Learning

Abhinav Vishnu [#1], Hubertus van Dam [#2], Nathan R. Tallent [#3], Darren J. Kerbyson [#4], and Adolfy Hoisie [#5]

[#1,3,4,5] *Pacific Northwest National Laboratory, Richland, WA 99352*

[#2] *Brookhaven National Laboratory, Upton, NY 11973*

***Abstract*—Faults are commonplace in large scale systems. These systems experience a variety of faults such as transient, permanent and intermittent. Multi-bit faults are typically not corrected by the hardware resulting in an *error*. This paper attempts to answer an important question: Given a multi-bit fault in main memory, will it result in an application *error* — and hence a recovery algorithm should be invoked — or can it be safely ignored?**

We propose an application fault modeling methodology to answer this question. Given a fault signature (a set of attributes comprising of system and application state), we use machine learning to create a *model* which predicts whether a multi-bit permanent/transient main memory fault will *likely result in error*. We present the design elements such as the fault injection methodology for covering important data structures, the application and system attributes which should be used for learning the model, the supervised learning algorithms (and potentially ensembles), and important metrics. We use three applications — NWChem, LULESH and SVM — as examples for demonstrating the effectiveness of the proposed fault modeling methodology.

I. Introduction

Faults are a norm in large-scale systems [1], [2], [3]. A fault in a device may result in a failure, which may potentially corrupt application data, resulting in an error. Modern systems experience various types of faults, such as transient, intermittent and permanent. Recent literature suggests that devices such as main memory suffer from various types of faults [4], [1], [2]. While single-bit faults are automatically detected and corrected, multi-bit faults are detected but not corrected. Several application writers have attempted to handle faults in these systems by proposing techniques for fault detection (such as correctness assertions/invariants) and implementing customized recovery algorithms [5], [6], [7], [8]. These algorithms dramatically reduce the impact of various fault types on the application correctness.

Let us consider an application, which observes a permanent main memory fault during its execution. Modern x86 processors provide Enhanced Machine Check Architecture (EMCA) to notify the occurrence of a hardware fault (both correctable and uncorrectable), as shown in Figure 1. Naturally, a fault which is automatically corrected by the hardware (blue box in Figure 1) does not require a corrective step(s) from the application. Uncorrectable faults at hardware are forwarded to OS/VMM layer. Faults which are uncorrectable at OS/VMM layer are eventually forwarded to the application layer.

A conservative approach to handling uncorrectable faults is to execute a recovery algorithm. The overhead of executing a recovery algorithm is non-negligible, and becomes increasingly prohibitive with escalating fault rates. However, not all uncorrectable hardware faults result in an application error — and thus executing a recovery algorithm is potentially wasteful. **Hence the important question is: Given a multi-bit fault in main memory, will it result in an application error — and hence a recovery algorithm should be executed — or can it be safely ignored?**

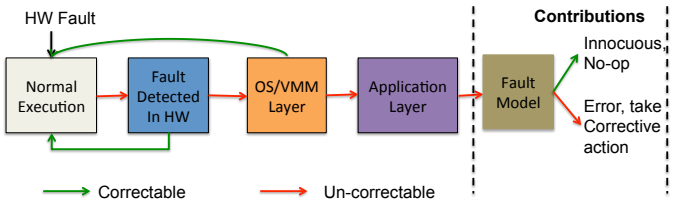

Fig. 1. Faults detected at HW, OS and Application Layer. Correctable faults are automatically fixed and uncorrectable faults result in using a recovery algorithm

A. Contributions

We make the following contributions in this paper:

- We cast the problem of classifying a fault signature (set of attributes comprising of system and application state) of a system and application as innocuous or error using *machine learning* methodology. Primarily, we use supervised learning methodology to create a fault model of several applications. Supervised learning methodology requires a *training set* (a set of samples) with a label (innocuous/error) for each sample. Each label represents the ground truth for the sample.
- To create a training set with ground truth, we inject permanent and transient multi-bit faults in the applications and observe the outcome of the fault. Unlike existing techniques — which typically use random fault injection — we record precise semantic information. The semantic information (which captures the temporal and spatial aspects) is then translated to a feature set. The combinatorial space of spatial (data structures in an application) and temporal (operations on data structures during the execution) aspects is very large. We propose techniques to prune this fault injection space.
- Another critical design element is feature engineering. We provide an in-depth discussion on selecting application-independent features — which makes the proposed

methodology attractive for other applications.

- We consider a total of seven supervised learning algorithms (Support Vector Machines (SVM), k-Nearest Neighbors) and ensemble methods (Adaboost, Bagging, Gradient Boosted Decision Trees, Random Forest, Extra Trees) for creating application fault models.
- We present solutions to addressing the issue of *imbalance* due to low cardinality of error samples in the datasets by using *under-sampling* and *over-sampling* techniques. The pivotal metric is the classification accuracy of error samples. We propose *imbalanced mixing* of error and innocuous samples for this purpose.
- We use several applications — Computational Chemistry (NWChem) [9], Shock Hydrodynamics (LULESH) [10] and Machine Learning (Support Vector Machines) [11] — as use cases of our fault modeling methodology. We observe that while the fault model is application specific, the methodology is generic and readily applicable to other applications.

Our evaluation using 4096 cores shows that the fault models for these applications can readily classify 97% error cases correctly (with 99% in several cases) and 82% of the innocuous cases correctly.

We expect the extreme scale application designers to benefit substantially from the proposed fault modeling methodology. An application writer can choose conservative/aggressive fault model depending upon fault rates and application properties. In many cases, the fault models will prevent unnecessary execution of a recovery algorithm — reducing time to scientific discovery.

The rest of the paper is organized as follows: In section II, we present a description of our problem and present a case for machine learning based fault modeling and a fault injection methodology. In section III, we present a brief description of three applications considered in this paper and fault injection methodology in section IV. In sections V and VI, we present a discussion on selecting important features and machine learning algorithms for generating the fault model. We present an evaluation of the proposed techniques in section VII, related work in section VIII and conclusions in section IX.

II. Problem Definition

In this section, we present a detailed description of our problem. We argue that a multi-bit fault in an application's (A) data structure (d) at an instance (t) will either result in an error or a no-op (innocuous). Previously, researchers have considered dividing these outcomes in other categories [5], [7]. As an example — for an iterative application solving a convergence problem (such as energy calculation in NWChem) — the outcome can be divided further in other categories. As shown in the figure 2, the top-left quadrant represents the innocuous category and the bottom quadrants show error categories. The top-right quadrant can itself be further subdivided. Let us consider triple-modulo-redundancy (TMR) as the baseline for handling multi-bit memory faults. Hence, an instance in top-right quadrant could actually be classified as an error, if its execution time exceeds 2x the execution time of the innocuous case.

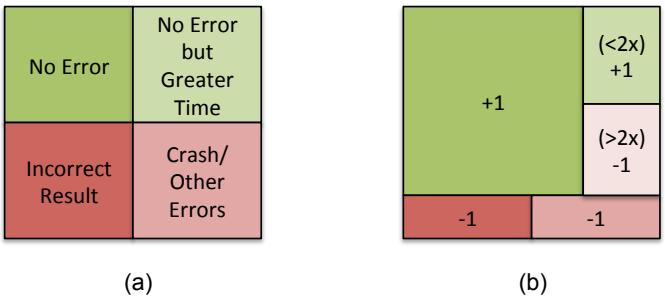

Fig. 2. (a) Possible outcomes of fault injection. (b) Annotation of each category as +1 (innocuous) or -1 (error/execution time $>$ 2x).

For generating an application fault model, we classify the quadrants using class labels. One possibility is to consider four classes (one-each for the quadrant shown in figure 2(a)). However, in practice, the cardinality of error cases is much smaller than of the innocuous cases [5], [6], [7]. Hence, we convert the four-class problem as a binary classification problem, as shown in figure 2(b) (the area occupied by each class also reflects a practical observation of errors in these classes). With this annotation, **we define the problem of an application fault modeling as a machine learning problem on a collection of observations (dataset) with innocuous (+1) and error (-1) outcomes**.

This problem definition leads to several important questions: 1) Where should the faults be injected? 2) How should the faults be injected — such that they resemble a multi-bit memory fault? 3) When should the faults be injected? 4) How to reasonably prune the combinatorial space of 1), 2) and 3), such that the dataset can be collected in a reasonable time? 5) What are the important features (application and system specific), which should be used for learning the fault model? and 6) What are the machine learning algorithms, which should be used while addressing the problem of *imbalance* in the dataset (due to lower cardinality of the error samples?)

We address these questions in the upcoming sections of the paper. We begin with a description of the three applications, which we have considered for evaluation in this paper — NWChem [9], [12], LULESH [10] and Support Vector Machines (SVM) [11], [13], [3].

III. Applications

A. NWChem

Northwest Chemistry (NWChem) [9] is a massively parallel general purpose computational chemistry application. It implements high accuracy algorithms such as Self-consistent Field (SCF), and Coupled Cluster (CC) methods. In this paper, we focus on the SCF algorithm — the *de facto* quantum chemistry algorithm.

There are eight primary data structures in SCF: Basis set, Geometry, Density Matrix, Integrals, Fock Matrix, Matrix Exponential, Orbital transform, and Orbital Orthonormalization. Let n represent the size of the molecule (the size of molecule is calculated using number of basis sets). The space complexity of Basis Set and Geometry is $\Theta(n)$, and hence they are replicated across processes. Other data structures are

distributed across processes. The Density Matrix, Fock matrix, the Matrix Exponential, the Transformed and Orthonormalized orbitals are square matrices and their space complexity is $\Theta(n^2)$. The Integrals form a fourth order tensor, resulting in a space complexity of $\Theta(n^4)$.

The majority of operations on these data structures are matrix transformations which have $\Theta(n^3)$ time complexity. However, as the molecule increases in size (more atoms) the average distance between atoms increases (figure 5) and the matrices become increasingly sparse. SCF uses an upper bound to eliminate the small integrals (generated using Gauchy-Schwarz inequality), which results in $\Theta(n^2)$ time complexity for the Integral calculation of large molecules. The overall time complexity for large molecules is $\Theta(n^3)$. We use a divergence in energy greater than 10^{-6} as an error.

B. LULESH

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) [10] is a proxy application for ALE3D full application code. It is one of the Department of Energy (DOE) co-design centers.

As shown in figure 3(a), LULESH implements the Sedov blast problem in three-dimensional space. The mesh is partitioned in domains — logically-rectangular collection of elements. Each element has eight corner points (also known as nodes). In LULESH, each node and element has specific properties. Each node has mass, acceleration, velocity and position in the Cartesian space. Each element has properties such as pressure, viscosity, energy and relative volume. Let s denote the problem size (input parameter to LULESH). For p processes (p is required to be cubic), we observe that the space-complexity of the nodes and elements is $\Theta(s^3 \cdot p)$.

Initially (at $t = 0$), a force is deposited at the origin. The objective of the algorithm is to calculate the energy by time-stepping. Specifically, at each time-step, *LagrangeNodal* (a set of functions to update nodes) and *LagrangeElements* (a set of functions to update elements) are executed. The application reports the final origin energy at the completion. We use energy divergence greater than 10^{-8} as an error (more details in LULESH document [10]).

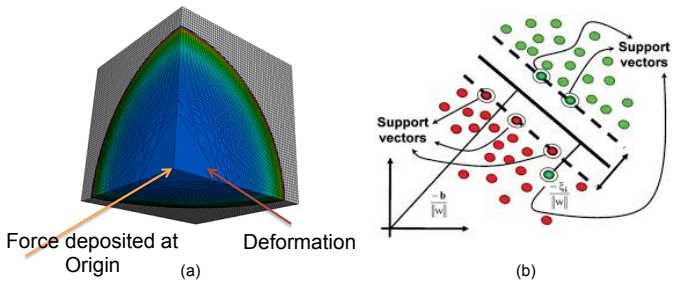

Fig. 3. (a) Deformation of hexahedrons in LULESH [10]. (b) Hyper-plane (solid line) and Support vectors in SVM. Green and Red points are samples in the two-dimensional Cartesian space.

C. Machine Learning: Support Vector Machines

Large scale Machine Learning algorithms are becoming popular in analyzing exorbitant volumes of data. Supervised algorithms — which perform classification/regression using a labeled dataset — are applied in many science domains. We specifically focus on SVM, since they provide very high accuracy — especially on non-linearly separable datasets.

We use a distributed memory SVM algorithm publicly available with Machine Learning Toolkit for Extreme Scale (MaTEx) [11]. MaTEx SVM is distributed Sequential Minimal Optimization (SMO) [14] — the most widely used SVM algorithm. There are several important data structures in SMO: The dataset (read-only), row-pointer (read-only with compressed sparse row (CSR) representation), α (Lagrange multiplier), y (label), s (set-info based on KKT conditions) and γ (gradient). Let us consider a dataset with m samples and n features. Let n_{avg} represent the average number of non-zeros in each sample. The space complexity of the dataset is $\Theta(m \cdot n_{avg})$; and row-pointer, α, y, s and γ are $\Theta(m)$. Row-pointer, set-info, y and column-value in CSR representation are stored as integers and other data structures are stored as doubles.

There are two main functions in SMO: *takestep* (performs gradient descent to find the Lagrange multipliers for next step) and *update-gradient* (use the Lagrange multipliers for updating gradient). The update-gradient function is the most computationally expensive part of the calculation. It conducts a series of compute intensive kernel-function calculations: ($\Phi(x, y) = e^{\frac{-1}{2 \cdot \sigma^2} \cdot ||x-y||^2}$). At each iteration, every sample is accessed (read-only) using row-pointer, and γ is updated. α and s are updated infrequently. The algorithm reports a convergence threshold (β). We use a divergence greater than 10^{-3} as an indicator of incorrect convergence.

IV. Fault Injection Methodology

Most researchers conduct software based fault injection for emulating the impact of faults in main memory hierarchy. Recently proposed tools such as Low level Fault Injector (LLFI) [15] provide compiler-based fault injection. Dynamic fault injectors such as Pin Fault Injector (PinFI), and BIFIT [16], (based on Intel Pin) [17] provide dynamic instrumentation based fault injection. Virtualization based fault injectors such as F-SEFI [18] provide fault injection without application code changes. Several other researchers have considered application-specific fault injection techniques [8], [7], [5].

We considered each of these approaches for fault injection in our applications. While parts of the previously presented approaches were applicable to our use cases, we observed a few limitations. As an example, F-SEFI is based on QEMU hypervisor — which is not supported on high performance platforms. LLFI allows a user to specify precise code lines/functions for fault injection. However, it is restricted to gcc and does not capture the temporal aspect (when to inject a fault) effectively. BIFIT and PinFI are dynamic instrumentation based fault injectors. Dynamic instrumentation based tools can possibly incur non-negligible overhead. In several cases, the overhead can create a false positive by increasing the execution time to be greater than 2x (figure 2). In addition to these limitations, these fault injectors consider only random bit-flips (one or more) during one execution.

Furthermore, we are interested in **capturing precise semantic information (state of the data structures and**

temporal information) by inserting main memory multi-bit flips, which is not considered by existing fault injection tools. Given these limitations, we develop a low-overhead application-specific fault injector, similar to previously proposed approaches [8], [7], [5].

A scalable fault injector should have the following properties: It should cover the *spatial* (data-structures and functions which operate on the data structures) and *temporal* (when the fault is injected) effectively. At the same time, the parameter sweep due to a combination of spatial and temporal aspects should be pruned — such that the collection of observations (dataset) can be collected in a realistic time. We consider each of these design challenges in next sections.

A. Fault Injection: Capturing Data Structures (Spatial Aspects)

An important design element of our fault injection methodology is to capture an application's spatial aspects by fault injection in the data structures. The spatial aspects include the data structures and operations (as part of various functions) on these data structures. As presented earlier in section II, each application considered in this paper has several key data structures. For example, SVM has six key data structures, NWChem has eight key data structures and LULESH has a total of fourteen key data structures. The proposed fault injection code considers perturbation in each of these data structures.

For each data structure, a perturbation is possible in any element. In the parallel implementations of the applications — as considered in this paper — this would refer to a perturbation in an index of a data structure within a process. For each element, there are 32/64 bits in which a perturbation is possible. Each application uses several functions to read/update one or more data structures. To capture the effect of different functions, a perturbation may be needed in each of these functions.

Hence, for an application (A) with (N) data structures, let n_i denote the size and b_i represent the number of bits in i-th data structure. Let f_i denote the number of functions in which a data structure can be updated. Hence the size of the combinatorial space for perturbation is $\sum_{i=1}^{N} n_i \cdot b_i \cdot f_i$. Clearly, this perturbation space is huge, and we need to prune this search space to collect the observations in a realistic time, while minimizing the impact of pruning.

B. Fault Injection: Capturing Temporal Aspects

Another important element of fault injection is — *When to inject the fault?* — in terms of relative time spent in executing the application. To address this problem, we first classify the applications in two categories: *convergence* and *time-stepping*.

A convergence problem executes till the convergence criteria are satisfied. Many high-end computing problems fall in this category, such as NWChem, Partial Differential Equations (PDEs), Machine Learning algorithms such as SVM, PageRank and k-means. A time-stepping problem executes for a pre-defined number of steps. As an example, LULESH is a time-stepping problem.

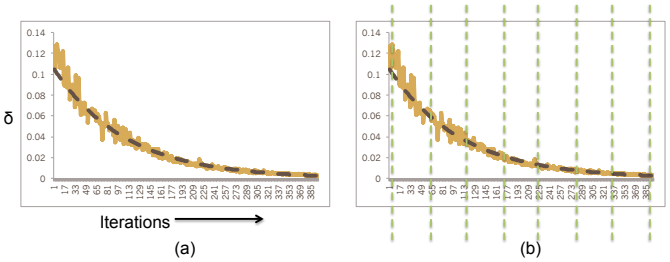

Fig. 4. A line-plot of δ with iterations for CERN's Higgs Boson Machine Learning Challenge Dataset. The curve can be approximated as an exponential decay function (shown with dotted black line). The time per iteration is constant (b) shows the bucketization of δ in seven buckets, as proposed in this paper

Let t_{end} represent the execution time of an application. Hence, it is possible to inject a multi-bit fault at any point from ($t = 0 \ldots t_{end}$). However, using time as a variable for fault injection requires an application model to predict the execution time — which is difficult for many convergence problems. Hence, using time as a variable will generate inaccuracies in fault modeling.

We propose an alternative to solving this problem. We use δ — the current deviation from solution (An example using CERN Higgs Boson dataset with SVM is shown in Figure 4(a)) — for capturing temporal aspects. We divide the current value of δ to the solution (for example 10^{-3} for SVM). The primary advantage of this approach is that we do not need to rely on performance prediction to capture the temporal aspect of an application. For time-stepping algorithms — such as LULESH — the number of iterations are calculated before the time-stepping or provided as an input, which can be used as an indicator of temporal aspects.

C. Pruning the Fault Injection Space

As observed in the previous sections, the combinatorial space of spatial and temporal aspects is very large. For an application which executes for a very long period of time, it may be infeasible to explore each possibility. We consider pruning of spatial and temporal space, as presented in the upcoming sections.

1) Pruning Spatial Aspects: Assuming b_i bits in a data structure, there are b_i fault injection points (not considering the temporal aspects). We consider pruning the spatial aspects by selecting a few elements — randomly — in each data structure. We first randomly select a process and then randomly select an element within the process for a multi-bit fault injection. For each data structure, this provides a method to collect a significant number of samples, which represent a fault injection in the data structure.

For the applications considered in this paper, we observed that data structures are stored as doubles/long — implying 64-bits for fault injection. We discretize the 64-bits in several *bit-buckets*. As an example, we use four buckets of 16-bits each, such that we can capture the effects of least and most significant bits effectively. Specifically for doubles, 11-bit exponent and sign-bit are captured as a part of one bucket (along with 4-bits of mantissa). For each application we

consider several input decks. As an example, we considered seven datasets for SVM, and five each for NWChem and LULESH.

2) Pruning Temporal Space: For convergence problems, we considered discretization using δ-*buckets*. We observed an interesting trend in δ_{iter} (value of δ at an iteration) for NWChem and SVM. Using CERN's Higgs Boson Dataset, we observed that the δ_{iter} follows an exponential decay function (Figure 4(b)). Similar trends were observed for NWChem. Hence, we pruned by the search space by defining the buckets using a boundaries of $(\delta, 2 \cdot \delta \ldots 128 \cdot \delta)$. This allows us to prune the temporal space in a logarithmic number of buckets. For time-stepping problems, we simply divide the temporal space using equal size iteration-buckets. As an example, with LULESH, we use four iteration buckets.

D. Putting it All Together

An important concern with pruning is the possible elimination of an important observation. We argue that with the current pruning methodology, we are able to collect a statistically significant number of observations. For example, in SVM with seven δ-buckets and four bit-buckets, we are able to consider at least 28 fault injections for each data structure. Similarly for each δ-bucket, we are able to consider at least 24 fault injections in the SVM applications. With these combinations of fault injections, we are likely to capture the statistically significant samples, and corner cases as well.

V. Feature Engineering

A critical part of proposed fault modeling methodology is feature engineering. For fault modeling, we consider two feature categories, which are application-independent and application-dependent. Ideally, we would like an application to be mostly dependent up on application-independent features — so that the proposed fault modeling methodology can be applied directly to other applications. However, application-dependent features can play an important role in fault modeling.

A. Application-Independent Features

In the previous section, we considered spatial and temporal aspects of applications for fault-injection. Within spatial features, we considered fault injection in the data structures and a bit within a randomly selected index of the data structure. Hence, we propose to use two features — *data structure index* and *bit-bucket* — as features for fault modeling. As an example, with LULESH there are fourteen data structures. Hence, the cardinality of data structure index feature for LULESH is fourteen. Similarly, we consider bit-bucket as another feature with a cardinality of four (one bucket for 16-bits each).

We propose to use another feature corresponding to δ-bucket, which captures the temporal aspect of the applications. For LULESH, the cardinality of δ-bucket is four (equally dividing the number of iterations in four equal size buckets). For convergence problems — NWChem and SVM — the δ-bucket is seven (by using an exponential decay function). Hence, there are a total of three application-independent features for fault modeling.

B. Application-Dependent Features

In this section, we consider application-specific features. In practice, it is not possible to define the *only* set of features, which should be used for learning the fault model. We substantiate application-dependent feature selection using intuitive reasoning.

1) NWChem: In NWChem [9], we consider two application specific-features: *sparsity* and *index-classification*. In NWChem, sparsity is defined as the number of non-zeroes in the Overlap matrix of an input deck. To understand the importance of sparsity visually, let us consider the molecular structure of the five input decks we have used for fault modeling of NWChem. These structures are shown in figure 5. Let us consider the diamond molecular structure. We observe

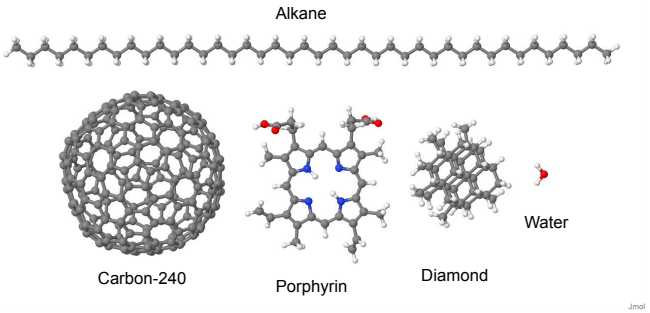

Fig. 5. NWChem Input decks considered in this paper with different sparsity patterns

that the the structure is very compact resulting in a high number of bonds per atom. As the other extreme case — considering the alkane molecule — we observe that the molecular structure is linear, resulting in higher sparsity in comparison to the diamond structure. The intuition is that sparsity of a molecule can play an important role in the impact of fault injection on the overall energy calculation. Since these are rather extreme cases of sparsity, we use three other input decks to cover intermediate sparsity patterns, readily observed in other molecules such as Porphyrin, Carbon-240 and Water.

The other feature specific to NWChem is index-classification. In several data structures within NWChem, an index can be classified as a *diagonal*, *off-diagonal* or *not-applicable* (since it is a vector). In several parts of the calculation, diagonal elements are treated differently than non-diagonal elements. It is intuitive to use this as a feature for learning the fault model of an application. Hence, including the application-independent features, we use a total of five features for fault modeling of NWChem.

2) SVM: Unlike NWChem, the SVM application operates on sparse representations of the dataset (we specifically use compressed sparse row (CSR) format). Since SVM operates of a collection of observations, it is intuitive to consider two features corresponding to the input — *number of samples* and

dimensionality. Using these two as separate features allows us to capture the problem size indirectly.

However, dimensionality of a dataset is not an accurate reflection of the number of non-zeros in a sample. As an example, the dimensionality of malicious URL dataset is 3.2 million. However, the maximum number of non-zeroes in a sample is less than ten thousand. In other datasets such as CERN Higgs Boson Machine Learning Challenge dataset [19], the number of dimensions is 32, and the dataset is dense. A simple feature to capture both sparse and dense datasets is to use the maximum number of non-zeroes in the dataset. We use this as the feature for fault modeling of SVM.

3) LULESH: Similar to NWChem, LULESH operates on dense data structures, which are represented in domains, each having several elements and nodes. However, unlike NWChem, the sparsity is not evident in LULESH. The typical input deck usually conducts the sedov blast simulation on a uniformly distributed material. Unlike NWChem, the simulation does not provide special properties to the diagonal elements. However, an important feature to consider is the problem size of the application. We use this as the additional feature for fault modeling of LULESH.

VI. Learning the Fault Model using Machine Learning

An important element of our fault modeling methodology is to use machine learning (ML) algorithms for generating the fault models. We have considered several supervised ML algorithms (base and ensembles) and one unsupervised learning algorithm. The objective of this section is to apply these ML algorithms to the datasets collected in the previous sections. While these algorithms are applied to the three applications considered in this paper, the properties of the datasets are observed with other applications as well [7].

Figure 6 shows the steps in generating the datasets using fault injection experiments. Specifically, we collect the dataset (Figure 6(b)) by fault injection (figure 6(a)) and then shuffle the dataset to remove any bias due to the ordering in fault injection experiments (figure 6(c)). Supervised machine learning algorithms — the backbone of fault modeling in this paper — typically use a training set and a testing set. Figures 6(d-g) show several possibilities of splitting the datasets in training and testing sets. In figure 6(d), we split the shuffled dataset such as by using 20% of the samples for training and 80% for testing sets. Figure 6(e) shows the case, where we select equal number of innocuous (green) and error (red) observations for training and testing sets, respectively. The splits shown here are just a few possibilities. An application writer may decide to use other splits, as necessary.

Figure 6(f) shows the case, where the testing set consists of only the error cases. For fault modeling, the pivotal metric is **how accurately can the fault model predict the error-cases?** Another important aspect is the accuracy of prediction on the innocuous cases. A very conservative classifier can accurately classify all the error cases, while also classifying the innocuous cases as error — which is not attractive. Hence, we define the metric to be the **high accuracy of predicting error cases (true positives), while minimizing the mis-classification of innocuous cases**.

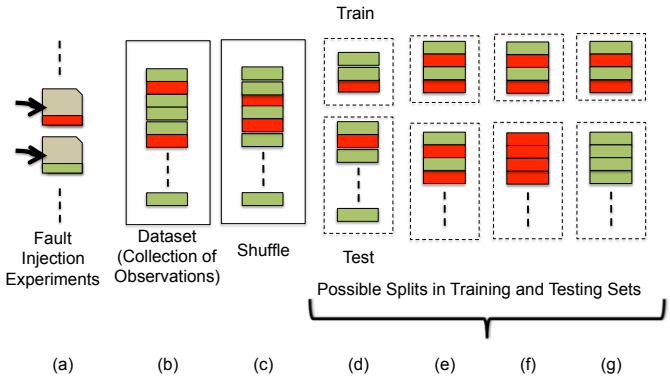

Fig. 6. The steps in fault injection, data collection, shuffling and splits of the datasets collected using fault injection experiments

A. Machine Learning Algorithms

In this section, we provide a brief overview of the machine learning algorithms we have used for learning the fault models.

1) Support Vector Machines (SVM): SVM is the *de facto* ML algorithm. It works well on non-linearly separable datasets, is independent of dimensions, and provides excellent accuracy.

2) k-Nearest Neighbors (KNN): k-nearest neighbors is one of the classical ML algorithms, which classifies a sample using the majority class of its neighbors. After finding k nearest neighbors, the algorithm selects the most frequent class of these neighbors.

3) AdaBoost: This ML algorithm iteratively improves the accuracy by providing more weights to the mis-classified samples. Since our datasets are imbalanced, AdaBoost has the potential to generate a better classifier, since it can give better weight to mis-classified samples.

4) Bootstrap Aggregation Techniques (Bagging): Bagging is an ensemble technique, where a dataset is randomly partitioned and separate classifiers are created using these partitions. These individual classifiers are then combined (using averaging/voting) to select the best classifier. We consider bagging with SVM as the base classifier.

5) Gradient Boosted Decision Trees (GB): GB is an ensemble technique which allows several weak learning based models to be combined together. Since this method allows the use of an arbitrary cost function, it has the potential to find a better classifier than simple boosting/bagging.

6) Random Forests (RF): A classical issue with ML algorithms is over-fitting — fitting the classifier very closely to the training set. Randomized forest address this issue by creating a set of decision tree based classifiers and averaging them. We consider this to be an important ML algorithm, since our testing set has slightly different properties than the training set.

7) Extremely Randomized Trees (Trees): Unlike Random Forests, this classifier uses randomized decision trees as the base classifier. A randomized decision tree can improve the

classification accuracy, especially if the cardinality of each feature is different. For example, in LULESH, the cardinality of data structure index is fourteen while δ-bucket is four. ET creates several individual classifiers, and then uses averaging to combine the individual classifiers.

8) One-Class Support Vector Machines (One-Class): Up to now, we have only considered supervised learning algorithms for fault modeling. One-class SVM is an *unsupervised* learning algorithm, which creates a decision surface assuming that the training set has only one class.

While it is counter-intuitive to use an unsupervised method — when the ground truth is available — there is a significant advantage to using one-class SVM. Specifically, this method can generate a conservative classifier surface by training on the error samples. We refer to this as a *conservative* classifier, since it can readily classify the error samples correctly, while potentially resulting in an accuracy loss for innocuous samples.

VII. Evaluation

A. Preliminaries

1) Experimental Testbed: We use the PNNL Cascade Supercomputer [20], which is equipped with Intel Sandybridge CPU and InfiniBand FDR interconnect. The performance evaluation uses up to 4096 cores (256 compute nodes). We use MVAPICH2-2.0.1 for performance evaluation.

B. Fault Types and Handling Class Imbalance

We demonstrate the results by emulating double-bit **permanent** and **transient** faults in the main memory hierarchy.

We observed that for NWChem, LULESH and SVM, the total number of error cases is less than 5% of the innocuous cases. Typically, ML algorithms work well on balanced datasets, where the number of samples of each class are roughly equal. We use two techniques to address this problem: *under-sampling* and *over-sampling* of samples in the dataset. In under-sampling, we use a subset of the dataset, which has roughly equal number of error and innocuous cases for the training set. In over-sampling, we consider an *imbalanced mixing* of the samples.

Specifically, we are interested in very high accuracy for the error samples — potentially at the loss of accuracy for innocuous samples. Hence, we consider several imbalanced mixes such as 20-80 (20% innocuous samples and 80% error samples in the training set), such that the classifier is biased towards the error cases. We consider other imbalanced mixes such as 30-70 and 40-60 as well. We use the ML algorithms publicly available in `scikit` [21] for learning the fault models.

1) Basic Performance: We observed that for each application, generating the fault model takes $\approx$ 10 seconds and classification takes $\approx$ 3 seconds. We also observed that emulating a multi-bit fault did not incur overhead, because a majority of fault injections were innocuous and did not affect the execution time of the application. Hence, we can attribute the degradation of execution time to the application properties only.

TABLE I
Input decks for SVM and hyper-parameter settings

Name	Training Size	Testing Size	C	σ^2
Forest	581012	N/A	10	4
Higgs	250000	N/A	10	4
real-sim	72309	N/A	10	4
MNIST	60000	10000	10	25
cod-rna	59535	271617	32	64
Adult-9 (a9a)	32561	16281	32	64
Web (w7a)	24692	25057	32	64

For NWChem using multi-bit permanent faults, we observed that 15% of the overall cases resulted in an execution time of up to 20x, while still converging correctly. For multi-bit transient faults, the trend was observed for 11% of the overall cases. On further inspection, we observed that the fault injection caused the application to diverge significantly from the optimal solution, and each iteration took longer due to additional internal checks in NWChem. We classified these samples as error, as discussed earlier in section II. We did not observe these cases for SVM and LULESH.

C. Detailed Applications Results

1) SVM: Table I shows the datasets which we have used for fault injection in SVM. Figures 7 and 8 shows the classification accuracy for SVM using the ML algorithms (1 is the highest possible accuracy). We show the results with 20-80 mix, since they provided the best overall classification accuracy. We observe that for several ML algorithms can achieve 99% accuracy for error classification, and 78% accuracy for innocuous cases with RF (permanent faults). For multi-bit transient transient faults, the peak accuracy is 99% and 63%. This is largely because the number of error cases with transient faults are lesser than permanent faults. With imbalanced mixing, the classifier mis-classifies more innocuous cases as error. However, with the fault models, many multi-bit cases can still be classified as innocuous — avoiding execution of costly recovery algorithm.

We anticipated a few trends such as a multi-bit fault injection in an integer data structure would result in an error. However, this trend was not evident, with an exception of row-pointer, where the application would terminate abruptly. We also expected that multi-bit fault injections (both permanent and transient) higher bit-buckets (which include exponent) to always result in an error. We did not observe this pattern as well. In many cases, a fault injection in higher order bit-buckets simply resulted in mis-classification of a sample as a non support vector, which did not affect the convergence criteria.

2) NWChem: Figures 9 and 10 show the results for NWChem with double-bit permanent and transient faults, respectively. We used five input decks with different sparsity patterns as shown in figure 5. We observed that most of the ML algorithms provide 99% accuracy for error cases (GB provides excellent accuracy for both fault types). For innocuous cases, the observed accuracy is $\approx$ 65%, which implies that the recovery algorithm is needlessly executed for roughly 35% of the innocuous faults.

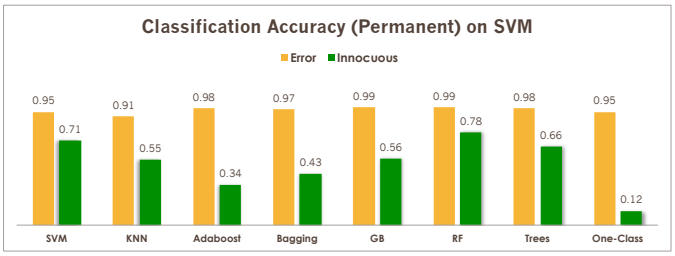

Fig. 7. Classification accuracy of ML algorithms on SVM using 20-80 imbalanced mixing of samples. 1 is the highest possible accuracy.

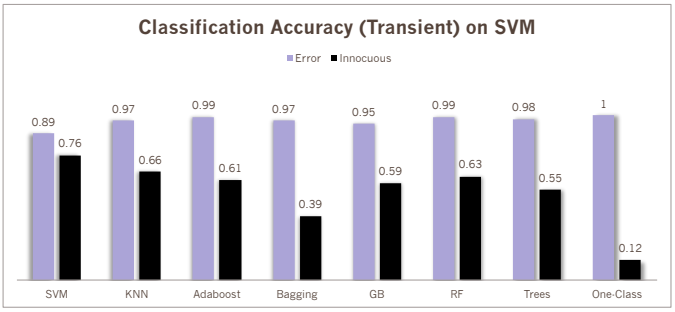

Fig. 8. Classification accuracy of ML algorithms on SVM using 20-80 imbalanced mixing of samples.

From SVM and NWChem results, we can conclude that it is difficult to predict apriori the suitability of an ML algorithm for fault modeling of an application. (For example RF provides best accuracy for SVM, however the best ML algorithm for NWChem is GB). This justifies using several ML algorithms for generating the fault models. In many cases, it is the property of the dataset, which identifies its applicability to the machine learning algorithm.

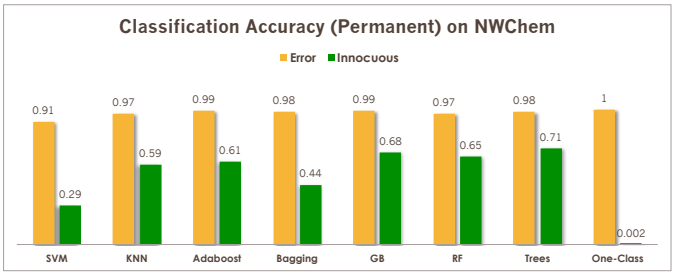

Fig. 9. Classification accuracy of ML algorithms on NWChem using 20-80 imbalanced mixing of samples.

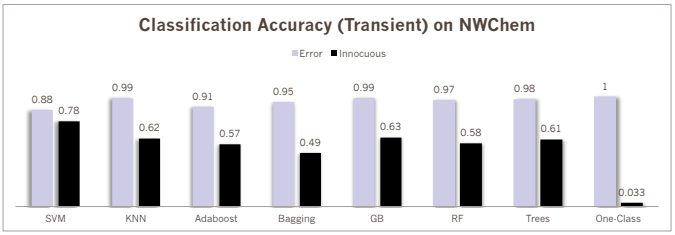

Fig. 10. Classification accuracy of ML algorithms on NWChem using 20-80 imbalanced mixing of samples.

3) LULESH: Figures 11 and 12 shows the classification accuracy of LULESH for double-bit permanent and transient faults, respectively. We have use three input $(20^3, 30^3, 40^3)$ in weak scaling mode (64 - 4096 processes). We observe that GB performs the best for permanent faults and transient faults, while providing an accuracy of 69% and 62%, respectively.

Across the three applications, we observe that it is hard to predict the efficacy of an ML algorithm, it depends up on the properties of the dataset itself. Typically ensemble based techniques perform better than base-classifiers, as we have readily observed for several applications.

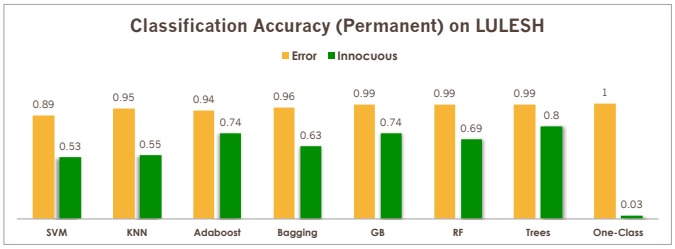

Fig. 11. Classification accuracy of ML algorithms on LULESH using 20-80 imbalanced mixing of samples.

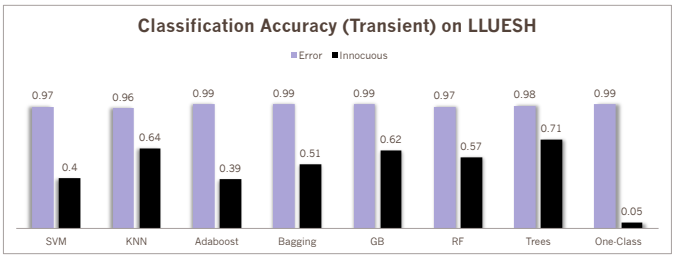

Fig. 12. Classification accuracy of ML algorithms on LULESH using 20-80 imbalanced mixing of samples.

4) Feature Importances: An important aspect of our evaluation is the importance of different features in fault modeling. Figure 13 shows these results. We observe that for SVM and NWChem, the data structure (black bar) is the most important feature. For NWChem, application sparsity — as discussed in section V is very important, with bit-bucket being the third most important feature. In NWChem, typically a contribution to a matrix (such as Fock Matrix) is a contribution from several matrices. With increasing sparsity — readily observed for larger molecules — the fault injections in elements for sparse molecules will not result in a significant change to the outcome of the energy. However, for more dense molecules, such as diamond, a fault injection could possibly result in a significant error.

For LULESH, problem size turns out to be the most important feature. In general, but not necessarily, the effect of permanent faults with increasing problem size was reduced. We observe that the impact of other features such as bit-bucket and δ-bucket is much lesser for LULESH.

5) Discussion on Bit-bucket: In modern architectures, an 8-bit ECC is provided for each 64-bits. Essentially, in practice, bit-bucket is not available. While we initially considered this to be a major issue, as shown in figure 13, bit-bucket is rarely

the most important feature for fault modeling of applications. Hence, we do not expect the unavailability of bit-buckets to significantly affect the accuracy of the proposed fault modeling methodology.

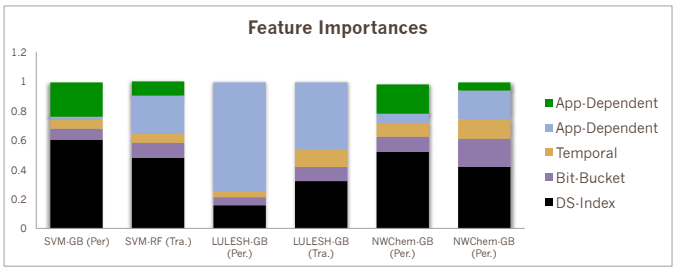

Fig. 13. Importances of different features in the three applications. Data structure index is an important feature, while several application-dependent features are important as well (problem size in LULESH and sparsity in NWChem)

D. Discussion

We consider the results from fault modeling to be very encouraging. The methodology and experimentation described can be used for other applications for creating a series of fault models such as aggressive (as discussed in previous sections) and conservative by an *imbalanced* mixing of samples from error and innocuous class.

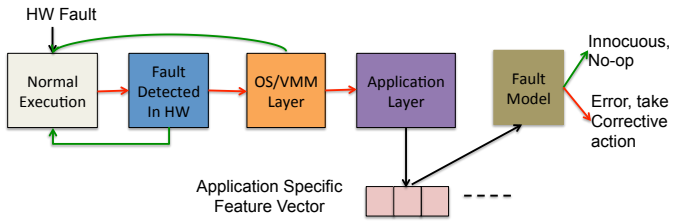

Fig. 14. A practical execution of the fault models generated using the proposed methodology

In Figure 14, we demonstrate a practical usage of the fault models using EMCA, which automatically corrects faults using hardware based techniques. However, if a fault is uncorrectable, it is forwarded to the OS/VMM layer, and eventually to the application layer, if uncorrectable at OS/VMM layer. Consider a fault handler, which handles the uncorrectable faults at an application layer. When the handler is invoked, the physical address of the fault is passed as a parameter, which can be readily converted to the virtual address (assuming no-swap and intact virtual-physical address translation). Hence, we can calculate the data-structure index from this translation. Other features such as δ-bucket, and application-dependent features (section V) can be calculated as well. We can create a feature vector using these values and apply one (or possibly more) fault models. These fault models can be used to determine, whether a corrective action needs to be taken or it can be ignored, as shown in figure 14.

VIII. Related Work

Several researchers have considered fault tolerance for large scale systems [22], [3], [23], [24]. We specifically focus on fault injection and fault modeling research.

Many researchers have considered fault injection tools such as LLFI [15], PinFI [17], [15], BIFIT [16], F-SEFI [18]. LLFI provides compiler based fault injection which allows a user to inject faults at specific functions/operations in a compiler. F-SEFI uses QEMU (a hypervisor) for fault injection — which is not necessarily available for high-end systems. BIFIT and PinFI provide dynamic instrumentation based fault injection. BIFIT and PinFI lose semantic information, which is required for fault modeling methodology proposed in this paper. Other researchers have considered application-specific fault injection [5], [25], [26], [27], [28], [29], [30], [31], which is similar to the approach presented in this paper.

Several other researchers have considered fault modeling at program [32], data [33] and architectural level [34] . For example, program vulnerability factor (PVF) [32] defines vulnerability of a software resource given a fault in a hardware resource. An application writer can use PVF to understand the relative vulnerability of the application to other applications. While PVF is an indicator of program's vulnerability to soft errors — being a scalar — it does not capture the multiple dimensions such as application-independent and dependent features, like considered in this paper. Data Vulnerability Factor (DVF) [33] calculates the vulnerability of individual data structures in an application. However, it relies indirectly on access patterns of various data structures to calculate vulnerability. We argue that access patterns is not a complete indicator of vulnerability of an application to a fault. For example, in the SVM application considered in this paper, a fault in the row-pointer data structure will likely result in an error, while a fault in the SVM dataset is less likely to cause an error, although, row-pointer is smaller in size, accessed similarly during the kernel calculations. Architecture vulnerability factor (AVF) [34] calculates the probability that a fault in a hardware structure will result in an error. Unlike AVF — which is a scalar — we consider several dimensions — referred to as features in this paper for computing a fault model, which can be used for classifying a fault as an error or innocuous.

IX. Conclusions

In this paper, we have created fault models to answer an important question: Given a multi-bit fault in main memory, will it result in an error or can it be safely ignored? We have used a machine learning methodology to answer this question. There are several important elements in this methodology such as considering spatial and temporal fault injection space and pruning it such that a collection of observations can be obtained in a realistic time. We have presented the limitations of the existing fault injection tools, which are not able to capture the critical semantic information required for our fault modeling. We have considered other aspects such as the important features — application-independent and application-dependent — which should be used for learning the model. We have looked at the properties of the applications considered in this paper, and provided an intuitive justification of the features. We have considered the *imbalance* problems in

the datasets and proposed to use *under-sampling* and *over-sampling* techniques to address them. We have considered seven supervised learning algorithms (base and ensembles) and one unsupervised learning algorithm for our purpose. We have evaluated our methodology using three applications — NWChem (computational chemistry), LULESH and Support Vector Machines on 4096 processes. We have used several input decks ranging in molecule sparsity for NWChem, several problem sizes for LULESH and datasets such as CERN's Higgs Boson Machine Learning Challenge dataset and Forest Cover for SVM.

By imbalanced mixing of the error (fault injections that result in an error) and innocuous (fault injections that do not result in an error) cases — such that the classifier is biased towards error cases — we are able to classify 99% of the error cases correctly for multi-bit permanent and transient faults, while classifying more than 60% of the innocuous cases correctly. This implies that when a double-bit fault occurs, the application needlessly executes a recovery algorithm only 40% of the time, in contrast to unconditionally executing a recovery algorithm at every double-bit fault in main memory.

We expect the contributions from this paper to benefit the large scale application researchers immensely. Using the proposed methodology, the application researchers can create fault models of their applications (both conservative and aggressive) and use them to classify a multi-bit memory fault as error/innocuous at runtime. In many cases, the fault models will prevent unnecessary execution of a recovery algorithm — significantly reducing the time to scientific discovery.

X. Acknowledgement

We would like to thank Analysis in Motion (AIM) Laboratory Directed Research and Development (LDRD) initiative for supporting this research.

References

[1] V. Sridharan and D. Liberty, "A study of dram failures in the field," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 76:1–76:11.

[2] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory: Positional effects in dram and sram faults," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 22:1–22:11. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503257

[3] A. Vishnu, H. Van Dam, W. De Jong, P. Balaji, and S. Song, "Fault Tolerant Communication Runtime Support for Data Centric Programming Models," in *International Conference on High Performance Computing*, 2010.

[4] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *Proceedings of the International Conference on Dependable Systems and Networks*, ser. DSN '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 249–258. [Online]. Available: http://dx.doi.org/10.1109/DSN.2006.5

[5] H. J. J. van Dam, A. Vishnu, and W. A. de Jong, "A case for soft error detection and correction in computational chemistry," *Journal of Chemical Theory and Computation*, vol. 9, no. 9, 2013.

[6] H J. J. van Dam, A. Vishnu and W. A. de Jong, "Designing a scalable fault tolerance model for high performance computational chemistry: A case study with coupled cluster perturbative triples," *Journal of Chemical Theory and Computation*, vol. 7, no. 1, pp. 66–75, 2011.

[7] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz, "Fault resilience of the algebraic multi-grid solver," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304590

[8] T. Davies and Z. Chen, "Correcting soft errors online in lu factorization," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 167–178. [Online]. Available: http://doi.acm.org/10.1145/2462902.2462920

[9] R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. P. Straatsma, T. L. Windus, and A. T. Wong, "High Performance Computational Chemistry: An Overview of NWChem, A Distributed Parallel Application," *Computer Physics Communications*, vol. 128, no. 1-2, pp. 260–283, June 2000.

[10] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[11] MaTEx, "Machine Learning Toolkit for Extreme Scale ," http://hpc.pnl.gov/matex.

[12] E. Aprà, A. P. Rendell, R. J. Harrison, V. Tipparaju, W. A. deJong, and S. S. Xantheas, "Liquid Water: Obtaining The Right Answer For The Right Reasons," in *SuperComputing*, 2009.

[13] A. Vishnu, M. Koop, A. Moody, A. Mamidala, S. Narravula, and D. K. Panda, "Topology Agnostic Hot-Spot Avoidance with InfiniBand," in *Concurrency and Computation: Practice and Experience, Special Issue of Best Papers from CCGrid '07*, 2008.

[14] J. C. Platt, "Advances in kernel methods," 1999, ch. Fast Training of Support Vector Machines Using Sequential Minimal Optimization.

[15] M. R. Aliabadi, K. Pattabiraman, and N. Bidokhti, "Soft-llfi: A comprehensive framework for software fault injection," in *25th IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Naples, Italy, November 3-6, 2014*, 2014, pp. 1–5.

[16] D. Li, J. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, 2012.

[17] G. Lueck, H. Patil, and C. Pereira, "Pinadx: An interface for customizable debugging with dynamic instrumentation," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12, 2012.

[18] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, 2014.

[19] HiggsML, "Higgs Boson Machine Learning Challenge," http://kaggle.com/c/higgs-boson.

[20] PNNL Cascade Supercomputer, "EMLS," cascade.emsl.pnl.gov.

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[22] W. Gropp and E. Lusk, "Fault Tolerance in Message Passing Interface Programs," *International Journal on High Performance Computing Applications*, vol. 18, no. 3, pp. 363–372, 2004.

[23] Network-Based Computing Laboratory, "MVAPICH/MVAPICH2: MPI-1/MPI-2 for InfiniBand and iWARP with OpenFabrics," http://mvapich.cse.ohio-state.edu/.

[24] OpenMPI, "Open Source High Performance Computing," http://www.open-mpi.org/.

[25] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS '08, 2008.

[26] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Characterizing the impact of soft errors on iterative methods in scientific computing," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11, 2011.

[27] A. Moody, G. Bronevetsky, K. Mohror, and B. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *SuperComputing*, 2010.

[28] A. Vishnu, A. Mamidala, S. Narravula, and D. K. Panda, "Automatic Path Migration over InfiniBand: Early Experiences," in *Proceedings of Third International Workshop on System Management Techniques, Processes, and Services, held in conjunction with IPDPS'07*, March 2007.

[29] A. Vishnu, P. Gupta, A. R. Mamidala, and D. K. Panda, "A Software Based Approach for Providing Network Fault Tolerance in Clusters with uDAPL Interface: MPI Level Design and Performance Evaluation," in *SuperComputing*, 2006, pp. 85–96.

[30] A. Vishnu, S. Song, A. Marquez, K. Barker, D. Kerbyson, K. Cameron, and P. Balaji, "Designing energy efficient communication runtime systems: a view from pgas models," *The Journal of Supercomputing*, vol. 63, no. 3, pp. 691–709, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11227-011-0699-9

[31] ——, "Designing energy efficient communication runtime systems for data centric programming models," in *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*, ser. GREENCOM-CPSCOM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 229–236. [Online]. Available: http://dx.doi.org/10.1109/GreenCom-CPSCom.2010.133

[32] V. Sridharan and D. R. Kaeli, "Quantifying software vulnerability," in *Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*, ser. WREFT '08, 2008.

[33] L. Yu, D. Li, S. Mittal, and J. S. Vetter, "Quantitatively modeling application resilience with the data vulnerability factor," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014.

[34] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "Measuring architectural vulnerability factors," *IEEE Micro*, vol. 23, no. 6, pp. 70–75, Nov. 2003.