Journal of Parallel and Distributed Computing 152 (2021) 111–124

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

PARIS: Predicting application resilience using machine learning

Luanzheng Guo [a,b,*], Dong Li [b], Ignacio Laguna [c]

[a] Pacific Northwest National Laboratory, United States of America
[b] EECS, University of California Merced, United States of America
[c] CASC, Lawrence Livermore National Laboratory, United States of America

ARTICLE INFO

Article history:
Received 11 March 2020
Received in revised form 3 February 2021
Accepted 8 February 2021
Available online 6 March 2021

Keywords:
HPC fault tolerance
Application resilience prediction
Transient faults
Fault injection
Silent data corruption

ABSTRACT

The traditional method to study application resilience to errors in HPC applications uses fault injection (FI), a time-consuming approach. While analytical models have been built to overcome the inefficiencies of FI, they lack accuracy. In this paper, we present PARIS, a machine-learning method to predict application resilience that avoids the time-consuming process of random FI and provides higher prediction accuracy than analytical models. PARIS captures the implicit relationship between application characteristics and application resilience, which is difficult to capture using most analytical models. We overcome many technical challenges for feature construction, extraction, and selection to use machine learning in our prediction approach. Our evaluation on 16 HPC benchmarks shows that PARIS achieves high prediction accuracy. PARIS is up to 450x faster than random FI (49x on average). Compared to the state-of-the-art analytical model, PARIS is at least 63% better in terms of accuracy and has comparable execution time on average.

Published by Elsevier Inc.

1. Introduction

As high performance computing (HPC) systems increase in scale, they become more susceptible to transient faults [6] due to feature size shrinking, lower voltages, and increasing densities in hardware infrastructures [59]. As a result, scientific applications running at extreme scales apply different resilience methods to tolerate frequent soft errors. Applying these methods to a given application often requires a deep understanding of the resilience of the application.

The common practice to study application resilience to errors in HPC systems is *Fault Injection (FI)* [11,17,62–64]. This approach uses a large amount of random injections, each of which randomly selects an instruction, and then triggers bit flips at the instruction input or output operands during application execution. Statistical results are then used to quantify application resilience.

While FI works in practice and is widely used in resilience studies, a key problem of this approach is that it is highly time consuming. To illustrate the problem, consider an application that runs for 6 hours—a common execution time for a large-scale scientific simulation [26]. Using statistical analysis (e.g., using [45]), the number of random FIs to obtain a low margin of error (e.g., 1%–3%) is in the order of thousands of injections. Thus, the total FI campaign could last several days. For multi-threaded or multi-process applications, this time is significantly higher since random faults must be injected into different threads or processes.

To address the limitations of FI, researchers have built error-propagation analytical models [46], which are faster than FI in estimating application resilience. However, they lack of accuracy as they estimate application resilience to errors based on analysis of possible errors in individual instructions. The analysis inaccuracy at individual instructions is accumulated, causing low accuracy to estimate the whole application resilience. Furthermore, these models do not consider effects of resilience computation patterns (e.g., dead corrupted locations and repeated addition [32]). Studying those patterns demands analyzing multiple instructions together, while most existing analytical models analyze instructions in isolation.

In summary, the community lacks a fundamental approach that enables fast and accurate evaluation of application resilience. In this paper, we present a novel framework called PARIS,[1] which avoids the time-consuming process of randomly selecting and executing many injections (as in FI), and provides higher prediction accuracy than analytical models, making it a unique solution to the problem. In essence, PARIS uses a machine learning model to predict application resilience, which provides several advantages. *First*, machine learning models, once trained, can be repeatedly

* Corresponding author at: Pacific Northwest National Laboratory, United States of America.
E-mail addresses: lenny.guo@pnnl.gov (L. Guo), dli35@ucmerced.edu (D. Li), ilaguna@llnl.gov (I. Laguna).

https://doi.org/10.1016/j.jpdc.2021.02.015
0743-7315/Published by Elsevier Inc.

[1] PARIS: **P**redicting **A**pplication **ReS**ilence.

used for any fault manifestations – silent data corruption (SDC), interruptions, and success cases – for new, previously unseen applications. Therefore, PARIS avoids a large amount of repeated fault injection tests, which leads to high efficiency in comparison to FI. *Second*, machine learning models can capture the implicit relationship between application characteristics (e.g., intensity of resilience computation patterns) and application resilience, which is difficult to capture by analytical models.

The most challenging part of using the machine learning approach is to build effective features. We use the following methods to construct a feature vector of 30 features.

First, we count the number of instruction instances within each instruction type as a feature; instruction instances are dynamic execution of instructions. We characterize instructions in such a way because different instruction types show different resilience to errors [12,37]. To reduce the number of features, we classify instruction types into four representative and discriminative groups in terms of the functionality of instructions. This reduction of features reduces the training complexity and avoids undertraining.

Second, we count resilience computation patterns as features. Guo et al. [32] discover six resilience computation patterns from HPC applications. Those patterns are considered the fundamental reason for application resilience. Four of those patterns are based on individual instructions, and can be included as features using the above instruction type-based approach. The remaining two ("dead locations" and "repeated addition") contain more than one instruction and cannot be captured by examining instructions individually. To efficiently count the two patterns, we introduce optimization techniques to avoid repeatedly scanning the instruction trace and find correlation between instructions.

Third, we introduce instruction execution order information into features to improve modeling accuracy. Execution order information is important to application resilience, because error propagation is highly correlated to the order and type of operations. Inspired by "N-gram" technique [16,56] in computational linguistics, we embedded the sequence of instruction chunks into features to introduce execution order of instructions. Our evaluation shows that having execution order information decreases prediction error by up to 30%.

Fourth, we introduce *resilience weight* when counting instruction instances. Different instruction instances, even though they have the same instruction type, can have different capabilities to tolerate faults. Resilience weight quantifies the resilience difference of those instruction instances. Introducing resilience weight decreases prediction error by 13% on average when predicting the rate of some fault manifestation (particularly, the interruption rate).

Based upon the above features, we use feature selection techniques to sort and further reduce features. We perform ablation study to understand the sensitivity of features to prediction accuracy. We reveal significance of memory-related instructions and data overwriting to application resilience.

In summary, our contributions are as follows.

- We present PARIS, a machine learning-based approach to predict application resilience. Our method breaks the fundamental tradeoff between evaluation speed and accuracy in the existing common practice to estimate application resilience.
- We develop a framework and overcome a series of technical challenges for feature construction, extraction and selection. We reveal how to use machine learning to effectively and efficiently model application resilience.
- We test our model on 16 benchmarks. We find that our approach is up to 450x faster than random FI (49x on average). The model has high prediction accuracy: a prediction error of 8.5% and 22% on average for predicting success rate and interruption rate (excluding two obvious outliers) respectively. We compare PARIS with Trident [46] (the state-of-the-art analytical model): PARIS can predict any fault manifestation rate (SDC, interruptions, and success), while Trident only predicts SDC rate; PARIS is at least 63% better than Trident in terms of accuracy for predicting SDC rate, and has comparable execution time (but faster for 12 out of the 16 benchmarks with 15x speedup on average).

2. Background

2.1. Fault model

We consider transient faults in computation units of processors. For example, transient faults in the Arithmetic Logic Unit (ALU) and the address computation for loads and stores. We do not consider transient faults in memory components, such as caches, because these components are usually protected by Error Correcting Code (ECC) or parity at the architecture level. Similar assumptions are made in existing work [46,63].

Furthermore, we consider single bit-flip model, not multiple bit-flip model. Because single bit-flip model is the de-facto fault model commonly adopted by existing work to emulate errors propagated to applications [44,46,63]. Despite transient faults can manifest as single and multiple bit-flips in applications, existing studies have demonstrated that multi-bit errors can have a similar impact on the application as single-bit errors [46]. Therefore, we use single bit-flip model in this paper.

2.2. Fault injection

We use PINFI [63] to perform fault injections into programs. PINFI triggers a single bit-flip into the destination register or memory location of a randomly chosen instruction to emulate the effect of transient faults. The registers or memory locations are chosen as the injection targets by PINFI, because any error in the computation/data paths of the processor shows up in the results of the executed instruction. PINFI's fault model is the same as ours. Comparing with other common FI tools (e.g., LLFI [61] and REFINE [29]), PINFI is very accurate and user-friendly. In our study, the number of FIs is determined by using a statistical approach [45] with the confidence level of 99% and the margin of error 1%.

2.3. Application resilience

We run FI campaigns to measure the application resilience. An FI campaign contains many FIs. In each FI, a single-bit error is injected into an input/output operand of an instruction. We classify the outcome, or *manifestation*, of programs corrupted by bit flips into three classes: success, SDC, and interruption:

- **Success**: the execution outcome is the same as outcome of fault-free runs. The execution outcome can also be different from outcome of fault-free runs, but the execution passes the result verification phase of the application.
- **SDC**: the program outcome is different from the outcome of the fault-free execution, and the execution does not pass the result verification phase of the application.
- **Interruption**: the execution does not reach the end of execution, i.e., it is interrupted in the middle of the execution, because of an exception, crash, or hang.

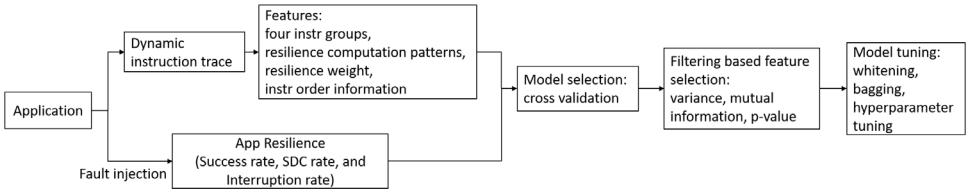


Fig. 1. Overview of PARIS and the workflow of the training process in our ML method.

Rates. To quantify the application resilience in an FI campaign, we measure the rate of each of the three classes of manifestations. In particular, we use the formula:

$$\#\text{Manifestations}/N \tag{1}$$

where #Manifestations is the number of times a given class of manifestation occurs, and N is the number of FIs in an FI campaign. We consider the rates of success, SDC and interruption as metrics to quantify application resilience. The rates are real numbers between 0.0 and 1.0. Since they are mutually exclusive, their addition for a given application is 1.0.

2.4. Machine learning model

Training and Testing Phases. The modeling process of ML includes training and testing. We use a set of representative applications to train the model—once it is trained, the model is used to predict, or test, the manifestation rates on new applications. We call the applications used for training and testing the training dataset and the testing dataset, respectively.

Prediction Accuracy. To evaluate the trained model, we compare the Mean Absolute Percentage Error (MAPE) [21] of the predicted application resilience against the ground-truth application resilience measured by performing FI. Eq. (2) gives the definition of MAPE. MAPE is often used for regression model evaluations because it can interpret modeling accuracy in terms of *relative errors* [21]. A low MAPE means a better accuracy. The lowest MAPE is zero.

$$MAPE = |\frac{Measured - Predicted}{Measured}|. \tag{2}$$

3. Overview

We give a high-level overview of PARIS. Fig. 1 depicts the workflow of the training process of PARIS. The most challenging part of the training process is to construct features relevant to application resilience that can produce high modeling accuracy.

Features Construction. We use instruction type and number of instruction instances for each type as a feature. A static instruction in a program has an instruction type (opcode), and can be executed many times, each of which is an *instruction instance*. Using the number of instruction instances for each instruction type as a feature will result in too many features, which demands a large training dataset. To reduce the number of features, we group all instruction types (65 in total) into four groups: control flow instructions, floating point instructions, integer instructions, and memory-related instructions. For each instruction group, we count the number of instruction instances as a feature.

Furthermore, we use six resilience computation patterns proposed in [32] as features. Among the six patterns, four of them (conditional statement, shifting, data truncation, and data overwriting) are individual instructions that are not grouped into the four instruction groups, because of the significance of these instructions to application resilience. Two of them (dead corrupted locations and repeated additions) include multiple instructions, where these instructions all together contribute to application resilience.

Counting dead corrupted locations and repeated additions from the dynamic instruction trace as features is challenging, because we must repeatedly search within the trace to find correlation between instructions. To detect dead corrupted locations, we use a technique that caches intermediate results of trace analysis to avoid repeated trace scanning. To detect repeated additions, we build a data dependency graph for addition instructions. Such graph enables easy detection of repeated additions.

Because different instruction instances can have different capabilities to tolerate errors, even though those instruction instances have the same instruction type (or the same resilience computation pattern), we introduce *resilience weight* when counting instruction instances. The resilience weight gives each instruction instance a weight quantifying the possible number of single-bit errors tolerable by the instance.

Furthermore, we introduce Instruction Execution Order (IEO) information as a feature. We demonstrate that a small change in IEO can affect the application resilience using an example illustrated in Fig. 3 and described in Section 4.2. However, representing the execution order information of all instruction instances as a feature is a challenge. We use N-gram [16,56], a technique commonly used for processing speech data, to capture the order information.

Putting all together, we manage to build a feature vector of 30 features.

4. Design

4.1. Feature construction

For feature construction, we have the following requirements: (1) features should be relevant to application resilience; (2) the number of features should be small enough (smaller than the number of applications used for training) to avoid underdetermination of the model; (3) we should avoid redundant and irrelevant features since these features can increase prediction error. Following the above requirements, we introduce instructions, resilience computation patterns, resilience weight, and instruction execution order as features. We describe why and how we collect these features in following subsections.

4.1.1. Instruction groups

The primary features are instruction types and number of instruction instances in each type. These features are highly relevant to application resilience. For example, recent studies [46,51] reveal that floating point instructions are highly related to resilience because the faults in mantissa bits of floating-point numbers can be negligible by the application (especially HPC applications). Load/store instructions also have a significant impact on application resilience, because computations following load/store instructions can take those loaded/stored values.

We use the following method to construct instruction-based features. We use LLVM-Tracer [58], an LLVM pass to compile the

Table 1
Four groups of instruction types and four resilience computation patterns as features to build our ML model.

Group name	Instruction types
Control Flow Instructions (CFI)	Br, Indirectbr, Select, PHI, Fence, DMAFence, Call
Floating Point Instructions (FPI)	Fadd, Fsub, Fmul, Fdiv, Frem, Cosine, Sine
Integer Instructions (II)	add, sub, mul, Udiv, Sdiv, Urem, Srem
Memory-related Instructions (MI)	Load, Store, DMAStore, DMALoad, Getelementptr, ExtractElement, InsertElement, ExtractValue, InsertValue, FPToUI, FPToSI, UIToFP, SIToFP, PtrToInt, IntToPtr, AddrSpaceCast
Pattern name	Instruction types
Conditional Statements	ICmp, FCmp, Switch, And, Or, Xor
Shifting	Shl, LShl, AShl
Data Truncation	Trunc, ZExt, Sext, FPTrunc, BitCast, FPExt
Data Overwriting (DO)	All instructions having at least one output operand

application and generate a dynamic LLVM instruction trace. The LLVM instructions are architecture independent, allowing us to build a more general and reusable model. We enumerate all LLVM IR instructions and get 65 instruction types.

We could add all 65 instruction types as features. However, this significantly increases the number of features. With the introduction of IEO as features (see Section 4.2 for why and how we introduce IEO into features), the number of features will be more than 195, larger than the number of training samples, which makes the training under-determined.

To address the above problem, we group 65 instruction types into four groups following heuristics and findings in recent studies [12,31]. In particular, our grouping is based on instructions functionality, which relies on instruction types; different instruction types can have different impacts on application resilience [12,31]. For example, we group control flow related instructions (e.g., `Br` and `Select`) into a group. Table 1 lists the four groups, including control flow instructions, floating point instructions, integer instructions, and memory-related instructions. For each instruction group, we count the number of instruction instances from the dynamic instruction trace, and then normalize the number by the total number of instruction instances. We use the normalized number as a feature to make the feature value independent of the size of the dynamic instruction trace. This enables us to fairly compare application resilience of applications with different trace sizes.

4.1.2. Using resilience computation patterns as features

Recent work [32] finds six resilience computation patterns (dead corrupted locations, repeated additions, conditional statements, shifting, data truncation, and data overwriting) the fundamental reason for application resilience. A resilience computation pattern is defined as a combination of computations that affect application resilience. The reason we introduce dead corrupted locations and repeated additions as features is that the two patterns are composed of multiple instructions that together contribute to application resilience [32]. The other four patterns (conditional statements, shifting, data truncation, and data overwriting) are individual instructions shown in Table 1. We use them separately as features because of their especial significance to application resilience [32].

To count the six patterns as features, we cannot use the method in [32], because it tracks error propagation after fault injection and leverages error masking to discover *unknown* patterns, whereas they do not provide a method to count patterns from the application. We must propose our own method to count resilience patterns from applications to construct features. To efficiently count patterns, we must address below challenges.

First, counting the number of pattern instances[2] for dead corrupted locations and repeated additions is time-consuming, because we must find correlations between instructions to determine if the location is dead or if addition repeatedly happens to the same variable. Doing so requires repeatedly scanning dynamic instruction trace. We discuss how to efficiently count pattern instances for the two patterns in Sections 4.1.3 and 4.1.4, respectively.

Second, for the patterns that are represented as individual instructions (see the last four rows in Table 1 for these instructions), simply counting the number of pattern instances cannot discriminate resilience capabilities of different pattern instances. For example, the resilience capability of the "shifting" pattern (a pattern involving a *shift* instruction) depends on how many bits are shifted. A *shift* instruction instance shifting three bits can tolerate three single-bit errors, while a *shift* instruction instance shifting one bit can only tolerate one single-bit error. To distinguish fault tolerance capabilities of different instruction instances, we introduce weights (named *resilience weight*) when counting instances of the patterns.

Besides introducing weights for the four patterns, we also introduce weights to instructions of instruction groups whose instances can also have different fault tolerance capabilities. We describe the relevant details in Section 4.1.5.

4.1.3. Extracting the feature of dead corrupted locations

A combination of operations (e.g., additions and multiplications) aggregates the values of corrupted input locations into fewer output locations. Meanwhile, many of these corrupted input locations are not used anymore (they become dead corrupted locations), which leads the total number of corrupted locations to decrease. A code region with a higher percentage of locations that are dead corrupted locations has higher resilience.

To efficiently detect dead corrupted locations and calculate the percentage of dead corrupted locations, we split the dynamic instruction trace into chunks and pre-process the chunks before detecting dead corrupted locations. A chunk of instructions is the dynamic instruction trace of a first-level inner loop or the code region between two neighbor first-level inner loops. During the trace pre-processing, we analyze instructions in each chunk and save locations of each chunk into an array. To determine if a location in a chunk is dead, we check whether the location is further used in any future chunks by examining the sequence of arrays. If the location is not used in any future chunks, then the location is a dead corrupted location. In essence, the arrays for chunks save instruction analysis results to avoid repeatedly scanning the trace. For each chunk, we compute the percentage of locations that are dead corrupted locations for the chunk. We use the average percentage of dead corrupted locations across all chunks (named "dead corrupted location rate" or DLR) as a feature.

4.1.4. Extracting the feature of repeated additions

Repeated additions (RA) refers to the addition operations repeatedly happening to the same variable, such that the corruption in the variable can be amortized. To decide if an addition instruction is part of repeated additions, we must first decide if the addition instruction is involved in a self addition. The self addition

[2] A pattern is repeatedly executed in application execution. We name the dynamic execution of a specific pattern the *pattern instance*.

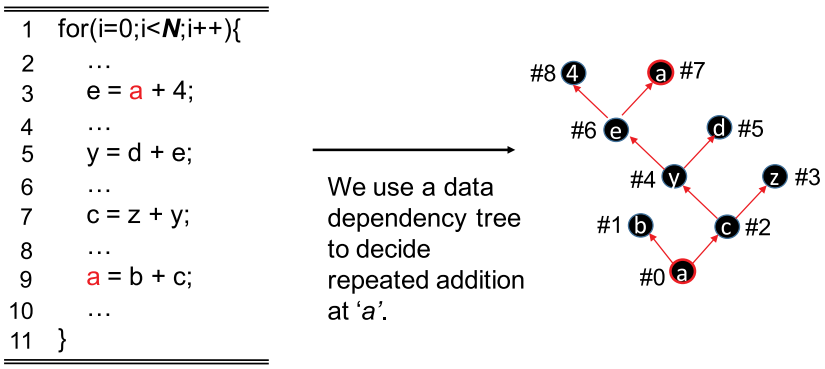


Fig. 2. An example to detect repeated additions.

is defined as that a location adds other locations to itself. The pseudo code in Fig. 2 is an example of self addition.

To detect a self addition, we first build a data dependency graph for addition operations, where nodes are locations; edges between nodes represent data dependency. When given an addition instruction, we examine its output operand and decide if the location (the output operand) is an input operand of a previous addition operation by backward traversing the graph.

Fig. 2 illustrates what a data dependency graph looks like and how a self addition is found. We have four addition statements (operations) in a *for* loop. The location a appears as the output of the last addition statement ($a = b + c$ in Line 9). To determine if the addition statement is involved in a self addition, we find the node 0 corresponding to a in the data dependency graph. We traverse the graph backward, and find a appears in a previous node, the node 7. The node 7 corresponds to a source operand of a previous addition statement ($e = a + 4$). Doing so, a self addition is detected. A pattern of repeated additions is composed of multiple self additions.

To use repeated additions as a feature, we normalize the number of repeated additions by total number of instruction instances. This makes the feature value independent of the size of the dynamic instruction trace.

4.1.5. Resilience weight

Given an instruction, all bit locations of its input and output operands are subject to error corruption. The resilience weight ($\mathcal{R}es$) of an instruction is defined below.

$$\mathcal{R}es = \frac{\#bit\ locations\ that\ tolerate\ errors}{\#of\ all\ bit\ locations} \tag{3}$$

Using the *right-shift* instruction as an example. The instruction has three 8-bit operands and in total 24 locations. Assume that an instance of the instruction shifts four least significant bits of an operand. The shifted four bits can tolerate four single-bit errors. Also, the eight bits in the output operand of the instruction can tolerate errors because of the result overwriting in the output operand. Hence, in this example, the resilience weight for this instruction instance is (4+8)/24 = 0.5. Consequently, the bit locations that can constantly tolerate errors are bit locations of the output operands, because we expect errors in the output operands to be overwritten. Notably, we use the weight in case counting the number of instruction instances or the number of pattern instances.

As a result of feature construction, we construct a feature vector of ten features, formulated in Eq. (4), where "DLR" and "RA" are the dead corrupted locations and repeated additions, respectively. Notations for the equation can be found in Table 1.

$$\mathcal{F}_{10}^{ave} = [CFI, FPI, II, MI, Condition, Shift, Truncation, DO, DLR, RA] \tag{4}$$

We call $\mathcal{F}_{10}^{ave}$ the *foundation feature vector* and consistently call the ten features *foundation features* in the rest of the paper.

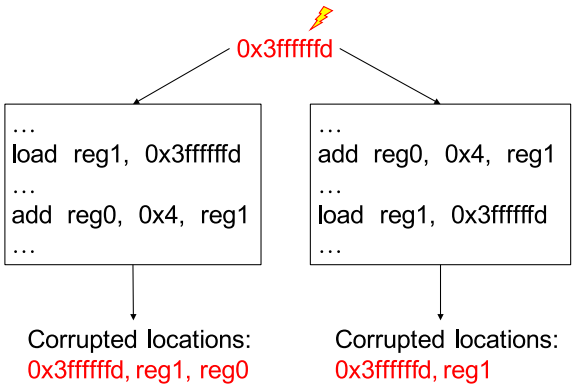


Fig. 3. An example to show that the instruction execution order matters to error propagation.

4.2. Introducing instruction execution order (IEO)

The foundation features are not good enough to achieve high prediction accuracy. In particular, the foundation features lack IEO information. Capturing the IEO is important because it matters in error propagation.

We use an example shown in Fig. 3 to depict why IEO matters. In this example, we have a *load* instruction and an *addition* instruction. Assume that an error happens in a memory address 0x3fffffffd. If the *load* instruction happens first, then the erroneous value in the memory address propagates to the locations *reg*1 and *reg*0. But if the *addition* instruction happens first, then the erroneous value in the memory address only propagates to the location *reg*1. This example is a demonstration of how IEO matters to error propagation.

To introduce IEO into the feature vector, we use the "N-gram" technique [16]. The N-gram is a technique used in computational linguistics. It can work on a sequence of streaming words, and predict the next word using sequences of previous words. N-gram can capture the word order information. Particularly, every n continuous words compose an n-gram ($n = 1, 2, 3, \ldots$). Fig. 4 depicts how we build the feature vector with IEO included. Particularly, we partition the dynamic instruction trace into chunks (each chunk is a gram). Each chunk is regarded as a

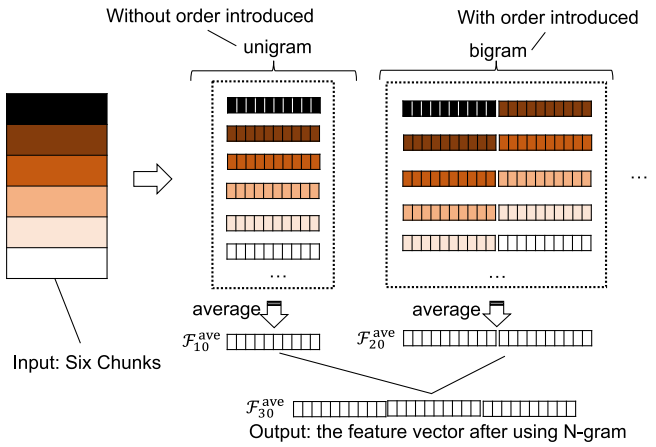


Fig. 4. Applying the N-gram technique to introduce instruction execution order information.

"word", and the sequence of chunks is processed as the sequence of "words". For each chunk, we collect the ten foundation features and build a foundation feature vector of size ten. Then, we build an average foundation feature vector (denoted as $\mathcal{F}_{10}^{ave}$) which is the average of foundation feature vectors of all chunks.

Furthermore, we combine every two neighboring chunks to build a bigram (n=2 for n-gram). Particularly, we concatenate two foundation feature vectors to build a bigram feature vector of size 20. We then build an average bigram feature vector (denoted as $\mathcal{F}_{20}^{ave}$) which is the average of all bigram vectors.

Putting All Together. We manage to build a feature vector of 30 features. We have $\mathcal{F}_{10}^{ave}$ of size 10 and $\mathcal{F}_{20}^{ave}$ of size 20. The final feature vector $\mathcal{F}_{30}^{ave}$ is the combination of $\mathcal{F}_{10}^{ave}$ and $\mathcal{F}_{20}^{ave}$. The final feature vector has 30 features.

We do not consider trigram (i.e., 3-gram) or higher gram, because existing research [16] demonstrate that there is no need to use higher grams than bigram. In [16], bigram achieves better accuracy than trigram while using trigram or higher grams does not provide better prediction accuracy but dramatically increases feature vector size and complexity of model training.

4.3. Feature selection

Following the requirement of feature construction, we aim to eliminate irrelevant and redundant features and further reduce the feature vector size. We use three filtering-based methods to select features. Compared to other feature selection methods such as wrappers and embedded methods, the filtering-based methods are faster because of their simplicity and low complexity. In addition, the filtering-based methods are independent of the prediction model [33]. In such a way, the selected features can be used with different prediction models.

We use the following filtering-based methods to select features: the p-value-based method [9], the mutual information-based method [5], and the method of calculating variance [33]. Simply speaking, the p-value is a metric that measures the significance level between a feature and the modeling result (i.e., the success, SDC, or interruption rate). The mutual information measures the mutual dependency between a feature and the modeling result. The variance measures the variance of feature values across different input applications. Using each of the three methods, we can rank features into a sorted list according to the importance of features with respect to application resilience. In total, we have three lists.

Using a voting strategy, we combine the three sorted lists of features into one list for feature selection. This voting strategy and feature selection algorithm are common in ML [67]. In particular, each feature has an index in each of the three lists. For each feature, we add its three indexes to get a global index. We sort the features based on global indexes into a single list.

We then decide how many features we want to use to construct the feature vector for modeling. Based on the sorted features in the single list, we choose the best k (where $k = 2, 3, \ldots, 30$) features to build a sublist of features. In total, we have 29 sublists. We choose the features in the best sublist (in terms of the prediction accuracy) as the final features.

4.4. Model construction

Model Selection.

Prediction of application resilience is naturally a regression problem. More formally, we aim to find a model $f()$, such that given an feature vector v corresponding to an application A, $f(v)$ outputs the rates of SDC, interruption, and success for A.

There are tens of regression models. Each of them has pros and cons, and should fit into different scenarios. We use scikit-learn [55] and test all regressions models in scikit-learn (18 in total), such as Gradient Boosting Regression [66], Random Forest Regression [48], and Multi-Layer Perceptron Regression [18]. Cross-validation (CV) [42] is the standard practice used for model selection. We use cross-validation (CV) to test the 18 regression models on the training dataset to select a regression model with the best prediction accuracy. CV partitions the dataset into p folds. q of p folds are used for training, while the remaining $p-q$ folds are used for testing. There are $p/(p-q)$ rounds of training/testing. In each round, different $p-q$ folds are used for testing. We choose the regression model that has the lowest prediction error on average. We use 10-fold cross validation in our study. Based on the CV results, we choose the *Gradient Boosting Regression* to predict application resilience.

Unlike other regression models, Gradient Boosting Regression ensembles many weak learners into a strong learner in an iterative way. It is not surprising that Gradient Boosting Regression achieves the lowest prediction error among all regression models. Gradient Boosting models have demonstrated excellent performance on problems with a relatively small training dataset theoretically and practically [10,18,25,27,50]. There are about 100 computation kernels in our dataset, which is tiny compared to large datasets like ImageNet [23]. Note that neural network models, such as Multi-Layer Perceptron Regression, are not workable for our case. These models can have tens of thousands of parameters to resolve. Small training datasets cannot address that many parameters.

Model Tuning. We use the following techniques to tune the model for better prediction accuracy. (1) Whitening [19]. Whitening is used to normalize features to avoid domination effects of any features for better generalization and to improve the modeling accuracy. (2) Bagging (model averaging) [24], which is often used for reducing variation in training data. We use this technique to eliminate the effect of bad outliers. (3) Hyperparameters tuning. Each regression model has multiple hyperparameters. We use "grid-search" [7] to decide the values of hyperparameters for training.

5. Implementation

Dataset Construction. We have multiple requirements for creating training and testing dataset. (1) The training dataset must be large to avoid model underdetermination; (2) Applications used to generate training and testing dataset must have diverse computation and diverse resilience characteristics; (3) Applications used to generate training and testing dataset

must have explicit result verification phases. Having the verification phase allows us to determine the fault manifestations.

Testing Dataset. We use representative benchmark suites and scientific applications to create the testing dataset, including NAS parallel benchmark suite [4], PARSEC benchmark suite [8], CORAL benchmark suite [1], Rodinia benchmark suite [15], SPEC CPU2000 [36], and two scientific applications (Hercules for earthquake simulation [2] and PuReMD for reactive molecular dynamics simulation [60]). From these resources, we choose 16 applications for testing because of their diverse characteristics. The 16 applications are shown in Table 2. We call the 16 applications *big benchmarks* in the rest of the paper.

Training Dataset. To train PARIS, we use 100 common computation kernels obtained from HackerRank [34]. These kernels are smaller than the big benchmarks, but these kernels all have explicit verification phases. With these kernels, the ranges of modeling output during training are [0.126; 0.982], [0.000; 0.656], and [0.018; 0.874], for the rates of success, SDC, and interruption, respectively; The average values of modeling output during training are 0.502, 0.155, and 0.348 with a variance of 0.033, 0.019, and 0.021 for the rates of success, SDC, and interruption, respectively. The above numbers show that our training is sufficient with these kernels. Also, using the 100 computation kernels is adequate for training because the training is determined when the size of training dataset (100) is larger than the number of features (30).

We require the training and testing datasets to be exclusive to address the representation and generalization of our prediction model on new, unseen applications.

Trace Generation. We use LLVM-Tracer [58], a tool to generate dynamic LLVM IR traces based on LLVM instrumentation. The trace includes LLVM IR instructions and their operands. We extend LLVM-tracer to generate a subtrace for each chunk of instructions and generate traces for MPI programs.

Whitening. We use the whitening technique [19] to normalize features to avoid domination effects of any features for better generalization and to improve the modeling accuracy.

6. Evaluation

We evaluate our model and modeling methods from two perspectives: (1) modeling accuracy; (2) contributions of features and optimization techniques to modeling accuracy.

To evaluate modeling accuracy, we calculate MAPE for the predicted success, SDC, and interruption rates compared with the ground-truth rates measured by performing FI campaigns. We use PINFI [63] for FI. For each program in our dataset, we perform an FI campaign of 3000 random fault injections, following the statistical principles in [45] with the confidence of 99% and margin of error 1%. Furthermore, we compare our modeling accuracy with the modeling accuracy of `Trident`, the state-of-the-art analytical model predicting the SDC rate.

To study contributions of features and optimization techniques to modeling accuracy, we perform a feature selection study to understand the importance of features, and an ablation study to understand impact of each optimization technique on accuracy.

Notably, we do not directly predict the SDC rate, because the value of SDC rates can be zero for small computation kernels, in which any variation when predicting the SDC rate can cause unreasonable MAPE of infinite values when the denominator in the MAPE Equation is zero. Instead, we use the trained model to predict the rate of success and interruption (two classes of fault manifestation). We then calculate the SDC rate by subtracting the rates of success and interruption from one ("1"). Hence, Table 3, Figs. 6 and 7 do not have results for SDC.

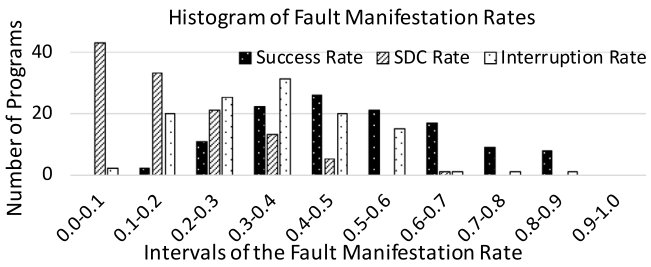


Fig. 5. Histogram of the three fault manifestation rates.

We notice that using the above approach to predict the SDC rate can cause a negative SDC rate. This is because we predict success and interruption rates independently, and there is a chance that the sum of predicted success and interruption rates is larger than one ("1"). For such cases, we force the value of the SDC rate to be zero. Also, we normalize the three rates by their sum in case the sum of the three rates is larger than one.

Artifact Description. We conduct experiments on compute nodes each equipped with Intel(R) Xeon(R) CPU E5-2630 v3 and Ubuntu-14.04.5. Each compute node has Clang-v3.4, OpenMP-v4.0, and scikit-learn installed.

6.1. Prediction accuracy

Table 2 shows the prediction results. Using the results of traditional fault injection as ground truth, MAPE for success rate and SDC rate are 8% and 45%, respectively. Our prediction accuracy for success rate is overall good, but our prediction accuracy for SDC rate is relatively low, but better than the state-of-the-art (see the following discussion in "Comparison with the state-of-the-art for predicting SDC rate"). Predicting SDC rate is challenging because SDC rate can be very small or even zero. A small deviation from the ground truth can cause a large prediction error to MAPE.

To support the statement that the SDC rate tends to be small, we study 116 programs from training and testing datasets. We perform random fault injection and count the histogram (shown in Fig. 5) of the three fault manifestation rates of these programs. Fig. 5 shows that there are more than 65% of programs whose SDC rates are distributed in the range of 0.0–0.2, while values of success rate and interruption rate are distributed in a greater range. We further find that 40% of the programs have the SDC rate less than 0.1.

Comparison with the State-of-the-Art for Predicting the SDC Rate. We compare PARIS with `Trident` [46], a recent work that uses analytical models to estimate the SDC rate. We use `Trident` downloaded from their GitHub website (commit #90b38ab) to estimate the SDC rate for the 16 big benchmarks. The 16 benchmarks include all the benchmarks used in Trident; the number of benchmarks used in Trident is 11. For the 11 benchmarks, we use the same input as in [46]. Table 2 shows the prediction error of `Trident` in the fourth last column.

Table 2 shows that the MAPE of PARIS for SDC rate is 45%, while the MAPE of `Trident` for SDC rate is 680%. We notice that there are two outliers (MG and PuReMD) that make the average prediction error of `Trident` very large. To make the comparison fair, we remove the two outliers. After that, the new MAPE of `Trident` is 108%, which is still worse than the prediction of PARIS. We conclude that PARIS is better than `Trident` in terms of the prediction accuracy on SDC.

Notably, Li et al. [46] report Mean Absolute Error (MAE), which is different from MAPE we report. When evaluating the SDC rate, MAE may not be as appropriate as MAPE. A small MAE (e.g., 0.01) can cause a large MAPE. MAPE measures the relative error. When

Table 2
The detailed prediction results for 16 big benchmarks. Notation: SR = Success Rate; SDCR = SDC Rate; IR = Interruption Rate; Pred.=Prediction; Meas. = Measured.

(a) Prediction results for success rate

Big benchmarks	Suite	Program input	Meas. SR	Pred. SR	Relative error for SR
IS	NAS	Class S	0.653	0.625	4.23%
Nn	Rodinia	filelist_4 5 30 90	0.980	0.910	7.16%
Myocyte	Rodinia	100 1 0 4	0.741	0.764	3.11%
MG	NAS	Class S	0.781	0.721	7.75%
Kmeans	Rodinia	100	0.843	0.749	11.12%
Libquantum	SPEC	33 5	0.863	0.879	1.85%
Blackscholes	PARSEC	in_4.txt	0.663	0.591	10.81%
Sad	Parboil	reference.bin frame.bin	0.475	0.506	6.53%
Bfs-parboil	Parboil	graph_input.dat	0.960	0.906	5.61%
Hercules	CMU	scan simple_case.e	0.580	0.646	11.36%
PuReMD	Purdue Univ.	geo ffield control	0.420	0.438	4.26%
Lulesh	CORAL	-s 1 -p	0.634	0.441	30.44%
Hotspot	Rodinia	64 64 1 1 temp_64 power_64	0.714	0.752	5.30%
Bfs-rodinia	Rodinia	graph4096.txt	0.655	0.674	2.92%
Nw	Rodinia	2048 10 1	0.664	0.647	2.49%
Pathfinder	Rodinia	1000 10	0.623	0.759	21.89%
MAPE	N/A	N/A	N/A	N/A	**8.55%**

(b) Prediction results for SDC rate

Big benchmarks	Meas. SDCR	Pred. SDCR	Relative error for SDCR	Relative error for SDCR by `Trident`
IS	0.083	0.092	11.14%	192.31%
Nn	0.000	0.000	0.00%	93.39%
Myocyte	0.022	0.025	14.67%	826.67%
MG	0.008	0.010	31.14%	5633.33%
Kmeans	0.045	0.098	117.93%	42.64%
Libquantum	0.034	0.000	100.00%	7.60%
Blackscholes	0.122	0.210	72.05%	12.22%
Sad	0.216	0.318	47.36%	34.95%
Bfs-parboil	0.000	0.000	0.00%	3.32%
Hercules	0.182	0.1822	0.11%	128.19%
PuReMD	0.090	0.018	80.00%	3740.00%
Lulesh	0.120	0.255	112.85%	39.01%
Hotspot	0.121	0.124	2.86%	58.97%
Bfs-rodinia	0.124	0.047	62.10%	31.31%
Nw	0.140	0.193	38.34%	20.96%
Pathfinder	0.080	0.052	35.02%	20.81%
MAPE	N/A	N/A	**45%**	**108% (with outliers removed)**

(c) Prediction results for interruption rate

Big benchmarks	Meas. IR	Pred. IR	Relative error for IR
IS	0.264	0.283	6.97%
Nn	0.02	0.090	350.95%
Myocyte	0.237	0.211	11.07%
MG	0.211	0.269	27.49%
Kmeans	0.112	0.153	36.32%
Libquantum	0.103	0.121	17.51%
Blackscholes	0.215	0.199	7.55%
Sad	0.309	0.176	42.91%
Bfs-parboil	0.040	0.094	134.54%
Hercules	0.238	0.172	27.76%
PuReMD	0.490	0.544	10.93%
Lulesh	0.246	0.304	23.69%
Hotspot	0.165	0.124	25.03%
Bfs-rodinia	0.221	0.279	26.43%
Nw	0.196	0.159	18.94%
Pathfinder	0.279	0.189	32.38%
MAPE	N/A	N/A	**22% (with outliers removed)**

relative variation matters and needs to be considered, MAPE is better than MAE [21].

Even though `PARIS` is better than `Trident` in predicting the SDC rate, `PARIS` shows a high relative error on some benchmarks. For example, the relative prediction error for SDC rate for Kmeans, Libquantum, and Lulesh are 117%, 100%, and 112%, respectively. After examining the prediction results closely, we find that the absolute prediction error for the three benchmarks is 0.053, 0.034, and 0.135, respectively, which is small; the ground truth of the SDC rate for the three benchmarks is 0.045, 0.034, and 0.120, respectively, which is also small and close to zero. Accordingly, although the absolute prediction error for SDC is smaller with `PARIS` comparing to `Trident` (on average 0.041 with `PARIS` vs. 0.063 with `Trident`), the relative prediction error with `PARIS` for SDC can be large but still smaller comparing to `Trident`.

Table 3
Feature voting scores for each dimension of the feature vector $\mathcal{F}_{30}^{ave}$.

(a) Feature voting scores for predicting the success rate.										
Dimension number	4	24	8	28	17	12	14	22	18	27
Sorted voting score (Smaller is better)	20	22	23	24	25	27	29	29	31	32
Dimension number	2	3	23	7	20	16	6	21	13	26
Sorted voting score (Smaller is better)	33	39	39	40	43	45	46	48	50	50
Dimension number	11	1	30	15	5	10	25	29	9	19
Sorted voting score (Smaller is better)	53	54	62	69	70	71	74	74	86	87
(b) Feature voting scores for predicting the interruption rate.										
Dimension number	14	18	4	8	27	24	28	7	30	16
Sorted voting score (Smaller is better)	20	23	24	27	27	32	32	34	37	38
Dimension number	6	17	10	26	12	13	1	3	11	2
Sorted voting score (Smaller is better)	39	40	42	43	46	46	47	47	52	53
Dimension number	21	19	20	23	5	15	22	25	9	29
Sorted voting score (Smaller is better)	53	55	55	56	62	63	69	69	77	87

(c) The application characteristics that each dimension of the feature vector represents. Dimensions larger than 9 have the information of instruction execution order using the N-gram technique.					
Dimension#	1, 11, 21	2, 12, 22	3, 13, 23	4, 14, 24	5, 15, 25
Meaning of dimension#	CFI	FPI	II	MI	Condition
Dimension#	6, 16, 26	7, 17, 27	8, 18, 28	9, 19, 29	10, 20, 30
Meaning of dimension#	Shift	Trunc	DO	DLR	RA

Prediction of the Interruption Rate. The MAPE for predicting the interruption rate is 50%. This prediction error seems relatively high. However, we find two outlier benchmarks, which contribute to the bad prediction accuracy. They are Nn and Bfs_parboil. The MAPE for them are 350% and 134%, respectively. Excluding the two outliers, the new MAPE for predicting interruption rate is 22% which is much acceptable.

After we profile Nn and Bfs_parboil, we find that these codes have a relatively large number of load instructions (19% and 44% of total instructions), which is larger than that in most of the benchmarks we study. Predicting the interruption rate accurately depends on accurately counting load instructions because loading data from an incorrect address often cause segmentation faults (or interruptions). However, we do not accurately count load instructions during feature construction, because load and other memory-related instructions are counted together as an instruction group (see Table 1). Thus using a group (as opposed to a single instruction class) for counting causes low prediction accuracy in this case.

In summary, while the method of using instruction groups as features may cause high prediction error, we use groups to limit the number of features to reduce training time and the necessity of using many training samples. Hence, there is a tradeoff between training efficiency and prediction accuracy.

Discussion. We achieve a high prediction accuracy for predicting success rate in contrast to the prediction on SDC and interruption rates. Our quick (see Section 6.4 for the efficiency study) and accurate prediction on success rate is valuable in practical. For example, when deciding the application-level fault tolerance mechanism for a code, the resilience (or success rate) of the code in the presence of errors is the key concern [13]. When the success rate is high (close to 1), which means the code has a high resilience to errors. In this case, one would use cheap fault tolerance mechanisms rather than expensive ones. Therefore, having an efficient and accurate way to estimate resilience (or success rate) of the code is beneficial for directing fault tolerance mechanisms.

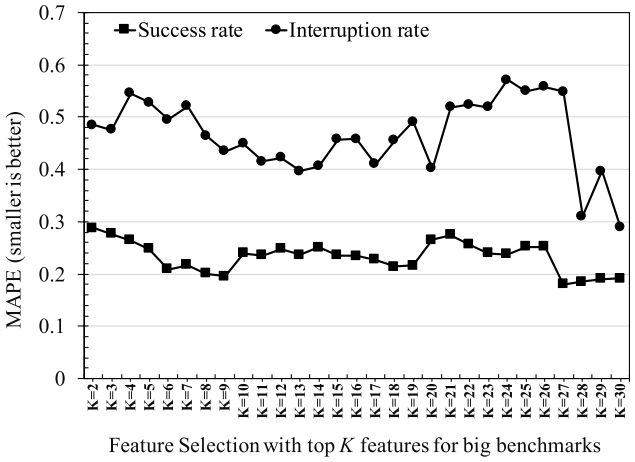


Fig. 6. The ablation study result: the average prediction error for predicting the rates of success and interruption when the best k features are selected (k ranges from 2 to 30).

6.2. Feature selection and analysis

Recalling that we use a voting strategy for feature selection. With the voting strategy, we have a global index for each feature. The global index aggregates voting results of the three feature selection methods (p-value, mutual information, and variance). Table 3 shows the global indexes for all 30 dimensions of the feature vector. The application characteristics that each dimension of the feature vector represents are summarized in Table 3.c. Table 3.a reveals that the 4th dimension (the memory-related instructions), 24th dimension (the memory-related instructions in bigram), and 8th dimension (the pattern of overwriting) in $\mathcal{F}_{30}^{ave}$ rank the highest; Table 3.b reveals that the 14th dimension (the memory-related instructions in bigram), 18th dimension (the pattern of overwriting in bigram), and 4th dimension (the

memory-related instructions) in $\mathcal{F}_{30}^{ave}$ rank the highest. Those dimensions are memory-related instructions, which seem to matter most to the application resilience.

In addition, both tables reveal that the 9th dimension (i.e., the pattern of dead location), 19th dimension (i.e., the pattern of dead location in bigram), and 29th dimension (i.e., the pattern of dead location in bigram) rank relatively low. This result indicates that dead location seems to have less impact to application resilience than the other features.

Ablation study. In this study, we show the effect of using the best k features to make a prediction ($k = 2, 3, \ldots, 30$) to prediction accuracy. This study can also help us understand the contributions of each feature to prediction accuracy. Fig. 6 shows the result of the ablation study. The figure shows the prediction error for the rates of success and interruption.

In Fig. 6, the prediction error decreases by 17% (from 0.3 to 0.25) for predicting the success rate when adding MI-related features (4th and 24th dimensions in $\mathcal{F}_{30}^{ave}$) and data overwriting related features (8th and 28th dimensions in $\mathcal{F}_{30}^{ave}$). Moreover, the prediction error decreases another 24% (from 0.25 to 0.19) after adding truncation in bigram (17th dimension) into features. We then conclude that MI-related instructions, data overwriting-related instructions and truncation have a significant impact on application resilience in terms of the success rate. This finding is consistent with our findings for feature voting scores for predicting the success rate in Table 3.a.

When predicting the interruption rate, the prediction error decreases by 20% (from 0.5 to 0.4), when adding the 2nd dimension in $\mathcal{F}_{30}^{ave}$ to features. The 2nd dimension is the floating point instructions. When k is 28, the prediction error decreases 45% (from 0.55 to 0.3) when adding the 25th dimension in $\mathcal{F}_{30}^{ave}$ to features. The 25th dimension is the conditional statement in bigram. This suggests that floating point instructions and conditional statement significantly affect application resilience in terms of interruption.

On the other hand, we see an increase of MAPE after adding a new feature to the feature vector. For example, after adding the 23rd dimension in $\mathcal{F}_{30}^{ave}$ to features when k is 24 for predicting the interruption rate, the MAPE of interruption rate goes up to 0.57 from 0.51. However, this does not necessarily mean that this feature plays a less important role to predict application error resilience. This feature together with the successive features can make a significant contribution to application resilience with respect to interruption. For example, we can see a significant decrease in MAPE when k is 28 for predicting the interruption rate (the MAPE decreases to 0.31 from 0.55). Lacking this feature, we may not achieve such a big decrease in MAPE when k is 28.

In Fig. 6, we notice that the MAPE value is the lowest for both success rate and interruption rate when k is 30. At this point, MAPE for predicting the success and interruption rates are 0.19 and 0.28, respectively. In consequence, we choose k equal to 30 for both the success and interruption rates. We also notice that the MAPE values when k is 30 in Fig. 6 are different from those in Table 2. The reason is as follows. The MAPE when k is 30 in Fig. 6 is the result of feature selection before applying the two model tuning techniques: hyperparameter tuning and bagging. However, the MAPE in Table 2 is the final result after applying all model tuning techniques and feature construction optimizations. Therefore, the MAPE values in Table 2 are smaller than those when k is 30 in Fig. 6. Also note that the two results in Table 2 (0.08 for success rate and 0.22 for interruption rate) are consistent with the results in Fig. 7.

Table 4
The margin of error (%) for different numbers of fault injections.

#FIs	Success	SDC	Interruption
300	5.6%	3.3%	5.1%
1000	3.2%	2.0%	2.9%
2000	2.2%	1.6%	2.2%
3000	1.3%	1.1%	1.2%

6.3. Evaluation of model tuning and feature construction optimization

We study the impact of our model tuning (whitening, bagging and tuning hyperparameters) and feature construction techniques (bigram and resilience weight) to prediction accuracy. We use 100 small computation kernels (for training) for our study. We start with the model without using any of the five techniques, and then apply them one by one in each step.

Fig. 7 shows the results. We can see that the prediction error continues decreasing after we apply all these techniques. Overall, the MAPE of predicting success rate decreases by 71%; the MAPE of predicting interruption rate decreases by 33%. This demonstrates the effectiveness of all the five techniques in predicting application resilience. Among the five techniques, the most effective ones are bigram and bagging for predicting success rate, and resilience weight for predicting interruption rate.

We notice that after introducing bigram, the MAPE decreases by 30% when predicting success rate. Despite the MAPE reduces slightly when predicting interruption rate after introducing bigram, we find that 58% of kernels have lower prediction error, with up to 20% decrease in MAPE. After introducing resilience weight, the MAPE decreases by 12% when predicting success rate and by 13% when predicting interruption rate. We also observe that the MAPE decreases by 33% when predicting success rate after introducing bagging. After considering resilience weight, the MAPE reduces 12.5% when predicting success rate and 13.3% when predicting interruption rate. The above results demonstrate the effectiveness of bigram, resilience weight, and bagging in predicting application resilience.

6.4. Efficiency study—comparing PARIS to random fault injection and *Trident*

We compare the execution time of using FI, using `Trident`, and using `PARIS` to predict the rate of manifestations on the 16 big benchmarks. The number of FIs is determined by using a statistical approach [45] with the confidence level of 99% and the margin of error 1%. The number of FIs is about 3000.

To justify the number of random FIs (3000) that is necessary to guarantee the high-fidelity of the statistical measurement, we re-run the fault injection campaigns, with reduced numbers of random FIs (300, 1000, 2000) in each fault injection campaign, to calculate the margin of error while fixing the confidence level of 99%, following the same statistical approach in [45]. In particular, we calculate the average margin of error by different fault manifestations for each reduced number of random FIs. We summarize the results in Table 4. We observe that the numerical values of the margin of error for the reduced number of fault injections are significantly larger than the margin of error of 3000 random FIs. We conclude that the number of random FIs has to be 3000 or higher to guarantee statistical significance.

We measure the execution time of 3000 random FIs as the execution time of FI for each benchmark. When measuring the execution time of using `PARIS`, we measure the execution time spent on the whole workflow of predicting application resilience for a new, unseen application, including dynamic instruction trace

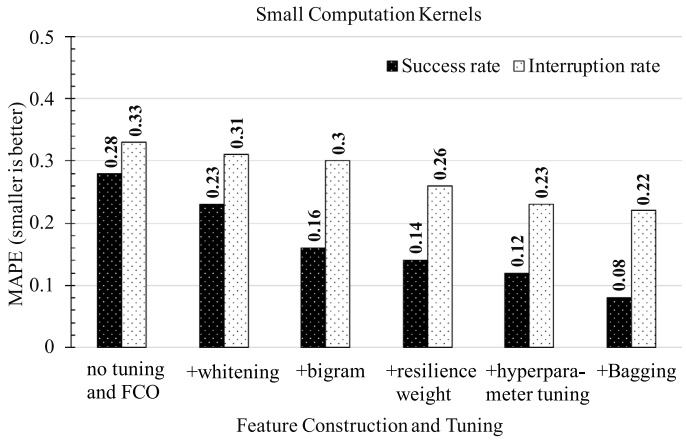


Fig. 7. Evaluating the impact of model tuning and feature construction optimization on the prediction error for the two fault manifestation rates. FCO = "feature construction optimization". In terms of MAPE, Lower is better.

Table 5
The efficiency comparison between FI, `Trident`, and `PARIS`. The table includes breakdown of execution time for the `PARIS` workflow and speedup (using FI as the baseline).

Benchmarks	FI (s)	`Trident` (s)	`PARIS` (s)	Trace generation(s)	Feature construction(s)	Prediction(s)	Speedup over FI
IS	15740	5158	4765	712.3	4052.5	0.3	3x
Nn	8860	4820	395	16.5	378.5	0.3	20x
Myocyte	16380	1215	582	87.2	494.8	0.3	28x
MG	9270	10980	4915	1359.3	3555.9	0.3	2x
Kmeans	4680	1083	234	51.8	182.2	0.3	20x
Libquantum	4714.3	1179	558	0.4	557.6	0.3	8x
Blackscholes	4793	918	23.3	1.1	21.9	0.3	205x
Sad	58890.8	9723	13408	4187.6	9220.4	0.3	4x
Bfs-parboil	11340.4	2835	10450	553.2	9896.8	0.3	1x
Hercules	4703.2	1170	194	7.6	186.4	0.3	24x
PuReMD	1099350	4410	360947	48640.3	312307.2	0.3	3x
Lulesh	9089.3	1896	20.3	1.8	18.2	0.3	450x
Hotspot	43650	15740	10480	3749.7	6730.3	0.3	4x
Bfs-rodinia	36630	10913	15952	6051.7	9900.3	0.3	2x
Nw	16470	4618	4232	859	3373	0.3	4x
Pathfinder	102960	16509	8240	2507	5733	0.3	13x

generation, feature extraction, and making prediction with the trained model. It is important to note that the model training time is not counted into the execution time of the whole workload of predicting application resilience, because once the model is trained, it can be reused repeatedly for an unlimited number of applications, which amortizes the cost of training.

Table 5 shows the results. In general, the speedup of using `PARIS` over using FI is up to 450x (see LULESH) and 49x on average. `PARIS` is faster than FI for all 16 benchmarks. Furthermore, `PARIS` is faster than `Trident` for 12 out of the 16 benchmarks with 15x speedup on average. For the four benchmarks (Sad, Bfs-parboil, PuReMD, and Bfs-rodinia), `PARIS` is slower, due to the time-consuming trace generation.

We further break down the execution time for the workflow of `PARIS` and compute the speedup of using `PARIS` over FI in Table 5. The execution time of FI is in the second column. The execution time of FI can be affected by instruction profiling and the complexity of the FI tool. Furthermore, the time can be significantly affected if the program hangs after FI. The time breakdown of `PARIS` is shown in the third, fourth, and fifth columns. The time spent on making the prediction is constant, which is always around 0.3 s. The time spent on dynamic instruction trace generation changes significantly across benchmarks, which is correlated to input problem size and computation complexity of the benchmark. The time spent on feature extraction varies significantly for different benchmarks, which is affected by instruction trace size and complexity of computations in the application.

7. Discussions

Use of `PARIS`. To use `PARIS`, the user only needs to train the prediction model once, and then the trained model can be repeatedly used for predicting error resilience of any application. Predicting application resilience is useful for improving application resilience [14,32] and optimizing fault tolerance mechanisms [20,39,46,65]. To train the prediction model, the user must follow the training workflow in Fig. 1. Given a new application, the user needs to generate a dynamic instruction trace and feeds it to `PARIS`, and `PARIS` will output three numerical values: the predicted success, SDC, and interruption rates.

Furthermore, `PARIS` can work on different hardware architectures and for parallel applications with different input problems. We discuss these scenarios as follows.

Support for Different Hardware Architectures. `PARIS` currently supports x86 architectures. Because the LLVM IR instructions that we use as features are related to the x86 ISA. Other different ISAs (such as RISC) are expected to have different instructions, where our instruction grouping strategy probably does not fit. We plan to support new architectures that take different ISAs in future work. For that, we need to come up with new feature designs for unseen instructions coming with different ISAs.

Support for Different Input Problems. `PARIS` can work on applications with different input problems. `PARIS` treats it as a new application for the same application taking different input.

Table 6
Comparison between previous work and PARIS.

	Track error propagation?	Discover or count error masking?	Distinguish resilience difference of instructions?	Consider instruction execution order?
Random FI	No	No	No	No
FlipTracker [32]	Yes	Discover	No	No
Trident [46]	Yes	Count	Yes	No
MOARD [31]	Yes	Count	Yes	No
PRISM [39]	No	Count	Yes	No
IPAS [44]	Yes	No	Yes	No
Desh [20] and Nie et al. [52]	No	No	No	No
PARIS	No	Count	Yes	Yes

In this case, users need to generate a dynamic instruction trace for each different input, and PARIS will process the dynamic instruction trace and generate the predicted fault manifestation rates. Unlike the traditional FI, where users have to perform an expensive FI campaign, PARIS is much faster.

Support for Parallel Code. PARIS can work for MPI programs. This is supported by our extension to LLVM-tracer that enables LLVM-tracer to generate a trace for each MPI process. Also, the prediction model in PARIS has to be trained using parallel programs, in order to capture the effects of error propagation across MPI processes. If the user cannot train the prediction model using parallel programs, the user can still use the prediction model to make the prediction for serial programs, and then make the prediction for parallel programs based on recent work [41].

Pros and Cons. PARIS provides an alternative solution to estimate application resilience in addition to random FI and Trident. Random FI is the common practice and often referred to as the ground truth for accuracy. However, random FI performs FI campaigns, which are very time-consuming. Trident is much faster than random FI but introduces errors to application resilience measurement. PARIS is faster than random FI and achieves comparable efficiency and higher prediction accuracy than Trident. Note that we consider training a part of the prediction tool development. We, therefore, do not count the training cost in the efficiency comparison. Furthermore, PARIS can predict any fault manifestation rate (SDC, interruption, and success), while Trident only predicts the SDC rate. For applications that need quick application resilience estimation, PARIS is an alternative, better solution to Trident.

However, PARIS is trained on a small dataset of 116 applications and computation kernels. This affects prediction accuracy significantly. We plan to develop a larger dataset to enrich the representation of the training dataset to further advance prediction accuracy. We also plan to improve the performance of trace generation and feature extraction using parallel programming.

8. Related work

Using Machine Learning to Address Resilience Problems. Recent research starts to use ML to address resilience problems [3,20,28,30,39,44,52]. Laguna et al. [44] train an ML classifier IPAS. IPAS learns which instructions can have a high likelihood of leading to a silent output corruption. Desh [20] predicts node failures by training a recurrent neural network model using system logs. Nie et al. [52] use system logs to predict the future occurrence of GPU errors. PRISM [39] predicts resilience for GPU applications using application properties. However, different from PARIS, PRISM focuses on GPU applications, and PRISM does not consider instructions execution order for feature design.

Random FI. This is the most common method to study application resilience [17,22,40,43,47,49,54]. Typically application-level FI has to be performed many times to ensure statistical significance. Some research prunes unnecessary FIs to reduce FI efforts. Hari et al. [35] and Kaliorakis et al. [38] explore fault equivalence for selective FI by grouping instructions that have the similar effects on program execution at the same static instruction. They further reduce FI positions by leveraging the equivalence of intermediate states in execution and instruction-level approximate computing [57,62]. Although they use instruction grouping, their method is different from ours. They group static instructions at the program level, while we group dynamic instructions based on their functionality and our instruction grouping is independent of the program. Nie et al. [53] prune fault injection sites by only analyzing a subset of threads and a subset of registers that are representative for GPGPU applications. Our work tries to address the inefficiency of using FI to study application resilience by circumventing performing fault injections. But the above existing work is complementary to our work for model training.

Application Resilience Analysis. Application-level error propagation has been widely studied. Li et al. propose Trident [46], a three-level error propagation model, to predict SDC probabilities. MOARD [31] develops an analytical model and a tool to count error masking events on individual data objects. Our work does not trace error propagation but includes an N-gram based technique to embed the instruction execution order information into the feature vector to involve error propagation. FlipTracker [32] discovers six common resilience patterns by tracing error propagation and identifying error masking. Distinctively, our work focuses on detecting and counting these patterns.

We summarize the key differences between previous work and ours in Table 6.

9. Conclusions

Understanding application resilience to errors becomes increasingly important to ensure result correctness for HPC applications. The traditional method (FI) to understand application resilience is too expensive. Analytical models are faster but they are not as accurate as FI. This paper introduces PARIS, a new solution based on ML to solve the above problems. We discuss feature constructions, extraction and selection, which are the keys to enable high-performance ML for predicting application resilience. Using a broad spectrum of benchmarks for evaluation, we show that PARIS is much faster than FI, and provides better accuracy (at least 63% better) than the state-of-the-art analytical model. PARIS provides comparable execution time (on average) than the analytical model, but is faster for 12 out of the 16 evaluated benchmarks.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Karthik Pattabiraman, University of British Columbia

Acknowledgments

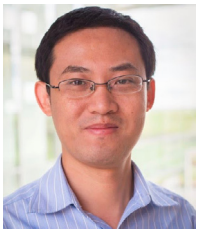
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-766363). This research was partially supported by U.S. National Science Foundation (CNS-1617967, CCF-1553645 and CCF-1718194).

References

[1] Coral Benchmark Codes [online], 2006.
[2] H.M. Aktulga, J.C. Fogarty, S.A. Pandit, A.Y. Grama, Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques, Parallel Comput. (2012).
[3] R. Ashraf, R. Gioiosa, G. Kestor, R.F. DeMara, C. Cher, P. Bose, Understanding the propagation of transient errors in HPC applications, in: SC, 2015.
[4] D.H. Bailey, L. Dagum, E. Barszcz, H.D. Simon, NAS Parallel benchmark results, in: SC, 1992.
[5] R. Battiti, Using mutual information for selecting features in supervised neural net learning, IEEE Trans. Neural Netw. 5 (4) (1994) 537–550.
[6] R.C. Baumann, Radiation-induced soft errors in advanced semiconductor technologies, IEEE Trans. Device Mater. Reliab. 5 (3) (2005) 305–316.
[7] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization, JMLR (2012).
[8] C. Bienia, S. Kumar, J.P. Singh, K. Li, The parsec benchmark suite: Characterization and architectural implications, in: PACT, 2008.
[9] P.S. Bradley, O.L. Mangasarian, Feature selection via concave minimization and support vector machines, in: ICML, 98, 1998.
[10] L. Cai, J. Gu, J. Ma, Z. Jin, Probabilistic wind power forecasting approach via instance-based transfer learning embedded gradient boosting decision trees, Energies (2019).
[11] J. Calhoun, L. Olson, M. Snir, Flipit: An LLVM based fault injector for HPC, in: Workshops in Euro-Par, 2014.
[12] J. Calhoun, M. Snir, L.N. Olson, W.D. Gropp, Towards a more complete understanding of sdc propagation, in: HPDC, 2017.
[13] F. Cappello, Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities, Int. J. High Perform. Comput. Appl. 23 (3) (2009) 212–226.
[14] M. Casas, B.R. de Supinski, G. Bronevetsky, M. Schulz, Fault resilience of the multi-grid solver, in: ICS, 2012.
[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: IISWC, 2009.
[16] X. Chen, X. Qiu, C. Zhu, X. Huang, Gated recursive neural network for chinese word segmentation, in: ACL, 2015.
[17] C.-Y. Cher, M.S. Gupta, P. Bose, K.P. Muller, Understanding soft error resiliency of BlueGene/Q compute chip through hardware proton irradiation and software fault injection, in: SC, 2014.
[18] B. Choubin, S. Khalighi-Sigaroodi, A. Malekian, Ö. Kişi, Multiple linear regression, multi-layer perceptron network and adaptive neuro-fuzzy inference system for forecasting precipitation based on large-scale climate signals, Hydrol. Sci. J. (2016).
[19] A. Coates, A. Ng, H. Lee, An analysis of single-layer networks in unsupervised feature learning, in: AISTATS, 2011.
[20] A. Das, F. Mueller, C. Siegel, A. Vishnu, Desh: deep learning for system health prediction of lead times to failure in hpc, in: HPDC, 2018.
[21] A. De Myttenaere, B. Golden, B. Le Grand, F. Rossi, Mean absolute percentage error for regression models, Neurocomputing 192 (2016) 38–48.
[22] D.A.G. De Oliveira, L.L. Pilla, M. Hanzich, V. Fratin, F. Fernandes, C. Lunardi, J.M. Cela, P.O.A. Navaux, L. Carro, P. Rech, Radiation-induced error criticality in modern hpc parallel accelerators, in: HPCA, 2017.
[23] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: A large-scale hierarchical image database, in: 2009 IEEE Conference on Computer Vision and Pattern Recognition, Ieee, 2009.
[24] P. Domingos, Bayesian Averaging of classifiers and the overfitting problem, in: ICML, 2000.
[25] H. Drucker, C. Cortes, L.D. Jackel, Y. LeCun, V. Vapnik, Boosting and other ensemble methods, Neural Comput. (1994).
[26] I.P. Egwutuoha, D. Levy, B. Selic, S. Chen, A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems, J. Supercomput. 65 (3) (2013) 1302–1326.
[27] J.H. Friedman, Stochastic gradient boosting, Comput. Stat. Data Anal. (2002).
[28] G. Georgakoudis, L. Guo, I. Laguna, Reinit++: Evaluating the performance of global-restart recovery methods for mpi fault tolerance, in: ISC, 2020.
[29] G. Georgakoudis, I. Laguna, D.S. Nikolopoulos, M. Schulz, REFINE : Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed, in: SC, 2017.
[30] L. Guo, G. Georgakoudis, K. Parasyris, I. Laguna, D. Li, Match: An mpi fault tolerance benchmark suite, in: 2020 IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2020.
[31] L. Guo, D. Li, MOARD: Modeling application resilience to transient faults on data objects, in: International Parallel and Distributed Processing Symposium, 2019.
[32] L. Guo, D. Li, I. Laguna, M. Schulz, Fliptracker: Understanding natural error resilience in hpc applications, in: SC, 2018.
[33] I. Guyon, A. Elisseeff, An introduction to variable and feature selection, JMLR 3 (Mar) (2003) 1157–1182.
[34] HackerRank, Hackerrank home page, 2009, https://www.hackerrank.com/.
[35] S.K.S. Hari, S.V. Adve, H. Naeimi, P. Ramachandran, Relyzer: Exploiting application-level fault equivalence to analyze app. Resiliency to transient faults, in: ASPLOS, 2012.
[36] J.L. Henning, Spec cpu2000: measuring cpu performance in the new millennium, Computer (2000).
[37] K. Hoste, L. Eeckhout, Comparing benchmarks using key microarchitecture-independent characteristics, in: 2006 IEEE International Symposium on Workload Characterization, 2006.
[38] M. Kaliorakis, D. Gizopoulos, R. Canal, A. Gonzalez, Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment, in: ISCA, 2017.
[39] C. Kalra, F. Previlon, X. Li, N. Rubin, D. Kaeli, Prism: predicting resilience of gpu applications using statistical methods, in: SC, 2018.
[40] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, U. Gunneflo, Using heavy-ion radiation to validate fault-handling mechanisms, IEEE micro.
[41] G. Kestor, I.B. Peng, R. Gioiosa, S. Krishnamoorthy, Understanding Scale-Dependent soft-Error Behavior of Scientific Applications, in: International Symposium on Cluster, Cloud and Grid Computing, 2018.
[42] R. Kohavi, A study of cross-validation and bootstrap for accuracy estimation and model selection, in: IJCAI, 1995.
[43] S. Kumar Sastry Hari, T. Tsai, M. Stephenson, S.W. Keckler, J. Emer, Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation, in: ISPASS, 2017.
[44] I. Laguna, M. Schulz, D.F. Richards, J. Calhoun, L. Olson, IPAS: Intelligent protection against silent output corruption in scientific applications, in: CGO, 2016.
[45] R. Leveugle, A. Calvez, P. Maistri, P. Vanhauwaert, Statistical fault injection: Quantified error and confidence, in: Proceedings of the Conference on Design, Automation and Test in Europe, European Design and Automation Association, 2009, pp. 502–506.
[46] G. Li, K. Pattabiraman, S.K.S. Hari, M. Sullivan, T. Tsai, Modeling soft-error propagation in programs, in: DSN, 2018.
[47] D. Li, J.S. Vetter, W. Yu, Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool, in: SC, 2012.
[48] A. Liaw, M. Wiener, et al., Classification and regression by randomforest, R News 2 (3) (2002) 18–22.
[49] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, O. Mutlu, Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory, in: DSN, 2014.
[50] L. Lusa, et al., Boosting for high-dimensional two-class prediction, BMC Bioinform. (2015).
[51] H. Menon, K. Mohror, Discvar: discovering critical variables using algorithmic differentiation for transient faults, in: PPOPP, 2018.
[52] B. Nie, J. Xue, S. Gupta, T. Patel, C. Engelmann, E. Smirni, D. Tiwari, Machine learning models for gpu error prediction in a large scale hpc system, in: DSN, 2018.
[53] B. Nie, L. Yang, A. Jog, E. Smirni, Fault site pruning for practical reliability analysis of gpgpu applications, in: Proceedings of the International Symposium on Microarchitecture MICRO, 2018.
[54] K. Parasyris, G. Tziantzoulis, C.D. Antonopoulos, N. Bellas, Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates, in: DSN, 2014.

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.
[56] W. Pei, T. Ge, B. Chang, Max-margin tensor neural network for chinese word segmentation, in: ACL, 2014.
[57] S.K. Sastry Hari, R. Venkatagiri, S.V. Adve, H. Naeimi, GangES: Gang Error Simulation for Hardware Resiliency Evaluation, in: International Symposium on Computer Arch. 2014.
[58] Y.S. Shao, D. Brooks, ISA-Independent workload characterization and its implications for specialized architectures, in: ISPASS, 2013.
[59] V. Sridharan, N. DeBardeleben, S. Blanchard, K.B. Ferreira, S. Gurumurthi, Memory errors in modern systems: The good, the bad, and the ugly, in: ASPLOS, 2015.
[60] R. Taborda, J. Bielak, Large-scale earthquake simulation: computational seismology and complex engineering systems, Comput. Sci. Eng. (2011).
[61] A. Thomas, K. Pattabiraman, Llfi: An intermediate code level fault injector for soft computing applications, in: SELSE, 2013.
[62] R. Venkatagiri, A. Mahmoud, S.K.S. Hari, S.V. Adve, Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency, in: MICRO, 2016.
[63] J. Wei, A. Thomas, G. Li, K. Pattabiraman, Quantifying the accuracy of high-level fault injection techniques for hardware faults, in: DSN, 2014.
[64] X. Xu, M.-L. Li, Understanding soft error propagation using vulnerability-driven fault injection, in: DSN, 2012.
[65] L. Yu, D. Li, S. Mittal, J.S. Vetter, Quantitatively modeling app resiliency with data vulnerability factor, in: SC, 2014.
[66] R.S. Zemel, T. Pitassi, A gradient-based boosting algorithm for regression problems, in: Advances in neural information processing systems, 2001.
[67] X. Zhang, X. Lu, Q. Shi, X.-q. Xu, E.L. Hon-chiu, L.N. Harris, J.D. Iglehart, A. Miron, J.S. Liu, W.H. Wong, Recursive svm feature selection and sample classification for mass-spectrometry and microarray data, BMC bioinformatics 7 (1) (2006) 197.

Dr. Luanzheng Guo is a postdoctoral researcher at the Pacific Northwest National Laboratory, working with the HPC Group in the research area between HPC and Machine Learning. He obtained his Ph.D. degree in Electrical Engineering and Computer Science from the University of California-Merced in 2020. His Ph.D. research focused on system resilience and reliability in large-scale parallel HPC systems.



Dr. Dong Li is an associate professor in the Department of Electrical Engineering and Computer Science, University of California, Merced. He is the director of the Parallel Architecture, System, and Algorithm Lab (PASA).

Dr. Ignacio Laguna is a Computer Scientist at the Center for Applied Scientific Computing (CASC) at the Lawrence Livermore National Laboratory (LLNL), California. His main area of research is high-performance computing (HPC); his main sub-area of research in HPC is programming models and systems. He is in particular interested in fault tolerance, fault resilience, debugging, software correctness and general software reliability.