Simulating and Detecting Radiation-Induced Errors for Onboard Machine Learning

Robert Granat, Kiri L. Wagstaff, Benjamin Bornstein, Benyang Tang, and Michael Turmon
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, Pasadena, CA 91109-8099
Email: firstname.lastname@jpl.nasa.gov

***Abstract*—Spacecraft processors and memory are subjected to high radiation doses and therefore employ radiation-hardened components. However, these components are orders of magnitude more expensive than typical desktop components, and they lag years behind in terms of speed and size. We have integrated algorithm-based fault tolerance (ABFT) methods into onboard data analysis algorithms to detect radiation-induced errors, which ultimately may permit the use of spacecraft memory that need not be fully hardened, reducing cost and increasing capability at the same time. We have also developed a lightweight software radiation simulator, BITFLIPS, that permits evaluation of error detection strategies in a controlled fashion, including the specification of the radiation rate and selective exposure of individual data structures. Using BITFLIPS, we evaluated our error detection methods when using a support vector machine to analyze data collected by the Mars Odyssey spacecraft. We observed good performance from both an existing ABFT method for matrix multiplication and a novel ABFT method for exponentiation. These techniques bring us a step closer to "rad-hard" machine learning algorithms.**

I. Introduction and Objectives

Onboard data analysis is a powerful capability now being adopted for current and future spacecraft missions. Rather than functioning as remote data collectors that simply stream information back to Earth for interpretation, spacecraft can now determine the relative priorities of different observations or generate compact summaries of large data sets, thereby maximizing the use of limited bandwidth. They can even detect and respond to short-lived events (e.g., dust devils [1]) that would otherwise be noticed only after they ended, if at all.

However, one of the major challenges to increasing the computational responsibility of a spacecraft is the radiation environment in which it operates. Bit errors in memory and altered computational results can all affect the output of an onboard analysis system, potentially resulting in the loss of data or a missed detection. Much work has gone into developing radiation-hardened processors and memory as well as software-based strategies for detecting and recovering from such errors. A common hardware technique for achieving radiation protection for SRAM is Triple-Modular Redundancy (TMR), in which three identical components perform the same memory operations and then vote on the result [2]. Software-based strategies include error detection and correction (EDAC) codes, which employ a "memory scrubber" process to run continually in the background to correct errors [3], and algorithm-specific tests to detect when an error has occurred (e.g., [4], [5]). Most of the latter has focused on general purpose computing.

In this work, we combine software-based error detection with onboard data analysis algorithms. We focus on the detection of computational errors caused by radiation-induced errors in onboard memory. Our first contribution is a software radiation simulator (BITFLIPS) that permits the specification of the radiation-induced bit error rate as well as precise control over which parts of memory are exposed. Second, we have adapted software-based error detection methods for use by support vector machines (SVMs), one of the the most widely used machine learning methods today. We also propose a new checksum-based strategy for detecting errors during exponentiation, needed by SVMs to construct nonlinear models. Finally, we tested these methods on data collected by the Mars Odyssey spacecraft.

II. Radiation Simulation: BITFLIPS

While radiation can cause errors both in spacecraft memory and in the processor, we focus on modeling and protecting against the former. The CPU is such a critical component to the entire spacecraft, not just the data analysis system, that it is likely to be radiation-hardened for the foreseeable future. However, spacecraft memory could potentially tolerate less hardening, if the software itself can detect and compensate for errors. The use of less-hardened memory components could greatly decrease the cost and increase the capability of a mission. Therefore, this seems the most realistic and profitable arena in which to advance onboard error detection. Further, even radiation-hardened memory experiences the occasional error, so the ability to detect and recover from those errors is useful even with more reliable components.

Radiation can cause a variety of errors in memory, include flipped bits, stuck bits, and damaged components. Little can be done in the latter two cases, but flipped bits (single-event upsets or SEUs) are transient effects for which recomputation can be a reasonable solution.

We designed and implemented a lightweight SEU software simulator, BITFLIPS (Basic Instrumentation Tool for Fault Localized Injection of Probabilistic SEUs), that is built on the Valgrind debugger/profiler [6]. BITFLIPS injects errors in a reproducible fashion and, for programs written in C, permits the specification of the SEU rate as well as which program

variables to expose and when. We used BITFLIPS to test the performance of our error detection algorithms at a wide range of error injection rates, using receiver operating characteristic (ROC) curves to determine the trade-offs between detection and false alarm rates at various detection thresholds.

The open source Valgrind debugging and profiling tool provides an ideal foundation for BITFLIPS. Valgrind simulates a CPU in software and provides a modular architecture for creating tools that hook into its simulation environment. Valgrind's stock tool suite contains a memory leak detector, CPU cache profiler, program caller-callee inspector, system heap profiler, and a thread synchronization debugger. BITFLIPS is patterned after Valgrind's memory leak detector, but instead of monitoring memory usage, BITFLIPS injects SEUs into memory during program execution.

BITFLIPS relies on Valgrind's on-the-fly program instrumentation capability to inject SEUs. To simplify instrumentation, Valgrind translates a program's processor specific instructions into VEX IR, a Reduced Instruction Set Computing (RISC)-like Intermediate Representation (IR) language. RISC-like instructions eliminate the need for plugin tools like BITFLIPS to contain specialized program logic tailored to complicated, possibly processor-specific instructions. Instead tools analyze and operate on basic load, store, arithmetic, comparison, and branch operations. The Valgrind simulator, and by extension, BITFLIPS, operates in an instrument-execute loop.

The instrumentation process begins when Valgrind translates the first (or next) block of a program's processor specific instructions into VEX IR. Next, Valgrind passes its VEX IR block to BITFLIPS for analysis and instrumentation. BITFLIPS then interleaves a special C-callback VEX IR instruction between each of the program's VEX IR instructions in the block. The callback instruction, when executed, results in a call to a BITFLIPS C function, `BF_doFaultCheck()`, which is responsible for deciding when and where to inject SEUs. The `BF_doFaultCheck()` function delegates to `BF_doFlipBits()` when appropriate to perform the actual SEU operation. When BITFLIPS finishes its instrumentation, the VEX IR block is passed back to Valgrind. Finally, Valgrind executes the instrumented instruction block. This process repeats until there are no more program instructions left to execute.

The rate at which BITFLIPS injects SEUs is governed by a radiation flux parameter which is fixed at the time of initial program execution. The units of this parameter are SEUs per kilobyte per second. We use kilobytes as a proxy for physical memory area; the larger the area, the more memory is exposed to radiation. The SEU density (number of bits flipped per SEU) is determined by a discrete Poisson distribution. Sixty percent of BITFLIPS' SEUs affect a single bit. Thirty percent of BITFLIPS' SEUs affect two bits, and so on. An upset affecting seven bits is exceedingly rare and accounts for only one percent of SEUs injected.

For both precise experimental control and improved reporting, BITFLIPS allows the specification of which program variables to expose to radiation. There are two requirements for this capability: 1) the program must be written in C and 2) the program source code must be accessible for compilation. Exposing (or shielding) individual variables is achieved through a Valgrind feature known as Client Request Macros (CRMs). Valgrind CRMs are C preprocessor macros whose substituted code results in a series of register bit shifts. When a program is run under Valgrind, these register operations are detected by the Valgrind simulator and mapped to C callbacks in BITFLIPS. When a program is run in its native environment (i.e., outside of the Valgrind simulator), the register operations are effectively no-ops, and they do not affect the operation of the program. Moreover, the run-time overhead imposed by CRMs on native programs is negligible: six simple integer instructions. BITFLIPS CRMs also communicate to BITFLIPS the C type (e.g., `char`, `int`, `float`, `double`, etc.) and layout (for row- or column-major matrices) of the exposed program variables. BITFLIPS uses this information, in its verbose output mode, to report variable values before and after an SEU, as well as the difference between them, so that the magnitude of the SEU's impact can be quantified.

The BITFLIPS CRMs are as follows:

- `VALGRIND_BITFLIPS_ON()` Enables SEU injection.
- `VALGRIND_BITFLIPS_OFF()` Disables SEU injection.
- `VALGRIND_BITFLIPS_MEM_ON(addr, nrows, ncols, type, order)` Exposes a block of memory beginning at address *addr* to SEUs. The memory block has *nrows* rows and *ncols* columns. The C *type* of the block (e.g. `char`, `int`, `float`, `double`, etc.) and matrix *order* (row- or column-major) give BITFLIPS additional information to use when reporting variable values before and after an SEU.
- `VALGRIND_BITFLIPS_MEM_OFF(addr)` Shields the previously exposed block of memory beginning at address *addr* from future SEUs.

III. Radiation Detection

Our approach to detection of radiation-induced errors is based around postcondition checks on numerical subroutines. If the operation was carried out successfully, certain relations between the routines inputs and its computed outputs should hold true; where they do not, an error is indicated. For example, when performing the matrix inverse operation $B = A^{-1}$, we expect $AB = I$. Due to the limitations of finite-precision arithmetic, most often postconditions will not hold true exactly; consequently we test whether they are true within some error bound.

In general, it is desirable for such postcondition checks to consume considerably less computational resources than the original computation. Otherwise, it would be more direct and informative to simply repeat the computation and compare the results. One way to avoid this sort of exhaustive check is to employ a probe vector w. Consider a linear operation with factorable inputs and outputs:

$$L_1 L_2 \cdots L_p \stackrel{?}{=} R_1 R_2 \cdots R_q. \tag{1}$$

Since an error in one element will often fan out across the result matrix as the computation progresses, we can use w to compute checksum vectors that are compared instead:

$$L_1 L_2 \cdots L_p w \stackrel{?}{=} R_1 R_2 \cdots R_q w. \quad (2)$$

This method, known as result-checking (RC), was used by Freivalds [7] to check multiplication, and was analyzed in a general context by Blum and Kannan [8]. This idea is also the basis of the checksum augmentation approach of Huang and Abraham [4] under the name algorithm-based fault tolerance (ABFT) (for a comparison of RC and ABFT, see [9]). Both approaches have since been extended by a number of authors to various linear decompositions [10]–[12], the FFT [13], [14], and other numerical operations. Boley et. al. [15], [16] explored fault location and correction using this method, as well as the use of multiple probe vectors. The effects of multiple faults, including those that occur during the postcondition test itself, have been explored through experiment [17]. Previous work has also explored the setting of error bounds for checksum tests [5], [11], [18].

In this work, we have applied this sort of algorithm-based fault tolerance approach to support vector machines (SVMs), one of the most widely used machine learning methods today. SVMs are currently in use onboard the EO-1 (Earth Observing 1) spacecraft to perform pixel-level classification of hyperspectral images [19] and can also be used to perform regression, such as estimating the dust and water ice content of the Martian atmosphere [20].

A. Support Vector Machines

Support vector machines [21] infer a hyperplane to separate labeled training data into two distinct classes. The hyperplane can then be used to classify new items. Arbitrarily complex decision boundaries (not just linear ones) can be created by mapping the input data via a kernel function into a higher-dimensional space in which the hyperplane is constructed. SVMs have also been extended to apply to regression problems [22], in which the goal is to estimate a real-valued quantity rather than assigning a discrete class to a new item.

Given a data set of n items $X = \{x_1, \ldots, x_n\}$, where each $x_i \in \mathcal{R}^d$ is a d-dimensional feature vector, and a vector y such that $y_i \in \{+1, -1\}$ is the label for x_i, an SVM is defined by $n+1$ parameters: a weight α_i for each x_i and a bias term, b. Each x_i with a non-zero weight α_i is termed a support vector, and it is only these items that influence the classification of new data. New items are classified as follows:

$$f(x) = \text{sign}(\sum_{i=1}^{n} \alpha_i y_i (x \cdot x_i) + b). \quad (3)$$

Let s be the number of support vectors, which we will refer to as z_j instead of x_i, obtaining:

$$f(x) = \text{sign}(\sum_{j=1}^{s} \alpha_j y_j (x \cdot z_j) + b), \quad (4)$$

If the two classes are not linearly separable, the dot product $(x \cdot z_j)$ can be replaced by a kernel function $\mathcal{K}(x, z_j)$, which is equivalent to using some mapping $\phi(x)$ to transform each x (and z) into a feature space with more (possibly infinite) dimensions and computing the dot products there. After adding the kernel function, the SVM decision function becomes:

$$f(x) = \text{sign}(\sum_{j=1}^{s} \alpha_j y_j \mathcal{K}(x, z_j) + b). \quad (5)$$

Common choices for kernel functions are polynomials and Gaussian radial basis functions. The Gaussian kernel is defined as

$$\mathcal{K}_G(x, z_j) = e^{-\frac{1}{\gamma}\|x - z_j\|^2}, \quad (6)$$

where $\gamma = 2\sigma^2$ and σ is the width of the kernel, or the standard deviation of values.

To train the SVM, we must compute values for α and b, which are usually obtained by solving the following quadratic programming problem:

$$\begin{array}{rl} \text{minimize:} & \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \mathcal{K}(x_i, x_j) - \sum_i \alpha_i \\ \text{subject to:} & 0 \le \alpha_i \le C, \sum_i \alpha_i y_i = 0, \end{array}$$

where C is a regularization parameter.

A similar derivation is obtained when using SVMs for regression, with the addition of a tolerance parameter ε, which specifies how tightly the learned model's predictions must fit to the true y labels in the training data. In addition, each support vector z_j has two Lagrange multipliers, α_j and α_j^*.

$$f(x) = \sum_{j=1}^{s} (\alpha_j - \alpha_j^*) \mathcal{K}(x, z_j) + b. \quad (7)$$

B. Algorithm-Based Fault Tolerance

To create an error-detecting SVM, we focused on the subroutines in which most of the computational time is spent. The classification of new data requires the calculation of the kernel values and then the computation of $f(x)$ using Equation 5 (classification) or 7 (regression). To classify m new items, the kernel computations require a matrix-matrix multiplication, and $f(x_i) \forall x_i$ amounts to a matrix-vector multiplication. Therefore, the kernel computation dominates the computational effort and that is where we invest our efforts in adding error detection abilities.

The two subroutines needed to calculate the kernel values are matrix multiplication and exponentiation (for Gaussian kernels). Each subroutine has a testable postcondition and is thus amenable to the ABFT approach.

1) Matrix Multiplication: Given a linear kernel, all of the kernel values can be computed via matrix multiplication:

$$K = XZ \quad (8)$$

where K is the kernel matrix, $X \in \mathcal{R}^{m \times d}$ is the matrix of data to be classified, and $Z \in \mathcal{R}^{d \times s}$ is the matrix of support vectors ($s \le n$). Note that if X is the training data set, then $m = n$, but more generally m can be any size for a new data set. We replace all calls to $\mathcal{K}(x_i, z_j)$ with $K_{i,j}$. Since the SVM relies on this matrix being accurately computed, any

errors that occur in the creation of K may result in errors in the SVM output (classification or regression).

Our goal is to determine whether or not an error occurred during the computation of K from X and Z. We need to apply a test to determine whether $K = XZ$ but more cheaply than doing a complete recomputation of the matrix multiplication. Therefore, let $\hat{w} = Zw$ for some arbitrary vector $w \in \mathcal{R}^{s\times 1}$ (in our tests, we used a w vector of all ones). Then by substitution, $Kw = XZw = X\hat{w}$. To determine whether K is correct, we simply compare $X\hat{w}$ to Kw, checking m values rather than all $m \times s$ values in K. We define the *relative error size* ϵ as the maximum difference between these values:

$$\epsilon = \frac{1}{C}\|Kw - X\hat{w}\|_\infty \tag{9}$$

subject to a normalization factor $C = \|w\|_\infty\|X\|_\infty\|Z\|_\infty$ that compensates for potential large variations in the values of the input matrices. $X\hat{w}$ is computed prior to the matrix multiplication, and Kw is computed afterwards. If ϵ exceeds a pre-specified tolerance, then an error is flagged. This process requires two matrix-vector multiplications, which is cheaper than recomputing all mn entries of K to use a voting or consensus strategy to detect errors.

2) Exponentiation (Gaussian Kernel): Computing the kernel matrix via matrix multiplication is sufficient for linear kernels. However, for Gaussian kernels, an additional operation is needed. First, we compute the linear kernel $K^{\text{lin}} = XZ$ as above. Then we update the kernel values as follows:

$$K_{ij}^{\text{rbf}} = e^{-\frac{1}{\gamma}\|x_i\|^2 - 2K_{ij}^{\text{lin}} + \|z_j\|^2} \tag{10}$$

We are concerned with whether an error occurs during the computation of the exponential value. To do this, we utilize a postcondition that compares the exponentiation of the sum of input values to the product of their exponentiations. Let T be a matrix with elements $t_{ij} = -\frac{1}{\gamma}\|x_i\|^2 - 2K_{ij}^{\text{lin}} + \|z_j\|^2$, so $K_{ij}^{\text{rbf}} = e^{t_{ij}}$. Then the checksum we calculate for column j of the kernel matrix before performing the individual exponentiations is

$$c_j = e^{\sum_i t_{ij}}. \tag{11}$$

We then compute $K_{ij} = e^{t_{ij}}$ as usual. Afterwards, we compute the second checksum:

$$\hat{c}_j = \prod_i K_{ij}^{\text{rbf}}. \tag{12}$$

If no error has occurred, then $c_j = \hat{c}_j, \forall j$. Our error is then the maximum relative error between the checksum elements:

$$\epsilon = \max_j \frac{|c_j - \hat{c}_j|}{\min(c_j, \hat{c}_j)}. \tag{13}$$

As with the matrix multiplication, if ϵ exceeds a pre-specified tolerance, then an error is flagged.

Where m is large there is significant potential for numerical underflow for this procedure. Underflow can be addressed in a computationally efficient way by augmenting the checksum vectors with buffer vectors b and $\hat{b}$. Terms that would cause underflow are added to (multiplied with) the buffer vector instead, so that for sets $K_c \subset \{1,\ldots,m\}$ and $K_b = \{1,\ldots,m\}\backslash K_c$,

$$c_j = e^{\sum_{k\in K_c} t_{kj}}, \tag{14}$$

$$b_j = e^{\sum_{k\in K_b} t_{kj}}, \tag{15}$$

$$\hat{c}_j = \prod_{k\in K_c} K_{kj}^{\text{rbf}}, \tag{16}$$

$$\hat{b}_j = \prod_{h\in K_b} K_{hj}^{\text{rbf}}. \tag{17}$$

The error is then

$$\epsilon = \max_j \frac{|c_j/\hat{b}_j - \hat{c}_j/b_j|}{\min(c_j/\hat{b}_j, \hat{c}_j/b_j)}. \tag{18}$$

This approach will eliminate underflow problems as long as $2\log\mu < \sum_i t_{ij}$, where μ is the smallest machine representable floating point number. In truly problematic cases, where this condition does not hold, we are forced to take a more computationally expensive approach to eliminating underflow. Here we replace Equations 11 and 12 with

$$c_j = e^{\frac{1}{m}\sum_i t_{ij}} \tag{19}$$

and

$$\hat{c}_j = \prod_i (K_{ij}^{\text{rbf}})^{1/m}. \tag{20}$$

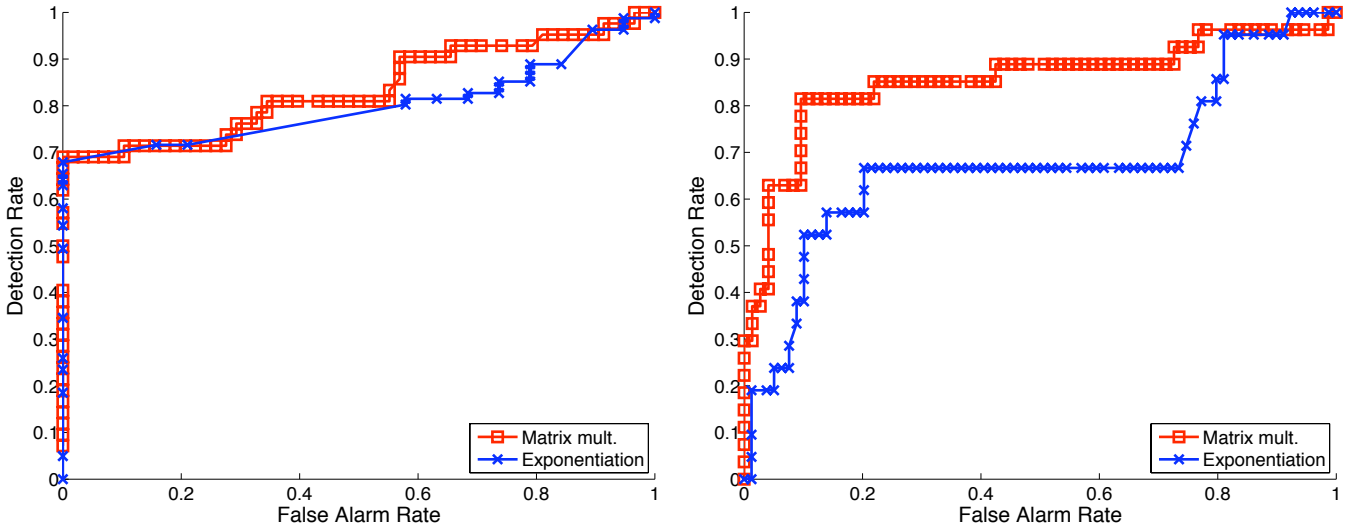
Normalizing the sum by m, the number of items being analyzed by the SVM, guarantees that the calculation will avoid underflow as long as no individual kernel element induces underflow (i.e., $e^{\min_{i,j}(t_{ij})} > \mu$). However, it does so at the cost of a large number of computationally expensive power operations.

IV. Experimental Results

A. Data Sets

We evaluated the ABFT error detection methods on both classification and regression tasks. The classification task comes from the "letter" classification data set provided by the UCI machine learning repository [23]. We used data for the letters A and B to generate a binary classification problem. Each letter was originally recorded as a rectangular matrix of black and white pixels, then converted into 16 numerical attributes that capture statistical information about the shape of the letter. We trained models on 100 randomly selected items from the full data set and tested on multiple disjoint sets of 100 randomly selected items. For this data set, we used a Gaussian kernel ($\gamma = 0.05$) and a regularization factor C value of 0.8. These hyperparameters were selected after a cross-validation search on held-out data.

The regression task uses spacecraft data and has been previously identified as a useful onboard data analysis problem in a high-radiation environment (Mars orbit). The data comes from the THEMIS instrument on the Mars Odyssey spacecraft. THEMIS is the Thermal EMission Imaging System, a camera that records observations at visible (VIS) and infrared (IR)



(a) Letter Classification (SEU rate: 5×10^{-9})

(b) Water Ice Regression (SEU rate: 5×10^{-8} (MM) and 5×10^{-9} (exp))

Fig. 1. Error detection results for both classification and regression tasks (100 trials).

wavelengths [24]. We previously developed onboard algorithms for analyzing the IR data to detect thermal anomalies, track the position of the polar cap edges, and estimate aerosol (dust or water ice) content in the atmosphere [20]. That evaluation assumed error-free computation. Here, we used the water ice opacity estimation task, which relies on an SVM, as the computation experiencing radiation.

The IR data consists of 8 distinct wavelength bands, with a spatial resolution of 100 meters per pixel. Each image is 320 pixels (32 km) wide and a variable number (3600 to 14352) of pixels long, divided into 256-line "framelets". Each item in the data set represents a single framelet; the feature values are the average pixel value for each wavelength, across the framelet, and the label for item is the water ice content (opacity) of the atmosphere observed in that pixel. The full data set contains 223,690 items. For each SVM we trained on this data, we used a Gaussian kernel ($\gamma = 0.1$) with a C value of 50. Since this is a regression problem, we must also specify the maximum error tolerance for the training process ($\varepsilon = 0.01$).

B. Methodology

One aspect of this work that distinguishes it from previous work is the incorporation of ABFT checksums to detect errors caused by SEUs in memory, rather than processor faults. We used the BITFLIPS radiation simulator to track all data structures and inject SEUs at a specified rate. We selected SEU injection rates that resulted in a substantial number of errors, but also several error-free runs, so that we could evaluate both detection and false alarm rates.

For both classification and regression, we conducted several trials and measured error detection performance in terms of detection rate

$$D = \frac{TP}{TP + FN} \tag{21}$$

and false alarm rate

$$F = \frac{FP}{FP + TN}, \tag{22}$$

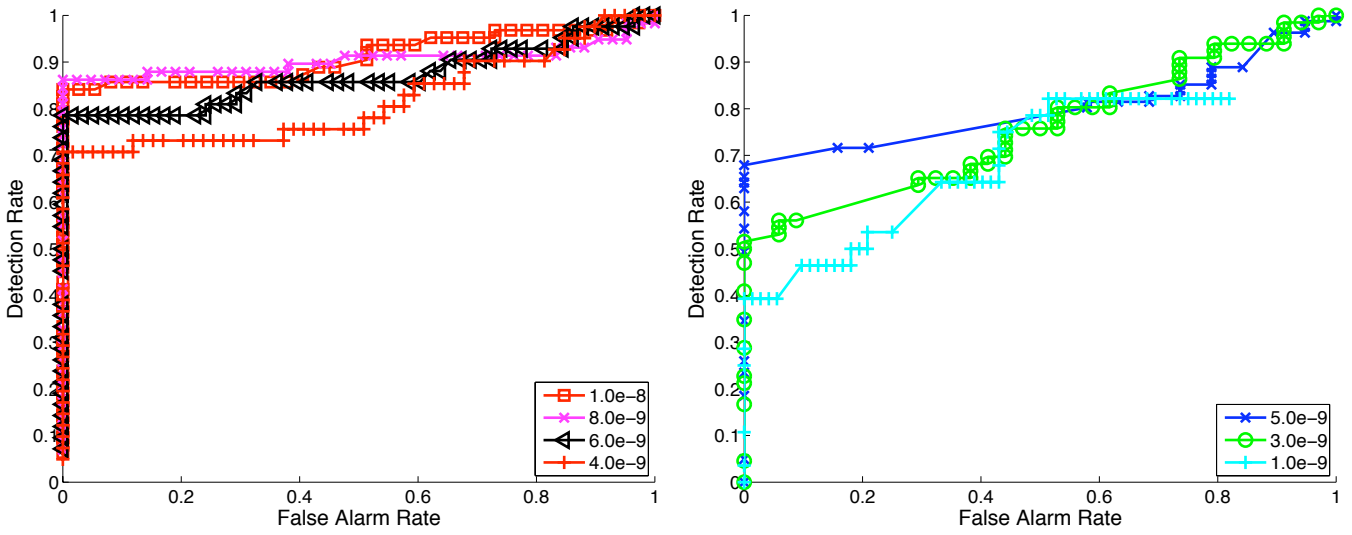
where TP is the number of true positives, FP is the false positives, TN is the true negatives, and FN is the false negatives. We varied the detection threshold to which ϵ is compared, to determine whether an error occurred, and obtained a range of performance values.

We focused on detecting end-result errors in the output of the SVM, rather than detecting each time an SEU occurred. That is, to determine whether an error had occurred (and therefore should have been detected), we compared the output of the SVM that was obtained when running without SEUs injected to that obtained when SEUs were injected. If they were the same, no error occurred. Note that there may still have been SEUs happening, but the SVM's natural tolerance for a low level of radiation prevented an error from occurring in the output. If detecting an error triggers rollback or recomputation, it is appropriate that this should only be done if the SEUs had an impact on the analysis result.

For both classification and regression, we first generated 100 distinct SVMs, each trained on a different subset of 100 items. We then tested each model on a disjoint set of 100 test items, running it multiple times with and without SEUs injected. When SEUs were being injected, we exposed the X, Z, and K matrices.

C. Results

Figure 1 shows the results for both classification and regression. The radiation rate used to evaluate error detection on the classification task was 5.0×10^{-9} SEUs per kB per second. For comparison, commercial SRAM in low-Earth orbit experiences about 1.2×10^{-7} SEUs per kB per second; radiation-hardened SRAM experiences up to 1.2×10^{-12}. Therefore, this rate not



(a) Matrix multiplication

(b) Exponentiation

Fig. 2. Error detection results with different specified error rates, for the letter classification task (100 trials).

only provides a good mixture of erroneous and error-free runs, but it also represents a realistic test environment. At this rate, we observed that 42 of 100 matrix multiplication trials, and 81 of 100 exponentiation trials, produced erroneous output. Note that all trials experienced SEUs, but not all of the SEUs caused the final output to contain errors.

For SVM classification, we observed similar performance for the matrix multiplication and exponentiation subroutines at low false alarm rates (e.g., achieving 70% detection with no false alarms). However, the matrix multiplication subroutine achieved 90% detection with a 58% false alarm rate, while the exponentiation subroutine had a 90% false alarm rate at that detection level.

Error detection during SVM regression provided similar results for the matrix multiplication subroutine, but lower performance for the exponentiation subroutine (Figure 1b). Here, the SEU rate used for exponentiation was again 5.0×10^{-9} SEUs per kB per second, but a higher rate was needed to obtain a sufficient number of erroneous trials for the matrix multiplication step (5.0×10^{-8}). The higher SEU rate was needed because this data set has only 8 features, compared with 16 features for the letter classification data set, so the X and Z matrices being multiplied were each only half as large. Given these rates, we observed 27 of 100 and 21 of 100 trials with erroneous output for the matrix multiplication and exponentiation subroutines, respectively.

Figure 2 shows detection results obtained when different SEU rates were specified for letter classification. For both matrix multiplication and exponentiation, we generally found better detection results in the presence of *more* radiation as compared with lower rates. This was especially pronounced at lower false alarm rates. For example, when performing matrix multiplication with an SEU rate of 1.0×10^{-8}, 85% of errors were detected with no false alarms, and 90% detection was achieved with only 53% false alarms. This is likely because although we seek only to detect whether at least one error occurred, at higher SEU rates multiple errors could strike, increasing the likelihood that at least one causes the error size ϵ to exceed the detection threshold. Increased detection at higher radiation rates is a useful attribute for onboard analysis systems.

The careful reader may notice that the rates tested with the exponentiation subroutine do not go as high as those used for matrix multiplication. The reason for this is that the exponentiation takes longer to compute, and therefore at the same rate experiences a larger number of SEUs. For rates greater than 5.0×10^{-9}, all 100 of the trials experienced errors and therefore did not provide the necessary mixture of erroneous and error-free trials to generate an ROC curve.

V. Conclusions and Future Directions

In this paper, we have described how algorithm-based fault tolerance (ABFT) methods can used in the context of onboard data analysis methods to increase their robustness in the presence of radiation-induced errors. We focused on errors that occur in memory (single-event upsets or SEUs) and later affect the analysis results provided by support vector machines (SVMs), a technology now being adopted for use onboard spacecraft both for classification and regression.

We also described BITFLIPS, a lightweight software radiation simulator that we developed. It provides precise control over both the simulated SEU injection rate as well as which elements of memory are exposed to the radiation. We believe this work to be the first effort to combine ABFT methods with SVMs and the first to test their combined capabilities in such a simulator.

The techniques we proposed for detecting errors caused by SEUs rely on the computation of checksum values before and after executing a critical calculation, such as matrix

multiplication or exponentiation for a Gaussian (RBF) kernel. Errors are detected when the difference between the expected and actual checksums differ by more than a threshold amount.

Our results indicate that onboard data analysis methods can successfully detect radiation-induced errors that strike during the critical matrix multiplication and exponentiation steps needed to compute the kernel matrix for a support vector machine. Interestingly, detection improved at higher SEU injection rates, as compared to lower rates.

Given the ability to detect errors, a clear next step is to add a rollback-and-recompute capability. While these experiments sought only to test the ability to determine whether any error had occurred during computation, it would be even more useful to identify which of the output values required recalculation. For matrix multiplication, the checksum is computed as the max (infinity norm) over a vector of m values, one per item being classified; $(Kw - X\hat{w}) \in \mathcal{R}^{m \times 1}$. Identifying which of these m items violates the error threshold provides guidance in re-running the matrix multiplication, this time with only a subset of X rather than the full matrix. This is much more efficient than blindly recomputing $K = XZ$ whenever any error is detected. Likewise, the checksums c_j and $\hat{c}_j$ that are used to compute the error size ϵ for exponentiation provide a result for each input item x_j, so they could be used to selectively decide which K_{ij} values must be recomputed via Equation 10.

Ultimately, we aim to provide error-detecting and correcting methods as a robust alternative to current onboard machine learning and data analysis efforts.

ACKNOWLEDGMENTS

This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

[1] A. Castano, A. Fukunaga, J. Biesiadecki, L. Neakrase, P. Whelley, R. Greeley, M. Lemmon, R. Castano, and S. Chien, "Autonomous detection of dust devils and clouds on Mars," in *Proceedings of the IEEE International Conference on Image Processing*, 2006, pp. 2765–2768.

[2] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal*, pp. 200–209, 1962.

[3] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, "Software-implemented EDAC protection against SEUs," *IEEE Transactions on Reliability*, vol. 49, no. 3, pp. 273–284, 2000.

[4] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 33, no. 6, pp. 518–528, 1984.

[5] M. Turmon, R. Granat, D. Katz, and J. Lou, "Tests and tolerances for high-performance software-implemented fault detection," *IEEE Transactions on Computers*, vol. 52, no. 5, pp. 579–591, May 2003.

[6] N. Netercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007, pp. 89–100.

[7] R. Freivalds, "Fast probabilistic algorithms," in *Proc. 8th Symp. Mathemat. Foundat. Comput. Sci.*, 1979, pp. 57–69, also in *Lecture Notes in Computer Science*, vol. 74, Springer.

[8] M. Blum and S. Kannan, "Designing programs that check their work," in *Proc. 21st Symp. Theor. Comput.*, 1989, pp. 86–97.

[9] P. Prata and J. G. Silva, "Algorithm-based fault tolerance versus result-checking for matrix computations," in *Proc. FTCS-29*, 1999, pp. 4–11.

[10] J.-Y. Jou and J. A. Abraham, "Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures," *Proc. IEEE*, vol. 74, no. 5, pp. 732–741, 1986.

[11] F. T. Luk and H. Park, "An analysis of algorithm-based fault tolerance techniques," *J. Parallel and Dist. Comput.*, vol. 5, pp. 172–184, 1988.

[12] M. P. Connolly and P. Fitzpatrick, "Fault-tolerant QRD recursive least squares," *IEE Proc. Comput. Digit. Tech.*, vol. 143, no. 2, pp. 137–144, 1996.

[13] Y.-H. Choi and M. Malek, "A fault-tolerant FFT processor," *IEEE Trans. Comput.*, vol. 37, no. 5, pp. 617–621, 1988.

[14] S. J. Wang and N. K. Jha, "Algorithm-based fault tolerance for FFT networks," *IEEE Trans. Comput.*, vol. 43, no. 7, pp. 849–854, 1994.

[15] D. L. Boley, R. P. Brent, G. H. Golub, and F. T. Luk, "Algorithmic fault tolerance using the Lanczos method," *SIAM J. Matrix Anal. Appl.*, vol. 13, no. 1, pp. 312–332, 1992.

[16] D. L. Boley and F. T. Luk, "A well-conditioned checksum scheme for algorithmic fault tolerance," *Integration, The VLSI Journal*, vol. 12, pp. 21–32, 1991.

[17] J. G. Silva, P. Prata, M. Rela, and H. Madeira, "Practical issues in the use of ABFT and a new failure model," in *Proc. FTCS-28*, 1998, pp. 26–35.

[18] D. Boley, G. H. Golub, S. Makar, N. Saxena, and E. J. McCluskey, "Floating point fault tolerance with backward error assertions," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 302–311, 1995.

[19] R. Castano, D. Mazzoni, N. Tang, T. Doggett, S. Chien, R. Greeley, B. Cichy, and A. Davies, "Learning classifiers for science event detection in remote sensing imagery," in *Proceedings of the 8th International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, 2005.

[20] R. Castano, K. L. Wagstaff, S. Chien, T. M. Stough, and B. Tang, "On-board analysis of uncalibrated data for a spacecraft at Mars," in *Proceedings of the Thirteenth International Conference on Knowledge Discovery and Data Mining*, 2007, pp. 922–930.

[21] C. Cortes and V. Vapnik, "Support-vector network," *Machine Learning*, vol. 20, pp. 273–297, 1995.

[22] H. Drucker, C. J. Burges, L. Kaufman, A. Smola, and V. Vapnik, "Support vector regression machines," in *Advances in Neural Information Processing Systems 9*. MIT Press, 1997, pp. 155–161.

[23] A. Asuncion and D. Newman, "UCI machine learning repository," http://www.ics.uci.edu/ mlearn/MLRepository.html, 2007.

[24] P. R. Christensen *et al.*, "Morphology and composition of the surface of Mars: Mars Odyssey THEMIS results," *Science*, vol. 300, pp. 2056–2061, June 2003.