International Journal of Parallel Programming (2021) 49:410–439
https://doi.org/10.1007/s10766-021-00707-0

Predicting the Soft Error Vulnerability of Parallel Applications Using Machine Learning

Işıl Öz[1] · Sanem Arslan[2]

Received: 11 August 2020 / Accepted: 12 March 2021 / Published online: 28 March 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

Abstract
With the widespread use of the multicore systems having smaller transistor sizes, soft errors become an important issue for parallel program execution. Fault injection is a prevalent method to quantify the soft error rates of the applications. However, it is very time consuming to perform detailed fault injection experiments. Therefore, prediction-based techniques have been proposed to evaluate the soft error vulnerability in a faster way. In this work, we present a soft error vulnerability prediction approach for parallel applications using machine learning algorithms. We define a set of features including thread communication, data sharing, parallel programming, and performance characteristics; and train our models based on three ML algorithms. This study uses the parallel programming features, as well as the combination of all features for the first time in vulnerability prediction of parallel programs. We propose two models for the soft error vulnerability prediction: (1) A regression model with rigorous feature selection analysis that estimates correct execution rates, (2) A novel classification model that predicts the vulnerability level of the target programs. We get maximum prediction accuracy rate of 73.2% for the regression-based model, and achieve 89% F-score for our classification model.

Keywords Soft error analysis · Fault injection · Parallel programming · Machine Learning

✉ Işıl Öz
isiloz@iyte.edu.tr

Sanem Arslan
sanem.arslan@marmara.edu.tr

[1] Computer Engineering Department, Izmir Institute of Technology, Izmir, Turkey

[2] Computer Engineering Department, Marmara University, Istanbul, Turkey

1 Introduction

With the larger availability of multicore architectures and their high potential on performance improvement, the number of parallel programs utilizing the parallel execution units on those architectures has been increasing. While the parallel programs reduce the execution times substantially, they tend to be more error prone due to their complexity in addition to soft error vulnerability of host parallel systems. The soft error rates, which result from bit-flips in hardware components due to environmental factors, keep increasing with design choices in parallel systems like smaller transistor sizes and more aggressive power modes [35–37]. Therefore, the reliability becomes an important criteria in computer architecture.

Since the soft error vulnerability of modern computer systems tends to increase, fault tolerance techniques providing higher reliability are becoming more crucial. These hardware or software redundancy based techniques induce extra cost in the system in terms of money or performance. Due to limited amount of resources and budget, it may not make sense to protect programs that are not highly vulnerable to soft errors. By obtaining the soft error vulnerability of the programs, we can make decisions whether we need fault tolerance techniques, or how much redundancy we need to have in our target system.

The prevalent method to analyze the program soft error vulnerability is fault injection [15, 41]. This method deliberately introduces faults in the hardware structures during the program execution, then examines the program outcomes to understand the effect of the injected fault. By repeating this injection for several times (enough to be statistically significant [19]), silent data corruption (SDC) rates have been obtained and used as a soft error vulnerability metric. By utilizing SDC rates acquired by fault injection experiments, we can decide possible fault tolerance methods for our target execution. Specifically, we may choose applying explicit redundancy techniques for the programs with higher SDC rates, while we rely on available protection methods provided as part of the system for the execution of the programs with low SDC rates.

Although the fault injection yields useful results, the large number of experiments may become impractical, especially for the large-scale long-running applications. For instance, if we have a program that takes 1-h to be completed, and we need to repeat fault injection experiments for 1000 times (decided by statistical analysis); we should spend roughly 40 days for our experiments. Since the fault injection also requires profiling and tracing the application during its execution, the required time to obtain SDC rates would be impractical. Moreover, profiling the parallel applications with several threads requires more effort/time to track all execution contexts.

Since the fault injection is too time-consuming for soft error vulnerability evaluation of programs, prediction-based approaches have been proposed recently [14, 18, 21–23, 29]. The proposed methods build a machine learning model based on program/system characteristics, and use the model to predict soft error proneness of the programs without conducting fault injection experiments. Although they achieve comparable accuracy rates in prediction of program

vulnerability, they either require a detailed fault propagation analysis among the program instructions [18, 21, 22], or only focus on performance-related program characteristics [14, 29].

In this work, we present a soft error vulnerability prediction study for parallel applications using machine learning. Our work aims to eliminate long fault injection times, and predict the vulnerability of programs by using application characteristics gathered from program profile. With our approach, parallel program developers can understand which applications should be focused and heavily protected against soft errors without conducting a fault injection study. While the existing studies evaluate ML-based prediction mechanisms for reliability prediction, they do not consider parallel program characteristics as model features. In this study, for the first time, we consider parallel programming features within the thread communication, data sharing, and performance features. Additionally, we propose a novel classification model to predict the vulnerability level of the applications. To the best of our knowledge, such an approach is not applied before. Previous studies use either classification models for classifying individual instructions or regression models to predict SDC rate of the applications. In our classification model, we classify applications as vulnerable and not (less) vulnerable ones by putting an SDC threshold. Our main contributions are as follows:

- We present ML-based soft error vulnerability prediction models for parallel applications. Our approach uses parallel program features, for the first time, in addition to traditional performance characteristics.
- We build a regression model with a detailed feature analysis to estimate correct execution rates, and a novel classification model to assign a vulnerability level for parallel programs.
- We collect both fault injection and profiling data for 30 parallel programs based on *pthread* and *OpenMP* programming models, and utilize three ML algorithms for our evaluation.
- While the prediction accuracy of our regression model reaches to 73.2% for random forest regression with feature selection, our gradient boosting-based classification model achieves 89% F-score for the balanced dataset.

The remainder of this paper is organized as follows: Sect. 2 explains our fault model and presents some background on soft error vulnerability evaluation. We explain our prediction models in Sect. 3. Then the experimental results are outlined in Sect. 4. Section 5 presents the related work on vulnerability prediction methods. Finally, in Sect. 6, we summarize the work with some conclusive remarks.

2 Soft Error Vulnerability

In this section, we present our fault model, and terminology related to soft error vulnerability.

2.1 Fault Model

In this paper, we consider soft errors [26], i.e., transient hardware faults flipping the bits in register file. We do not consider the faults in memory or caches assuming that they are protected by ECC. We focus on single-bit errors since they are much more common.

2.2 Soft Error Vulnerability Evaluation

The most common way to characterize the soft error vulnerability of a program is to perform fault injection experiments. We classify the possible outcomes of a fault injection case as follows: **(1) Correct Execution:** The program terminates successfully, and generates the expected output. **(2) SDC:** The program terminates, but its output differs from the expected output. **(3) Hang:** The program continues its execution for a time higher than the threshold. **(4) Crash:** The program execution terminates with an error code. We measure the rate of each condition as collected from fault injection experiments, and use SDC and correct execution rates as part of our evaluation.

3 Methodology

3.1 Overview of Our Approach

Figure 1 presents the overview of our approach by demonstrating the machine learning model flow. We assume that our target parallel programs have been implemented by *pthread* or *OpenMP* programming model. First, as the input variables, we extract program characteristics by profiling the program (We explain the details in Sect. 3.2). Additionally, we perform fault injection experiments and extract the SDC, crash, hang and the correct execution rates, which are utilized in evaluation of our models. As the response variable, we use SDC and correct execution rates collected from fault injection experiments. Then we build our

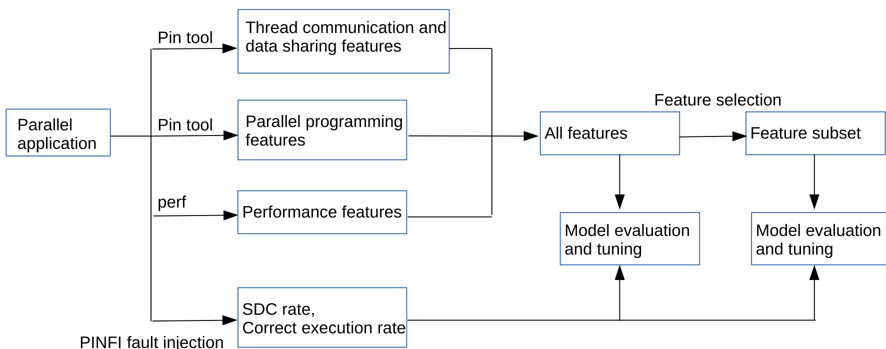

Fig. 1 Flow of our machine learning model

models based on gathered program profiling and fault injection data. We apply hyperparameter tuning in order to get the best parameters for the specific algorithms. As part of our regression-based model, we also perform a feature selection analysis to get the most influential features by eliminating redundant ones, and train our models by using the subset of features. We explain the details of our prediction model in Sect. 3.4.

3.2 Application Characterization

We analyze the characteristics that potentially affect the performance and the reliability of a parallel application. Those characteristics can be listed in four groups: *Thread Communication Characteristics*, *Data Sharing Characteristics*, *Parallel Programming Characteristics*, and *Performance Characteristics*. An application should be profiled in advance to obtain these characteristics. Then, the relation between them and the reliability behavior of the application can be analyzed in more detail. Our thread communication characteristics are similar to [11], and data sharing characteristics are similar to [9]. As opposed to [11] and [9], we use these features neither for performance prediction nor creating new synthetic parallel programs. We utilize them for our vulnerability prediction models. We formalize parallel programming characteristics as a new group which contains low-level sub-characteristics. The performance characteristics are very commonly studied microarchitecture dependent characteristics that have an impact on the application performance. All of the characteristics are described in more detail in the following subsections. Table 1 presents the characteristics we analyze as part of our proposed model.

3.2.1 Thread Communication Characteristics

Thread communication characteristics are listed in four subgroups: *communication amount*, *communication ratio*, *communication heterogeneity*, and *communication balance* similar to [11]. These sub-characteristics describe the volume and the structure of the communication events that may affect the propagation of a localized error to other threads in a shared memory model. The communication events are described in cache line granularity in this section. If the same cache line is accessed by two different threads, then a communication event is recorded in our system.

First of all, we profile the input application and extract thread communication matrix that shows communication events among the parallel executing threads. A visualization of a communication matrix for FFT application with 9 threads is shown in Fig. 2. In this figure, the labels indicate thread IDs and the darker cells refer to more communication events. The sub-characteristics are described formally based on the communication matrix of the application.

Communication amount and communication ratio metrics represent the volume of the communication among the application threads. Communication amount, represented as A_{Comm}, describes the average number of communication events per thread with the following equation [11]:

Table 1 Application characteristics

Characteristics	Sub-characteristics
Thread communication characteristics	Communication amount
	Communication ratio
	Communication heterogeneity
	Communication balance
Data sharing characteristics	Private
	Read-only
	Producer-consumer
	Migratory
Parallel programming characteristics	# Of pthread_create() functions (i.e., # of threads)
	# Of mutex variables
	# Of pthread_cond_wait() functions
	# Of pthread_cond_broadcast() functions
	# Of pthread_mutex_lock() - pthread_mutex_unlock function pairs
	# Of pthread_mutex_lock() - pthread_cond_wait() function pairs
	Average # of instructions between pthread_mutex_lock() - pthread_mutex_unlock function pairs
	Average # of instructions between pthread_mutex_lock() - pthread_cond_wait() function pairs
	% Of instructions inside a thread function relative to the total #of instructions
	% Of stack pointer accesses inside a thread function relative to the total stack pointer accesses in the program
Performance characteristics	# Of instructions
	% Of branch instructions
	% Of load instructions
	% Of store instructions
	% Of context switches
	# Of TLB misses
	Cache miss rate

$$A_{Comm} = \frac{\sum_{i=1}^{T} \sum_{j=1}^{T} M[i][j]}{T^2} \tag{1}$$

where T represents the total number of threads in an application and $M[i][j]$ represents the number of communication events between thread i and thread j. Communication ratio, represented as R_{Comm}, describes the ratio of communication events relative to the total number of memory accesses of a thread with the following equation [11]:

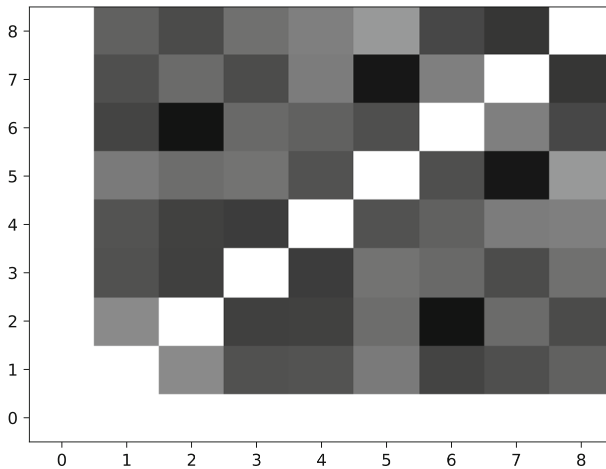

Fig. 2 Visualization of communication matrix for FFT application (darker cells refer to more communication)

$$R_{Comm} = \frac{A_{Comm}}{\sum_{i=1}^{T} AccV[i]} \tag{2}$$

where $AccV[i]$ is the total number of memory accesses belonging to the thread i. The amount metric itself may not be adequate to characterize the volume of communication; therefore, the ratio of the number of memory accesses for communication relative to its total number of memory accesses is considered in this metric. Communication heterogeneity, represented as H_{Comm}, describes the variation of the communication among thread pairs in an application with the following equation [11]:

$$\begin{aligned} M_{norm} &= \frac{M}{\max(M)} \times 100, \\ H_{Comm} &= \frac{\sum_{i=1}^{T} \mathrm{Var}(M_{norm}[i][1...T]))}{T} \end{aligned} \tag{3}$$

where M is a matrix that shows communication events among each thread pairs. In order to calculate the maximum and the variance values, the *max* and the *Var* functions are utilized. To calculate H_{Comm}, the communication matrix M is normalized firstly, and then the average variance of the number of communication events per thread is calculated. A higher variation of the communication events indicates that a group of threads communicates more in the group than communicating with the others. This information might be important in the sense that an error may propagate faster in a group of threads. Communication balance, represented as B_{Comm}, describes the balance among the thread communication events with the following equation [11]:

$$CommV[i] = \sum_{j=1}^{T} M[i][j],$$
$$B_{Comm} = \left(\frac{\max(CommV)}{\sum_{i=1}^{T} CommV[i]/T} - 1 \right) \times 100\% \qquad (4)$$

where $CommV[i]$ is a communication vector for the thread i. To calculate B_{Comm}, the communication vector for each thread is calculated firstly, and then the classic formulation for load balance is utilized [32]. The higher value of B_{Comm} indicates that there is a highly imbalanced communication behavior among the threads, on the contrary, if it is close to 0 a highly balanced communication behavior is occurred among the threads.

Figure 3 presents the normalized amount, ratio, heterogeneity, and balance values for our target programs. There is a fairly balanced communication among all threads for $water-nsquared$ with the highest amount value. On the other hand, for $susan.2$, which has a strong communication with a few thread groups, it is low and highly imbalanced. The communication behaviour is heterogeneous for most of the applications.

3.2.2 Data Sharing Characteristics

Data sharing characteristics are listed in four subgroups: *private*, *read-only*, *producer-consumer*, and *migratory* similar to [3, 9, 33]. Data sharing characteristics of an application may have an impact on the propagation of an error occurred on a shared data item to other threads. To calculate these values, the unique number of threads that access to a cache line for read or write purposes is counted. If the unique number of readers and writers equals to one, the corresponding cache line is tagged with the *private* sub-characteristic. If the unique number of readers is more than zero, and the unique number of writers equals to zero; the corresponding cache line

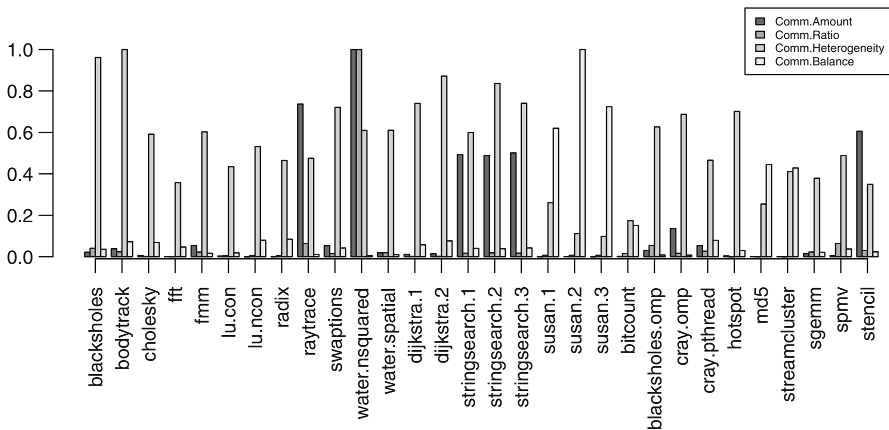

Fig. 3 Communication characteristics of our target programs

is tagged with the *read-only* sub-characteristic. If the same cache line is accessed by more than one thread, and at least one of the accesses is for write purpose; the corresponding cache line is shared among multiple threads. If there are more readers than the writers, the corresponding cache line is tagged with the *producer-consumer* sub-characteristic. On the contrary, if the readers are less than or equal to the writers, the corresponding cache line is tagged with the *migratory* sub-characteristic. We extract each of these sub-characteristics for each cache line, and present the average of each case as a percentage value for a given application.

Figure 4 presents the data sharing characteristics for our target programs. While search algorithms (like *dijkstra* and *stringsearch* versions) have larger read-only data, programs with heavy data-sharing (like *lu*) have high producer-consumer values.

3.2.3 Parallel Programming Characteristics

We consider parallel programming characteristics as influential features in our evaluation. We profile all *pthread* library function calls in our execution to capture multithreaded execution behavior of an application. Our framework supports both *pthread* and *OpenMP* programming models. Since *OpenMP* runtime uses *pthread*s for its implementation, we directly use the same tracking method for programs based on both programming models.

Firstly, we count the number of *pthread_create*() function calls to extract the actual number of threads created in the application. Moreover, we record synchronization events such as critical sections, barriers, condition variables to characterize an application's synchronization behavior. Since inter-thread synchronization events are the key factors to determine the behavior of parallel programs, we consider those in our evaluation. Specifically, we track *pthread_mutex_lock*(),

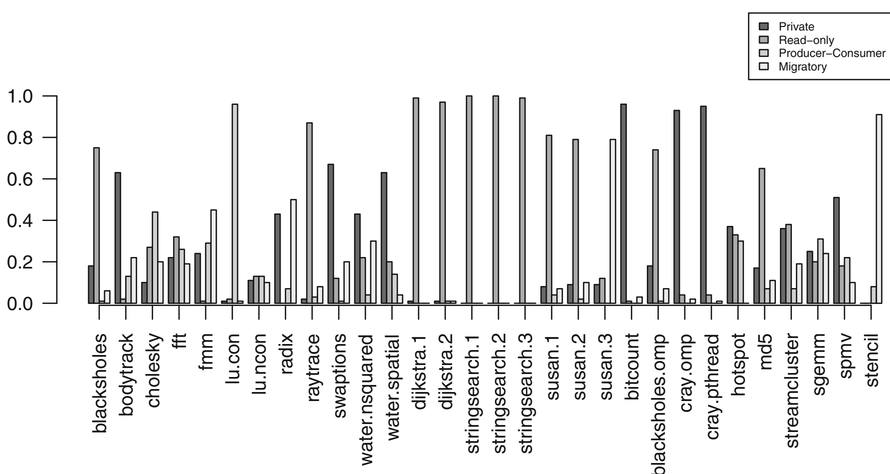

Fig. 4 Data sharing characteristics of our target programs

pthread_mutex_unlock(), *pthread_cond_wait*(), *pthread_cond_broadcast*(), *pthread_barrier_wait*(), *pthread_cond_signal*(), and *sem_wait*() function calls.

The applications with different complexity levels may utilize different kinds of synchronization events. Therefore, we also record several synchronization attributes such as the number of mutex variables used, the number of *pthread_cond_wait*() function calls, the number of *pthread_cond_broadcast*() calls, the number of *pthread_mutex_lock*() and *pthread_mutex_unlock*() function pairs, and the number of *pthread_mutex_lock*() and *pthread_cond_wait*() function pairs in order to distinguish different synchronization events. As an example, condition variables are commonly used synchronization primitives in parallel programming. They can be either used for barrier implementation to block the executing threads until a variable reaches to a specific value, or for implementing semaphores by using signal, broadcast, and wait functions. Since our applications do not utilize *pthread_barrier_wait*(), *pthread_cond_signal*(), *sem_wait*() functions, we omit these features in our list.

In order to define a critical section, we track *pthread_mutex_lock*() and *pthread_mutex_unlock*() function calls, and we calculate total number of instructions executed between these two calls. Since there might be multiple critical sections in an application, we consider the average number of the instructions executed in the critical sections. We consider critical sections as influential in our evaluation due to the behavior of the applications during the critical section execution. When one thread acquires a lock (mutex_lock), any other thread requiring that lock needs to enter in a waiting state without performing any useful computation. If we consider the time spent by the threads while waiting for a critical section inactively, we expect less vulnerability from those threads during this time interval. Since the threads do not perform any computation, it is less probable that a soft error hit during inactive time would cause an output corruption. If the size of the critical sections is too large, an occurred soft error may hit the data of only a single thread that executes the critical section, and the remaining threads may not be affected from this error. Essentially, we observe this behavior from our results: the SDC rates of the programs having more locks, and consequently more instructions in critical sections are lower than the ones with less critical sections. Specifically, when we look at the parallel programming characteristics (see Table 2) and SDC rates (see Fig. 5), you can see that *fmm*, *raytrace*, and *bitcount* programs with larger mutex counts have low SDC rates (considered as NOT VULNERABLE class in our classification model).

While using a mutex can provide a critical section implementation; if a thread needs to know whether a condition is met, using a condition variable lets the thread find out that the event occurred. While the same requirement can be satisfied by using a mutex, and continually polling the condition, the threads would be busy/active instead of just waiting for the condition variable. When a thread invokes *pthread_cond_wait*(), it automatically releases the lock in *pthread_mutex_lock*() function. Therefore, we account those regions as critical sections in our model, and calculate the average number of instructions between *pthread_mutex_lock*() and *pthread_cond_wait*() function calls as well.

Table 2 Parallel programming characteristics of our target programs

Applications	Parallel Programming Characteristics									
	# Of thread creations	# Of mutexes	# Of cond. wait	# Of cond. broadcast	# Of mutex_lock() - unlock()	# Of mutex_lock() - cond_wait()	Av. # of inst. between lock() - unlock()	Av. # of inst. between lock() - cond_wait()	Ratio of inst. inside TF	# of stack pointer accesses inside TF
blacksholes	8	2	0	0	2	0	7310	0	84.53	2867272
bodytrack	9	2967	187	24	2780	187	29742	1879	91.86	1281860
cholesky	7	22066	28	4	22038	28	169060	569	77.12	552006
fft	7	66	49	7	17	49	7683	796	9.66	6738
fmm	7	45230	238	34	44980	238	677887	2684	96.44	1060290
lu-con	7	546	469	67	77	469	8365	5245	89.47	528228
lu-ncon	7	162	133	19	29	133	7836	1741	77.19	34832
radix	7	156	96	41	60	96	7955	1083	16.58	1604
raytrace	7	152771	7	1	152764	7	1290343	329	96.36	2179400
swaptions	8	2	0	0	2	0	7419	0	99.88	8636027
water-nsquared	7	10538	140	20	10398	140	105065	1734	95.09	795704
water-spatial	7	315	140	20	175	140	11272	1716	97.44	1725045
dijkstra.1	8	10	7	1	3	7	7521	160	14.95	31286
dijkstra.2	8	31994	27993	3999	4001	27993	56558	186451	16.47	200546
stringsearch.1	8	43	0	0	43	0	311712	0	0.00	0
stringsearch.2	8	43	0	0	43	0	311712	0	0.00	0
stringsearch.3	8	43	0	0	43	0	311596	0	0.00	0
susan.1	24	115	28	4	87	28	343591	161	0.32	1134
susan.2	32	19	14	2	5	14	37	69	77.09	1739
susan.3	24	27	14	2	12	14	69	74	95.61	1705
bitcount	8	73587	64386	9198	9201	64386	107841	377073	84.51	472352

Table 2 continued

Applications	Parallel Programming Characteristics									
	# Of thread creations	# Of mutexes	# Of cond. wait	# Of cond. broadcast	# Of mutex_lock() - unlock()	# Of mutex_lock() - cond_wait()	Av. # of inst. between lock() - unlock()	Av. # of inst. between lock() - cond_wait()	Ratio of inst. inside TF	# of stack pointer accesses inside TF
blacksholes-omp	7	2	0	0	2	0	7414	0	0.79	3003500
cray-omp	7	2	0	0	2	0	7459	0	96.04	13507571
cray-pthread	8	11	8	1	3	8	7815	156	94.34	2227793
hotspot	7	2	0	0	2	0	7375	0	0.16	27259987
md5	8	2	0	0	2	0	7272	0	8.51	197
streamcluster	16	26	6	3	20	6	3770	77	84.27	17032
sgemm	7	3	0	0	3	0	8624	0	0.01	10406
spmv	7	2	0	0	2	0	7428	0	0.00	1718
stencil	7	2	0	0	2	0	7372	0	0.00	1554

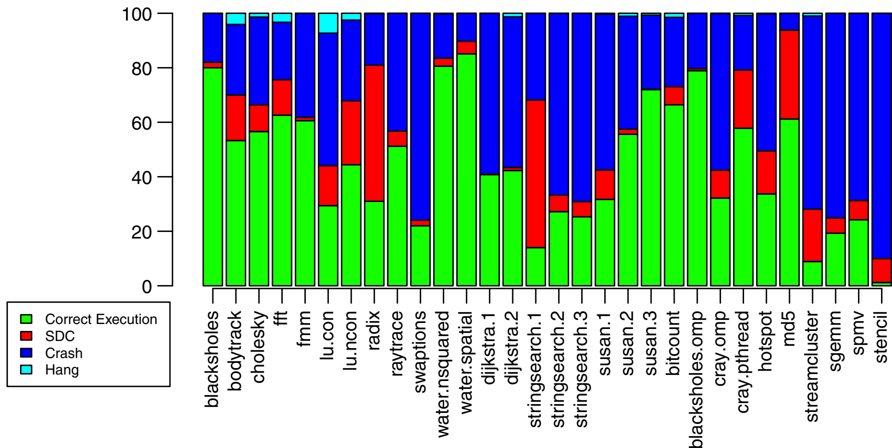

Fig. 5 Fault injection results

We also consider the different implementations of critical sections (whether based on lock/unlock pair or lock/cond_wait pair) as distinct characteristics due to the fact that threads spend those critical sections busy or inactive state. We believe that this busy/inactive waiting state of the threads may affect the soft error vulnerability of the application.

Furthermore, we record the average number of instructions in a thread function to show the complexity of the thread functions, and the total number of stack pointer accesses inside the thread functions to track local variable accesses. Those features may affect the reliability behaviour of the applications either directly or indirectly. As an example, if the number of stack pointer accesses inside a thread function is relatively high, an occurred error may corrupt the private data of the executing thread rather than shared data.

Table 2 presents the parallel programming characteristics of a set of applications.

3.2.4 Performance Characteristics

We analyze several performance metrics in this section including *the dynamic instruction count*, *the percentage of branch instructions executed*, *the percentage of load/store instructions executed*, *the number of context switches*, *the TLB miss count*, and *the total cache miss rate*.

While *the number of context switches* is related to operating system that the program is executing on, the other sub-characteristics are commonly used performance metrics in the literature and depend on the micro-architecture used. Some of these performance sub-characteristics have a high impact on the reliability requirements of applications [29]. As an example, an application with a low cache miss rate may utilize the processor resources more frequently than the other applications and its live data resident in the cache might be more vulnerable to errors. An occurrence of error on this data may easily propagate to the other sections of the program, which directly affects the SDC rate of the target application. As

another example, the percentage of load/store instructions might have a great impact on the application resilience, since the successive instructions may use the loaded/stored values [14]. Moreover, the number of context switches is an important metric that may affect the performance of a parallel application. There might be some direct overheads such as saving/restoring registers, reloading TLB, and flushing pipeline, or indirect overheads caused by cache sharing among the application threads. Since we focus on faults in register file, the utilization of registers by the switched threads may impact the vulnerability.

The metrics that might affect the performance of an application may also affect the vulnerability of an application; therefore, we include the performance sub-characteristics of parallel applications to predict the soft error vulnerability of the corresponding applications. Table 3 presents the performance characteristics of a set of applications.

3.2.5 Characterization Tools

To collect characteristics of a parallel application during its execution, we utilize *Pin* [24], a dynamic binary instrumentation tool. Specifically, we use *numalize* [10], a Pin-based memory tracing tool, to extract thread communication matrix and thread memory access behaviour. Additionally, we update this tool to extract data-sharing sub-characteristics during the application execution. We also implement a Pin tool to track *pthread* function calls and instructions. Moreover, we use *perf* tool [1] to gather performance characteristics.

3.3 Fault Injection Framework

We perform fault injection experiments to measure the soft error vulnerability of our benchmark applications. In this work, we use PINFI [41], an assembly-level instrumentation tool for x86-architecture processors. We prefer using an assembly code level fault injection due to its high accuracy and low cost in terms of both time and hardware. While physical beam experiments, performed directly on real hardware components, present soft error rates in realistic physical conditions, they are hard and costly [30]. On the other hand, microarchitectural fault injection experiments based on architectural simulation may lack accuracy in terms of target architecture details and the simulation may make the long running fault injection experiments impractical [31, 39]. Performing hardware-, simulator- or assembly-level fault injections presents a trade-off between accuracy vs. time vs. cost [6, 25]. The state-of-the-art fault prediction methods [14, 18, 23, 29] follow a similar way by performing high-level fault injection experiments to collect more input data for their ML frameworks, which require as much as possible data points. While there are some efforts that propose soft error evaluation tools to accelerate simulation-based fault injection experiments in order to eliminate large fault injection overheads [8], they focus on microarchitectural characteristics. Since we target analyzing the effects of high-level parallel programming features on the faults reaching to the application level in our prediction framework, we prefer utilizing the assembly-level PINFI framework for fault injection experiments.

Table 3 Performance characteristics of our target programs

Applications	Performance Characteristics						
	# Of instructions	% Of branch instructions	% Of memory store instructions	% Of memory load instructions	# Of context switches	% Of d-TLB misses	Cache miss rate (%)
blacksholes	322,067,753	9.65	8.19	22	18	0	0
bodytrack	1,221,094,063	8.95	4.65	19.09	705	0	5.26
cholesky	1,157,208,970	10.24	10.06	27.06	293	0	4.16
fft	3,966,865	15.97	14.53	22.55	69	0	0
fmm	2,495,344,110	9.44	2.52	14.48	302	0	1.25
lu-con	374,672,516	11.42	8.84	21.74	497	0	1.52
lu-ncon	6,755,904	12.03	10.49	24.19	171	0.01	0
radix	276,321,114	13.6	15.24	21.05	118	0	0.21
raytrace	5,053,472,794	11.36	2.95	38.27	357	0	27.59
swaptions	1,850,247,092	11.08	7.86	27.21	42	0	1.43
water-nsquared	460,053,343	10.6	11.25	27.97	151	0.01	3.16
water-spatial	449,358,785	9.98	10.85	27.94	180	0	0
dijkstra.1	2,049,008,292	17.34	19.94	29.53	53	0	0.03
dijkstra.2	2,086,238,122	17.1	19.3	28.51	28,003	0.0005	0.35
stringsearch.1	17,261,817,538	25.88	0.25	29.48	1,115	0.0004	7.86
stringsearch.2	10,235,589,415	23.07	0.01	46.42	1,975	0.0004	8.12
stringsearch.3	10,587,897,254	22.47	0.87	47.51	2,106	0.0004	7.82
susan.1	25,604,572	9.14	1.43	22.2	84	0	0
susan.2	2,505,445	8.32	6.84	35.83	82	0	0
susan.3	10,224,164	4.07	36.94	44.78	89	0	0
bitcount	101,203,986	17.18	13.23	12.37	64,438	0.0161	13.55
blacksholes-omp	342,669,007	10.77	7.72	21.55	4	0.0001	0
cray-omp	2,378,564,425	6.29	14.44	46.54	9	0	0.08
cray-pthread	1,178,891,158	8.3	8.52	21.82	16	0	0.46
hotspot	7,207,307,204	17.82	15.94	21.68	11	0	0.09
md5	4,032,348	13.81	16.68	22.93	16	0	0
streamcluster	1,942,325	13.6	14.32	23.14	12,281	0.0005	0
sgemm	248,757,223	15.57	12.13	33.67	19	0.0002	11.1
spmv	51,395,480	21.1	6.05	25.57	5	0.0003	0
stencil	7,183,198,670	3.28	1.56	47.7	13	0	5.76

PINFI emulates transient faults in the register file by flipping random bits in the registers using binary instrumentation. As a result of a fault injection experiment, we collect the SDC, crash, hang, and correction execution rates for each application. Our prediction models utilize SDC or correct execution rates as a response variable.

For a fault injection point, we randomly select one register among 132 user-accessible registers and one bit among 64 bits of the target register as our target architecture indicates its fault location space with available register bits. We use 1000 fault injections per each benchmark by using statistical approach [19] with the confidence level of 95% and the error margin 3%. We take a fixed number of fault injections, i.e., 1000, for all target applications, since an increase in the program execution time does not affect the sample size with the given confidence level and the error margin in the calculation. Figure 5 shows the results of fault injection experiments for each application by demonstrating the correct execution, SDC, crash, and the hang cases.

3.4 Prediction Model

We evaluate two approaches for vulnerability prediction problem: Regression model and Classification model. Our first prediction model relies on regression analysis, where we target to predict the correct execution rates of the parallel programs without executing fault injection experiments. In our model, the SDC rates of the applications are not predicted directly since those values might be too small to make accurate predictions. Therefore, we prefer to predict the correct execution rates of the applications, which are calculated by subtracting the SDC, the crash, and the hang rates from 1.0. Since these rates are represented with real numbers between 0.0 and 1.0, our problem is modeled as a regression problem as the first attempt. Since it is difficult to predict SDC rates, as an alternative approach, we evaluate SDC prediction as a classification problem. We assume that there are two distinct classes of SDC rate ranges, which can provide a classification for the vulnerability level of the applications. We define the vulnerability level of target program as following:

$$VL = \begin{cases} \text{VULNERABLE}, & \text{if } SDC \geq 0.1 \\ \text{NOT VULNERABLE}, & \text{otherwise} \end{cases} \quad (5)$$

We assume that by predicting a program's vulnerability class, we can have a basic idea about the soft error characteristics of the given application.

3.4.1 Machine Learning Algorithms

We use three machine learning methods: Support Vector Machine (SVM), Gradient Boosting (GB), and Random Forest (RF). Support Vector Machine is a commonly used supervised learning based method with high accuracy values in classification problems [7]. This method is extended to regression problems with Support Vector Regressions [12]. Gradient Boosting is an ensembling based algorithm which combines multiple learners sequentially [13]. Random Forest is another kind of ensembling algorithm, which is based on combined multiple independent decision trees to make accurate predictions [5]. We use these three algorithms for both approaches, namely predicting the correct execution rates as a regression model, and

predicting the vulnerability levels as a classification model for the given parallel program.

3.4.2 Features

We utilize 25 application characteristics given in Table 1 as features in our prediction model. Either related or unrelated features may affect the prediction accuracy of a machine learning model. We perform a correlation analysis to extract the most significant features and determine the most influential features on the correct execution of the programs. Specifically, we apply Pearson's correlation method and Spearman's rank correlation method, and select 6 and 4 features, respectively, as more correlated features with the correct execution rates of the applications. Table 4 and Table 5 present those selected features with their correlation values, respectively. We can see that both methods show higher positive correlation for *migratory* and the *percentage of thread instructions over the total number of instructions* with correct execution rates. On the other hand, the *total number of instructions* are found to be negatively correlated. This implies that parallel programs with relatively larger thread functions and parallel programs with threads having migratory (the reader threads are less than or equal to the writer threads) characteristics tend to complete their execution correctly; while larger programs, e.g., programs having more instructions, tend to finish with an error or data corruption. By using this observation, the parallel program developers may focus on thread characteristics if they aim to write fault-tolerant programs.

As a result, we include *migratory* sub-characteristic from the data sharing features, the *percentage of thread instructions over the total number of instructions* from the parallel programming characteristics, and the *total number of instructions* from the performance characteristics. Additionally, the *private* feature from the thread communication characteristics is selected based on Spearman's correlation method. The Pearson correlation method additionally selects the *communication ratio* from the thread communication characteristics, and *the percentage of branch and memory load instructions over the total number of instructions* from the

Table 4 Selected features by Pearson's correlation method

Features	Pearson's correlation value
Comm. ratio	0.347
Migratory	0.275
% Of instructions inside a thread function relative to the total # of instructions	0.430
# Of instructions	–0.535
# Of branch-instructions	–0.448
# Of memory-load-instructions	–0.298

Table 5 Selected features by Spearman's rank correlation method

Features	Spearman's rank correlation value
Private	0.494
Migratory	0.455
% Of instructions inside a thread function relative to the total # of instructions	0.454
# Of instructions	–0.424

performance characteristics. We discuss the performance of the regression models based on those selected features in Sect. 4.2.1.

Since we have only two-classes (vulnerable vs not vulnerable) in our classification model, we do not prefer to make correlation analysis.

3.4.3 Performance Metrics

We mainly evaluate the prediction accuracy metric to compute the success level of our prediction models. Specifically, for the regression model, *Paccuracy* is calculated as follows:

$$P_{accuracy} = \left(1 - \frac{|Predicted_{rate} - Observed_{rate}|}{Observed_{rate}}\right) * 100\%$$

where $Predicted_{rate}$ is the value predicted by the corresponding machine learning model, and $Observed_{rate}$ is the actual value taken as the results of fault injection experiments. It simply calculates the relative error between the predicted value and the observed value. A $P_{accuracy}$ value close to 100% indicates an accurate prediction model.

For the classification model, accuracy is simply calculated as follows:

$$accuracy = \frac{number\ of\ correctly\ labeled\ instances}{total\ number\ of\ instances}$$

Additionally, for our classification model evaluation, we use precision, recall, and F-score metrics. We compute our measures for both NOT VULNERABLE (NV) and VULNERABLE (V) classes. The metrics for NV class (similarly for V class) are computed as follows:

$$precision(NV) = \frac{number\ of\ NV\ instances\ correctly\ labeled}{total\ number\ of\ instances\ labeled\ NV}$$

$$recall(NV) = \frac{number\ of\ NV\ instances\ correctly\ labeled}{number\ of\ instances\ that\ are\ NV}$$

$$F - score(NV) = 2 \times \frac{precision(NV) \times recall(NV)}{precision(NV) + recall(NV)}$$

3.4.4 Training and Testing Phases

In a machine learning algorithm, the input dataset should be divided into two parts, which are training and testing sets. The algorithm should be trained with the training data, and the success of the algorithm should be measured by using a set of unseen data, which is the testing set. In our model, we firstly apply leave-one-out cross validation method in order to eliminate overfitting occurrence. In this cross validation method, one datum is left for testing, and the model is trained with the rest of the input data. This process is repeated by the size of the input data. The leave-one-out cross validation method is a suitable technique especially for the small-size datasets. Alternatively, in our regression model, we utilize 80:20 split method, in which 80% of dataset is used as the training data, and 20% of dataset is used as the testing data. Since the accuracy of the prediction model might change depending on the data used in the training and testing sets, an average of 10 runs is taken for each test case. We explore different splitting in our classification model, where the details are explained in Sect. 4.2.2. We evaluate the effectiveness of our prediction framework systematically through testing phase by comparing performance metrics given in Sect. 3.4.3, which are the common metrics defined for the accuracy evaluation of the machine-learning techniques.

3.4.5 Model Tuning

Our three ML models have multiple hyperparameters that affect the accuracy level of the prediction. A grid-search based method is performed to decide the final set of model parameters with the highest prediction accuracy for testing phase.

4 Experimental Study

4.1 Experimental Setup

We perform our experiments in an Intel Xeon-based workstation with 2x Xeon Silver 4114 processors, and 32 GB main memory. We select *pthread* implementations of *blackscholes*, *bodytrack*, *radix*, *swaptions* programs from PARSEC [4], and *cholesky*, *fft*, *ffm*, $lu-contiguous$, $lu-non-contiguous$, *raytrace*, $water-nsquared$, $water-spatial$ from SPLASH-2 [42] benchmark suites. Two implementations of *dijkstra* using either shared or private queues, and three implementations of *stringsearch* using three different algorithms (i.e., Pratt-Boyer-Moore String Search, Case-sensitive Boyer-Moore-Horspool String Search, and Case-Insensitive Boyer-Moore-Horspool String Search), three implementations of *susan* that either smooths an input image, or recognizes edges or corners in the input image, and *bitcount* applications are selected from ParMiBench benchmark suite [16]. Additionally, we use both *pthread* and *OpenMP* versions of *cray*, and *pthread* versions of *md*5 and *streamcluster* from Starbench benchmark suite [2]. Finally, we use *OpenMP* versions of *sgemm*, *spmv*, and *stencil* from Parboil

benchmark suite [38]. Therefore, a total of 30 parallel programs is profiled in our dataset. We utilize both *pthread* and *OpenMP* implementations of a set of applications since they might show different vulnerability characteristics to soft errors [34].

4.2 Experimental Results

In this section, we present the results of different machine learning algorithms in predicting soft error vulnerability of the applications. We also perform feature selection (only for the regression model) and hyperparameter tuning (for both model) to increase the accuracy level of each algorithm. We analyze the regression and the classification results separately in the following subsections.

4.2.1 Regression Results

In the concept of regression problem, the correct execution rates of the applications are estimated utilizing three machine learning algorithms, as mentioned earlier in Sect. 3.4.1. We utilize a total of 24 applications in this section since the applications with very low correct execution rates lead to incorrect estimation that decreases the average prediction accuracy aggressively. Therefore, we exclude 6 applications (i.e., *stencil*, *streamcluster*, *stringsearch*1, *sgemm*, *swaptions*, and *spmv*) with correct execution rates less than 0.25 by means of outliers. We expect that the accuracy of our regression models might be low for the applications with very low correct execution rates, however this is a general problem for regression models resulting from observing very small output values. We believe that dropping the outlier data is fair in prediction models. To deal with this limitation and be able to work with programs having very low correct execution rates, we offer our classification model which works feasible for the complete dataset (as given in Sect. 4.2.2).

As described in Section 3.4.4, we firstly use the leave-one-out cross validation method since we have a small set of applications in our dataset. The success level of each algorithm including feature selection and parameter tuning approaches is shown in Fig. 6. Our first observation is that each algorithm demonstrates a different success level with different number of features. As an example, the best performing algorithm is the base SVM algorithm with 67.2% average prediction accuracy rate based on a total of 25 features. On the other hand, the tuned version of the RFR algorithm is the best alternative with 72.2% average prediction accuracy rate, when we use only 6 features based on Pearson correlation method. The base SVM algorithm is performing best with 71.2% average prediction accuracy rate, when a total of 4 features is used based on Spearman's correlation method. Although these average prediction rates do not seem to be very high, the values are promising when we consider only a total of 24 inputs in our dataset.

Our second observation is that, the effect of feature selection is appeared as different in the success level of each algorithm. As an example, the base SVM algorithm has low accuracy rates with 25 and 6 features; however, its accuracy is the best with 71.2% average prediction accuracy in case of 4 features. On the other

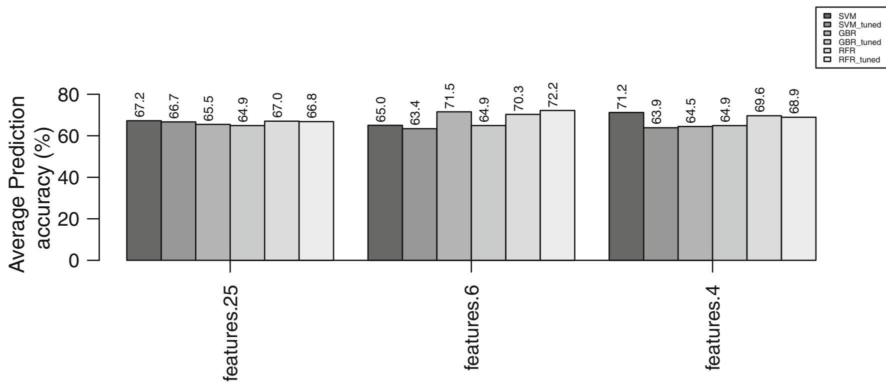

Fig. 6 The average prediction accuracy for correct execution rates with different number of features

hand, the tuned version of GBR algorithm is not affected with different number of features by having 64.9% accuracy rate, and the base GBR algorithm increases its accuracy level from 65.5% to 71.5%, when we decrease the number of features from 25 to 6. As another result, RFR algorithm benefits from 6 features a lot, and its performance is the best with the 70.3% and 72.2% accuracy rates for the base and the tuned versions, respectively. Those results indicate that the feature selection approach further improves the accuracy of a set of algorithms with more than 5%. On the other hand, some algorithms do not show improved results with different number of features based on that dataset.

As a second set of experiments, we split our dataset based on 80:20 method, in which the 80% of input dataset (i.e., 19 applications) is used in training phase, and 20% of input dataset (i.e., 5 applications) is used in testing phase. Since the prediction accuracy might change based on the data sample used in training/testing phases, we randomly select the data points to be used in training and testing phases, and report the average of 10 repeated experiments. The accuracy rate of each algorithm including feature selection and parameter tuning approaches is shown in

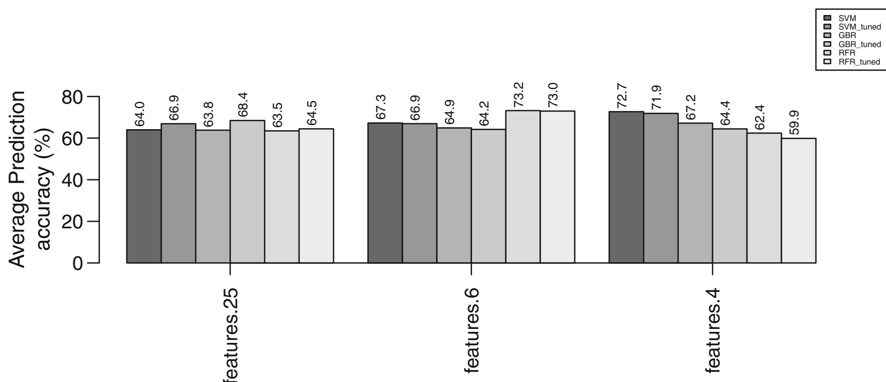

Fig. 7 The average prediction accuracy when an 80:20 split method is used in dataset

Fig. 7. In this case, the best performing algorithm is the tuned version of GBR algorithm with 68.4% accuracy rate based on 25 features. On the other hand, the base RFR shows the best performance with 73.2% accuracy rate in case of 6 features used. Finally, the base SVM shows 72.7% accuracy rate when we use 4 features based on Spearman's correlation method. These results are mostly consistent with previous results, where RFR algorithms are the best alternatives with 6 features, and the base SVM is the best alternative in case of 4 features. The best performing algorithm is the base SVM with 67.2% accuracy rate in leave-one-out cross validation when we use a total of 25 features; contrarily, the tuned version of GBR is the best option with 68.4% accuracy rate in 80:20 split method in case of 25 features. However, the values are quite close to each other. When we consider the grand average of all methods with all different case of feature sets, the average of 67.15% accuracy rate is obtained with leave-one-out cross validation method, and the average of 66.62% accuracy rate is obtained with 80:20 split method.

After these results, we select a representative machine learning model, the tuned version of RFR, which has the highest accuracy level with 72.2% among different algorithms, and different number of features based on leave-one-out cross validation method. Then, we analyze the application-specific prediction results in Fig. 8. As seen, the prediction results of only 4 applications (i.e., *lu.con*, *radix*, *susan*.1, and *cray.omp*) decrease the average value aggressively by having very small prediction accuracy. When we analyze the results of fault injection experiments in Fig. 5, those applications also have relatively low correct execution rates compared to the remaining ones. Therefore, our prediction model does not perform well for the applications with low correct execution rates. On the other hand, the prediction results of the remaining 20 applications are higher than (or close to) the average with the maximum of 98.5% for the *cholesky* and *susan*.2. Those results apparently demonstrate that we can make accurate predictions for a set of applications by

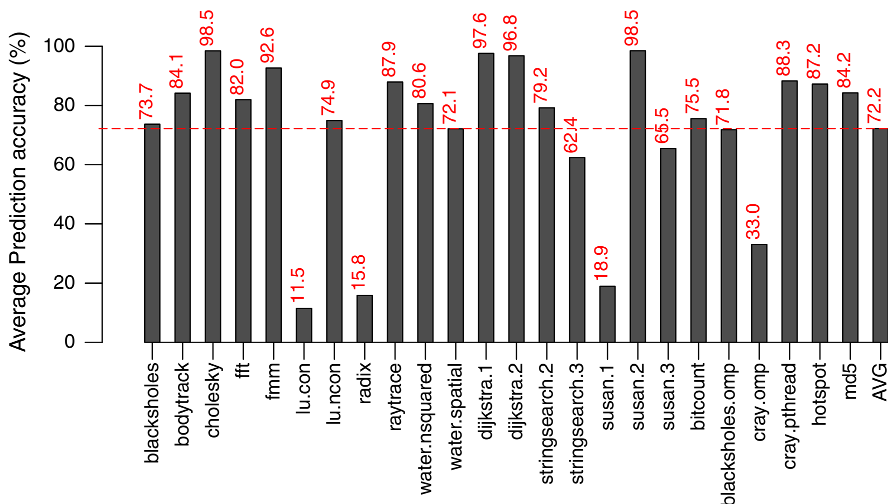

Fig. 8 Application-specific prediction results of RFR-tuned model with 6 features

analyzing high level and low level application features. Rather than performing long-lasting fault injection experiments, prediction results may give us an idea about the reliability behaviour of the applications.

4.2.2 Classification Results

We apply the same prediction methods for our classification model as in the regression model. We include all 25 features in our evaluation.

Firstly, we evaluate our classification model using complete dataset (30 instances), and apply leave-one-out cross validation. Table 6 presents our prediction results for each classifier algorithm. We observe that imbalance in the dataset, with 18 NOT VULNERABLE, 12 VULNERABLE classes, results in low accuracy values. While the precision and recall values are better for NOT VULNERABLE (NV) cases, we get lower measures for VULNERABLE (V) cases. The accuracy results seem to be biased against VULNERABLE cases. Although it is a minority class in the dataset, it might be more important to correctly predict the applications from this class.

Due to the problems of unbalanced data, for better accuracy, we construct balanced dataset including equal number of VULNERABLE and NOT VULNERABLE classes. Since there are many alternatives to build balanced data (eliminating 6 NOT VULNERABLE cases among 18), we evaluate accuracy values for random 30% of all possible choices (approximately 6000 datasets) for shortest-running RF method. Then we select the datasets with the best result (two datasets with the same accuracy in our case), and apply other methods for these balanced datasets as well.

Table 7 and Table 8 present our prediction results for the two best balanced datasets (Dataset-I and Dataset-II). We achieve 88% accuracy (21 correct predictions out of 24 instances) with tuned version of Gradient-Boosting method. We also observe that other methods exhibit more balanced values for precision/recall measures, which indicates the low accuracy results from the ineffectiveness of the algorithm rather than a problem in the dataset used. Therefore, we can rely on the stability of both dataset.

We also conduct experiments by splitting our 24 instance-datasets as training and test data, and apply our algorithms to observe the prediction accuracy. Since we need to have balanced data in both training and test sets, we include the same

Table 6 Classification results for full dataset

Method	Acc.	Prec. (NV)	Recall (NV)	F-score (NV)	Prec. (V)	Recall (V)	F-score (V)
SVM	0.57	0.63	0.67	0.65	0.45	0.42	0.43
SVM_tuned	0.57	0.63	0.67	0.65	0.45	0.42	0.43
GB	0.67	0.72	0.72	0.72	0.58	0.58	0.58
GB_tuned	0.63	0.73	0.61	0.67	0.53	0.67	0.59
RF	0.67	0.70	0.78	0.74	0.60	0.50	0.55
RF_tuned	0.67	0.70	0.78	0.74	0.60	0.50	0.55

Table 7 Classification results for Dataset-I

Method	Acc.	Prec. (NV)	Recall (NV)	F-score (NV)	Prec. (V)	Recall (V)	F-score (V)
SVM	0.38	0.36	0.33	0.35	0.38	0.42	0.40
SVM_tuned	0.63	0.67	0.50	0.57	0.60	0.75	0.67
GB	0.54	0.55	0.50	0.52	0.54	0.58	0.56
GB_tuned	**0.88**	**0.80**	**1.00**	**0.89**	**1.00**	**0.75**	**0.86**
RF	0.79	0.73	0.92	0.81	0.89	0.67	0.76
RF_tuned	0.75	0.71	0.83	0.77	0.80	0.67	0.73

Table 8 Classification results for Dataset-II

Method	Acc.	Prec. (NV)	Recall (NV)	F-score (NV)	Prec. (V)	Recall (V)	F-score (V)
SVM	0.42	0.40	0.33	0.36	0.43	0.50	0.46
SVM_tuned	0.42	0.40	0.33	0.36	0.43	0.50	0.46
GB	0.63	0.62	0.67	0.64	0.64	0.58	0.61
GB_tuned	**0.88**	**0.85**	**0.92**	**0.88**	**0.91**	**0.83**	**0.87**
RF	0.79	0.73	0.92	0.81	0.89	0.67	0.76
RF_tuned	0.67	0.64	0.75	0.69	0.70	0.58	0.64

number of instances from two different classes. Moreover, due to low performance in the leave-one-out cross validation, we skip SVM in this part of our evaluation. We also demonstrate the impact of splitting data differently.

In our experiments, we consider four different splitting scenarios: 12 training/12 test (12/12), 14 training/10 test (14/10), 16 training/8 test (16/8), 18 training/6 test (18/6). We apply random splitting by assuring the balanced distribution of classes in both training and test sets, and perform 10 random splits for each case. Figures 9 and 10 give accuracy distribution box-plots for each algorithm for Dataset-I and Dataset-II, respectively.

We observe that the splits with larger training set generally perform better. However, we achieve 69% average accuracy at most. Since the dataset is relatively small, comparing accuracy values becomes insignificant. For instance, in 18/6 splitting, the possible accuracy values are 100%, 83%, 67%, and so on. Mispredicting only one test case would result in 17% accuracy loss. Therefore, training/test splitting methodology for our classification approach does not yield similar results to the leave-one-out cross validation.

The classification results demonstrate that ML-based classifiers have a potential for vulnerability prediction of parallel programs. Since the information about the vulnerability level provides an important insight about the fault tolerance of a program, one can get benefit from our approach by formulating the problem as a classification. By using the prediction outcome, she can decide whether to apply any fault tolerance techniques.

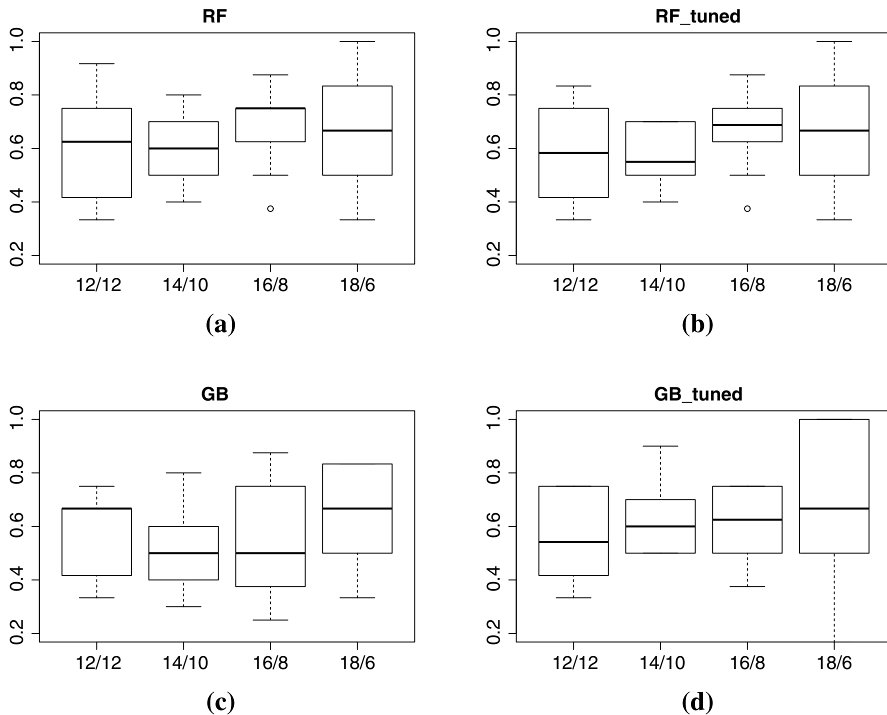

Fig. 9 Accuracy distribution with different splits for each algorithm (Dataset-I)

5 Related Work

To overcome long execution times of the fault injection experiments, prediction-based approaches for soft error vulnerability evaluation have been proposed in the literature.

Guo et al. [14] predicted the SDC, interrupt and success rates of a set of unseen applications by training various regression models using a set of small kernels. In their model, firstly, various instruction groups, resilience patterns, resilience weight, and instruction-order information are used as features, then more related features are selected by applying a filtering method. They also performed hyperparameter tuning methods to improve the prediction accuracy of the models used.

Laguna et al. [18] used machine learning to classify instructions based on their probability to produce erroneous outputs by considering instruction features like instruction type, the properties of the basic block and function the instruction belongs to, and the instructions that the instruction affects. They utilized SVM classifier to predict the instructions that generate silent output corruption. Their compiler framework duplicates only the selected instructions to mitigate the effect of the errors. Similarly, Liu et al. [21] trained a random forest model to predict SDC-causing instructions in a given application by using instruction-specific features. Additionally, Liu et al. [22] proposed instruction SDC vulnerability

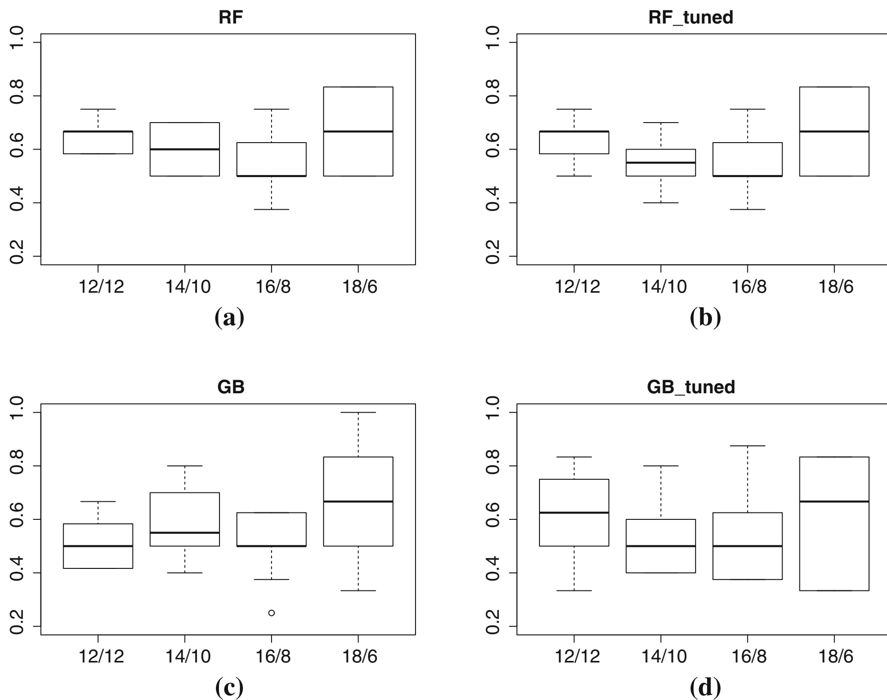

Fig. 10 Accuracy distribution with different splits for each algorithm (Dataset-II)

prediction based on an LSTM network model with instruction-specific features, and compared their approach with SVM and decision tree classifiers. Yang and Wang [43] built a machine learning model to predict the SDC vulnerability of instructions with a lower cost of fault injection by executing partial experiments. They include instruction and error propagation features to train their Classification and Regression Tree (CART)-based model.

Lu et al. [23] proposed heuristics to predict SDC-proneness of program variables by using static and dynamic analysis, and then extend their model to predict overall SDC-proneness of an application relative to another application. They also applied selective protection to reduce full-duplication overhead.

Oliveira et al. [29] presented program vulnerability factor (PVF) prediction mechanism for HPC applications. They built an SVM-based model by using application characteristics like cache miss rate, TLB miss rate, percentage of branch and load/store instructions. Vishnu et al. [40] utilized semantic information gathered from fault injection experiments to construct features for a machine learning model. They created application fault models to predict whether the extreme-scale applications will result in an error if there is a multi-bit memory error. Mutlu et al. [27] proposed a machine learning model to evaluate ground-truth prediction for soft-error of iterative methods. They utilized AdaBoost regression to construct a ground-truth predictor to reduce the size of the fault injection experiments.

Rosa et al. [8] proposed a soft error assessment flow including simulation-based fault injection and soft error criticality evaluation. They utilized ML techniques to identify the correlation between fault injection results and multicore parameters such as application and platform characteristics.

Nie et al. [28] used a machine learning approach to predict GPU errors by considering both temporal and spatial features, like application-specific characteristics, temperature/power consumption, and node location. They utilized Logistic Regression (LR), Gradient Boosting Decision Tree (GBDT), Support Vector Machine (SVM), and Neural Network (NN) to analyze ML-based prediction models for GPU soft errors.

Kalra et al. [17] presented a framework to predict vulnerability of GPU applications by using micro-architecture independent features. Their framework identifies a set of scalar and vector instructions as program features, and utilizes linear regression and k-nearest neighbor to capture the statistical relationship between the program features and the program resiliency.

Alternatively, Li et al. [20] proposed a compiler module that can predict SDC probability of individual instructions as well as whole program by analyzing error propagation in a program. They also applied selective duplication for individual instructions to reduce overall SDC rate.

Our work has significant differences compared to studies listed here. Specifically, we focus on prediction of soft error vulnerability for parallel programs (not a vulnerability of a single instruction). We propose a machine learning model based on high-level parallel programming features, for the first time, as opposed to previous studies. We also propose a novel classification model for vulnerability level prediction.

6 Conclusions

We present a soft error vulnerability prediction approach for parallel programs using machine learning. Our approach evaluates both correct execution rates and vulnerability level for target programs by applying a set of ML algorithms and techniques. We evaluate the effectiveness of our prediction framework through performance metrics defined for the accuracy evaluation of the machine-learning techniques. Our results demonstrate that ML-based approach is useful for reliability evaluation. A regression-based approach is more suitable for predicting the exact correct execution rates of the applications (especially for the ones having higher correct execution rates). On the other hand, a higher accuracy rate can be achieved by designing the problem as a binary classification problem, and predicting the relative vulnerability level.

Since our model performs highly accurate results for soft error outcomes, it can be utilized as a fault tolerance level recommendation for the safety-critical systems without performing costly fault injection experiments. Based on the prediction of vulnerability characteristics of applications, a limited number of protection resources (such as additional cores) might be utilized for highly vulnerable ones in a given multi-application workload. Such applications might be selectively

duplicated (to enable fault detection), triplicated (to enable fault correction), or re-executed at different times compared to the remaining applications in the workload to provide a cost-effective and reliable execution environment.

References

1. perf: Linux profiling with performance counters (2015). https://perf.wiki.kernel.org/index.php/Main_Page
2. Andersch, M., Juurlink, B., Chi, C.C.: A benchmark suite for evaluating parallel programming models. In: Proceedings 24th Workshop on Parallel Systems and Algorithms (2011)
3. Barrow-Williams, N., Fensch, C., Moore, S.: A communication characterisation of splash-2 and parsec. In: IEEE International Symposium on Workload Characterization (IISWC) (2009)
4. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: Characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (2008)
5. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
6. Chatzidimitriou, A., Bodmann, P., Papadimitriou, G., Gizopoulos, D., Rech, P.: Demystifying soft error assessment strategies on arm cpus: microarchitectural fault injection vs. neutron beam experiments. In: 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 26–38 (2019)
7. Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn. **20**(3), 273–297 (1995)
8. da Rosa, F.R., Garibotti, R., Ost, L., Reis, R.: Using machine learning techniques to evaluate multicore soft error reliability. IEEE Trans. Circuits Syst. I: Reg. Pap. **66**(6), 2151–2164 (2019)
9. Deniz, E., Sen, A., Kahne, B., Holt, J.: Minime: Pattern-aware multicore benchmark synthesizer. IEEE Trans. Comput. **64**(8), 2239–2252 (2015). https://doi.org/10.1109/TC.2014.2349522
10. Diener, M., Cruz, E.H., Pilla, L.L., Dupros, F., Navaux, P.O.: Characterizing communication and page usage of parallel applications for thread and data mapping. Perform. Eval. **88–89**, 18–36 (2015)
11. Diener, M., Cruz, E.H.M., Alves, M.A.Z., Alhakeem, M.S., Navaux, P.O.A., Heiß, H.U.: Locality and balance for communication-aware thread mapping in multicore systems. In: European Conference on Parallel Processing (Euro-Par) (2015)
12. Drucker, H., Burges, C.J.C., Kaufman, L., Smola, A., Vapnik, V.: Support vector regression machines. In: Proceedings of the 9th International Conference on Neural Information Processing Systems (NIPS) (1996)
13. Friedman, J.H.: Stochastic gradient boosting. Comput. Stat. Data Anal. **38**(4), 367–378 (2002)
14. Guo, L., Li, D., Laguna, I.: PARIS: Predicting Application Resilience Using Machine Learning. arXiv e-prints arXiv:1812.02944 (2018)
15. Hari, S.K.S., Tsai, T., Stephenson, M., Keckler, S.W., Emer, J.: Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (2017)
16. Iqbal, S.M.Z., Liang, Y., Grahn, H.: Parmibench—an open-source benchmark for embedded multiprocessor systems. IEEE Comput. Archit. Lett. **9**(2), 45–48 (2010)
17. Kalra, C., Previlon, F., Li, X., Rubin, N., Kaeli, D.: Prism: Predicting resilience of gpu applications using statistical methods. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis (2018)
18. Laguna, I., Schulz, M., Richards, D.F., Calhoun, J., Olson, L.: Ipas: Intelligent protection against silent output corruption in scientific applications. In: 2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (2016)
19. Leveugle, R., Calvez, A., Maistri, P., Vanhauwaert, P.: Statistical fault injection: quantified error and confidence. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE) (2009)
20. Li, G., Pattabiraman, K., Hari, S.K.S., Sullivan, M., Tsai, T.: Modeling soft-error propagation in programs. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (2018)

21. Liu, L., Ci, L., Liu, W., Yang, H.: Identifying sdc-causing instructions based on random forests algorithm. KSII Trans. Internet Inf. Syst. **13**, 1566–1582 (2019)
22. Liu, Y., Li, J., Zhuang, Y.: Instruction sdc vulnerability prediction using long short-term memory neural network. In: Gan, G., Li, B., Li, X., Wang, S. (eds.) Advanced Data Mining and Applications, pp. 140–149. Springer, Cham (2018)
23. Lu, Q., Pattabiraman, K., Gupta, M.S., Rivers, J.A.: Sdctune: A model for predicting the sdc proneness of an application for configurable protection. In: International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) (2014)
24. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2005)
25. Mittal, S., Vetter, J.S.: A survey of techniques for modeling and improving reliability of computing systems. IEEE Trans. Parall. Distrib. Syst. **27**(4), 1226–1238 (2016)
26. Mukherjee, S.: Architecture Design for Soft Errors. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
27. Mutlu, B.O., Kestor, G., Cristal, A., Unsal, O., Krishnamoorthy, S.: Ground-truth prediction to accelerate soft-error impact analysis for iterative methods. In: 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC) (2019)
28. Nie, B., Xue, J., Gupta, S., Patel, T., Engelmann, C., Smirni, E., Tiwari, D.: Machine learning models for gpu error prediction in a large scale hpc system. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (2018)
29. Oliveira, D., Moreira, F.B., Rech, P., Navaux, P.: Predicting the reliability behavior of hpc applications. In: International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) (2018)
30. Oliveira, D.A.G.D., Pilla, L.L., Hanzich, M., Fratin, V., Fernandes, F., Lunardi, C., Cela, J.M., Navaux, P.O.A., Carro, L., Rech, P.: Radiation-induced error criticality in modern hpc parallel accelerators. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 577–588 (2017)
31. Parasyris, K., Tziantzoulis, G., Antonopoulos, C.D., Bellas, N.: Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 622–629 (2014)
32. Pearce, O., Gamblin, T., de Supinski, B.R., Schulz, M., Amato, N.M.: Quantifying the effectiveness of load balance algorithms. In: Proceedings of the 26th ACM International Conference on Supercomputing (2012)
33. Poovey, J., Railing, B., Conte, T.: Parallel pattern detection for architectural improvements. In: Proceedings of the 3rd USENIX Conference Hot Topic Parallelism (2011)
34. Rodrigues, G.S., Kastensmidt, F.L., Reis, R., Rosa, F., Ost, L.: Analyzing the impact of using pthreads versus openmp under fault injection in arm cortex-a9 dual-core. In: 16th European Conference on Radiation and Its Effects on Components and Systems (RADECS) (2016)
35. Rosa, F.d., Bandeira, V., Reis, R., Ost, L.: Extensive evaluation of programming models and isas impact on multicore soft error reliability. In: Proceedings of the 55th Annual Design Automation Conference (DAC) (2018)
36. Snir, M., Wisniewski, R.W., Abraham, J.A., Adve, S.V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., Chien, A.A., Coteus, P., Debardeleben, N.A., Diniz, P.C., Engelmann, C., Erez, M., Fazzari, S., Geist, A., Gupta, R., Johnson, F., Krishnamoorthy, S., Leyffer, S., Liberty, D., Mitra, S., Munson, T., Schreiber, R., Stearley, J., Hensbergen, E.V.: Addressing failures in exascale computing. Int. J. High Perform. Comput. Appl. **28**(2), 129–173 (2014)
37. Sridharan, V., DeBardeleben, N., Blanchard, S., Ferreira, K.B., Stearley, J., Shalf, J., Gurumurthi, S.: Memory errors in modern systems: The good, the bad, and the ugly. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2015)
38. Stratton, J.A., Rodrigues, C., Sung, I.J., Obeid, N., Chang, L.W., Anssari, N., Liu, G.D., mei W. Hwu, W.: Parboil: A revised benchmark suite for scientific and commercial throughput computing. IMPACT Technical Report 12-01, University of Illinois at Urbana-Champaign (2012)
39. Tanikella, K., Koy, Y., Jeyapaul, R., Kyoungwoo Lee, Shrivastava, A.: gemv: A validated toolset for the early exploration of system reliability. In: 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 159–163 (2016)

40. Vishnu, A.V., Dam, H., Tallent, N.R., Kerbyson, D.J., Hoisie, A.: Fault modeling of extreme scale applications using machine learning. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2016)
41. Wei, J., Thomas, A., Li, G., Pattabiraman, K.: Quantifying the accuracy of high-level fault injection techniques for hardware faults. In: International Conference on Dependable Systems and Networks (DSN) (2014)
42. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The splash-2 programs: Characterization and methodological considerations. In: Proceedings of the 22Nd Annual International Symposium on Computer Architecture (ISCA) (1995)
43. Yang, N., Wang, Y.: Predicting the silent data corruption vulnerability of instructions in programs. In: 2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS) (2019)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.