IPAS: Intelligent Protection against Silent Output Corruption in Scientific Applications

Ignacio Laguna, Martin Schulz, David F. Richards
Lawrence Livermore National Laboratory
Livermore, CA, USA
{ilaguna, schulz6, richards12}@llnl.gov

Jon Calhoun, Luke Olson
University of Illinois at Urbana-Champaign
Urbana, IL, USA
{jccalho2, lukeo}@illinois.edu

Abstract

This paper presents IPAS, an instruction duplication technique that protects scientific applications from silent data corruption (SDC) in their output. The motivation for IPAS is that, due to natural error masking, only a subset of SDC errors actually affects the output of scientific codes—we call these errors silent output corruption (SOC) errors. Thus applications require duplication only on code that, when affected by a fault, yields SOC. We use machine learning to learn code instructions that must be protected to avoid SOC, and, using a compiler, we protect only those vulnerable instructions by duplication, thus significantly reducing the overhead that is introduced by instruction duplication. In our experiments with five workloads, IPAS reduces the percentage of SOC by up to 90% with a slowdown that ranges between $1.04\times$ and $1.35\times$, which corresponds to as much as 47% less slowdown than state-of-the-art instruction duplication techniques.

Categories and Subject Descriptors B.8.1 [*Performance and Reliability*]: Reliability, Testing, and Fault-Tolerance; D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Resilience, high-performance computing, compiler analysis, machine learning

1. Introduction

We present *IPAS*, a user-guided technique to reduce the degree of silent data corruption (SDC) in the outcome of scientific high-performance computing (HPC) applications caused by soft errors [6, 27].

The potential increase of soft errors due to scaling trends in device sizes and voltage is cause for concerns in the HPC community as larger supercomputers are being built [9, 22]. While a considerable portion of soft errors eventually manifests as visible failures, another portion remains silent. The former category yields architecture-level symptoms (e.g., hardware exceptions) and/or system-level symptoms (e.g., node crashes or hangs), which can be handled by architectural symptom-based [38] and system-based detection techniques [15, 18]; errors in the latter category, however, require more careful handling as they corrupt silently the output of applications, which may unknowingly yield incorrect scientific results. Although there exist effective techniques to protect the data that scientific applications operate on, such as parity and error-correcting codes (ECC) [11, 14], the question of efficiently protecting the unstructured logic that exists in processors remains open. As processor counts increase on the path to exascale computing, the total area of this logic (and of other unprotected regions) of a chip also increases, which may open the door to higher SDC rates.

Instruction duplication has been proposed as a software approach to mitigate SDC [32, 33]. This approach detects faults by duplicating computation instructions and comparing their results. Although effective on reducing SDC, most instruction duplication techniques incur substantial runtime overhead. Recent work proposes to reduce the overhead of duplication by protecting only instructions that yield SDC when subjected to a fault and leaving symptom-generating instructions unprotected since, when faults affect them, they become visible to the system and can be handled by symptom- or system-based detection techniques [17].

However, existing instruction duplication techniques do not consider that only a subset of SDC errors actually affects data and computations sufficiently to change the output of scientific codes—we call these errors *silent output corruption* (SOC) errors—and as a consequence, they tend to overprotect, causing unnecessary runtime overhead. In addition, these approaches rely on hardware-specific heuristics and little (or no) application-level information, thus they cannot adapt to application-specific requirements in terms of SOC. Our approach addresses these limitations.

CGO'16, March 12–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-3778-6/16/03...$15.00
http://dx.doi.org/10.1145/2854038.2854059

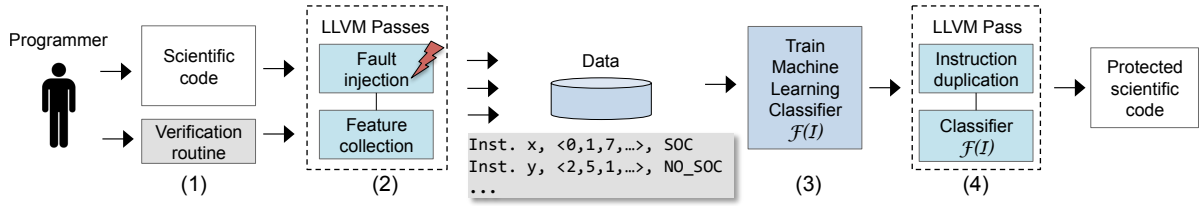


Figure 1: Steps in the workflow of IPAS.

Our approach is based on two observations. First, in most HPC codes, a portion of SDC errors are masked naturally and do not produce SOC, that is, the outcome of applications is acceptable to scientists even when errors occur. Protection code for these masked errors and the resulting runtime overhead is unnecessary. Second, most scientific computations allow algorithmic checks to determine if the application result is affected by SOC. For example, outcome of matrix multiplications or FFT kernels can be verified against exact solutions using a norm. This ability to check for SOC allows us to synthetically generate *examples* of SOC-generating code (by fault injection) and then use these examples to train a prediction model to protect the entire code.

Based on these observations, IPAS (**I**ntelligent **P**rotection **A**gainst **S**ilent output/result corruption) uses machine learning to build a classifier that is used to predict the minimal set of instructions that need to be duplicated to avoid SOC, therefore eliminating a large number of unnecessary checks. The machine learning classifier is trained with fault-injection data that is automatically labeled by a user/algorithm-defined function verifying the final outcome. Data include instruction features, such as the instruction type, the size of the basic block and function an instruction belongs, and the number of instructions that are affected. We then use the classifier in a compiler pass to protect selected instructions (i.e., SOC-generating instructions) by duplication. A key benefit of our machine-learning approach is that IPAS adapts to the particular level of protection that is required in an application to reduce SOC without incurring unnecessary overhead.

We prototype IPAS as compiler passes in LLVM and show its benefits in reducing overhead and SOC on five representative HPC workloads: two HPC mini applications (CoMD [1] and HPCCG [2]), two commonly used kernels in HPC (algebraic multigrid and FFT), and an HPC benchmark (NPB IS [13]). We compare IPAS to the state-of-the-art approach of selectively duplicating only non-symptom-generating instructions [17], and traditional full duplication via statistical fault injection with FlipIt [8]. In contrast to previous work, which focuses on serial code [17, 23, 32, 37], we demonstrate IPAS in parallel MPI scientific codes.

In summary, we make the following contributions:

- A novel hardware-independent framework to protect HPC codes against SOC with minimal performance overhead while providing a large degree of SOC reduction;

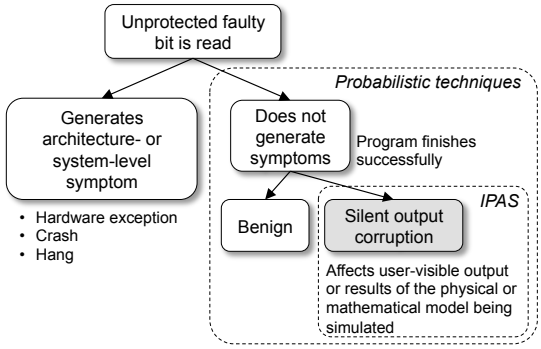


Figure 2: Classification of faulty application behavior.

- A machine learning approach to classify the instructions in an application that yields SOC in the event of an SDC;
- An implementation of this approach in LLVM that supports parallel scientific applications;
- An evaluation of the approach benefits on five small scientific codes.

We show that IPAS is scalable—a key feature for scientific HPC application—and that it can reduce SOC up to 90% in these codes, with an application slowdown of $1.04\times$–$1.35\times$, which corresponds to 47% less overhead compared to state-of-the-art instruction duplication techniques.

2. Motivation

To motivate our approach, it is useful to classify the application's behavior under the effect of soft errors as in Figure 2. A fault that propagates to the application may or may not generate a system- or architecture-level symptom, such as a hardware exception, a crash, or a hang. While symptom-generating faults can be handled by commonly used HPC fault-tolerance techniques, such as checkpoint/restart, SDC, which does not generate symptoms, remains a major concern.

We take advantage of the observation that, although from the hardware's perspective SDC errors corrupt an application's outcome, from the application's perspective, depending on the data that faults corrupt, they may not significantly change the results of the physical or mathematical model used in a scientific simulation. To illustrate this, consider a molecular dynamics code in which an error in a floating-point register corrupts the position of an atom. If the error affects a low significant bit of the mantissa, the atom position would

not change much, and, depending on the physical phenomena of interest in the simulation (e.g., total energy), the results would not be too different from the error-free case. On the other hand, if the error affects the exponent, the atom position would change substantially, possibly yielding different user-visible simulation results.

2.1 Verifiable Computations

Our targets are scientific computations that allow the verification of the final application output. In some cases this verification involves using application outcomes from error-free runs. For instance, we can verify the output of an FFT calculation for a particular input using the results of error-free runs and by calculating the difference using an norm (e.g., Euclidean norm or p-norm)—if the difference is lower than a particular tolerance threshold, we can consider it a valid result. In other cases, this only involves data of a single run. For instance, for an algebraic multigrid (AMG) calculation, the verification involves checking that the error of the solution is less than a user-defined tolerance level.

Soft computations. Note that verifiable scientific computations are not necessarily "soft" computations [20], such as multimedia decoding applications, which can inherently tolerate a level of noise. Soft computations have a fidelity metric that is usually defined in terms of user's perception (e.g., quality of audio and video), whereas in a scientific code a verification metric depends on the numerical accuracy of the physics or mathematical model being simulated. Although some verifiable scientific computations can tolerate low-fidelity outputs (e.g., Monte Carlo codes), most others are "hard" computations (e.g., matrix multiplication or FFT codes) that require high fidelity or exact results.

Techniques used to control overhead of instruction duplication in soft computations may be used in HPC, yet only heuristics have been introduced to determine instruction protection [23, 26, 37]. Extending these heuristics to large HPC codes is tedious, thus limiting the practical application of these techniques in HPC. Our goal is to design automatic, general-purpose techniques that avoid these decisions.

Use of verification in the workflow. Our framework only uses verification (that detects if SOC occurs) to create training examples of SOC-generating instructions. With these examples, it trains a prediction model that is used to decide whether or not to protect (via duplication) the entire set of instructions. Note that, although verification can be used alone for SOC detection instead of instruction duplication, the performance cost of this approach would be higher—with instruction duplication, SOC errors are detected close to their occurrence, enabling the use of recent checkpoints rather than wasting time in restarting the entire computation.

3. Overview of the Approach

Fault Model. We consider transient faults (or soft errors) that occur in the processor as single bit flips [6, 27]. We consider only faulty bits that are read and that are unprotected: if a fault corrupts a bit that is never read, it is benign; and if the bit is protected (e.g., by ECC) we assume that the fault will be either corrected, or, if correction is not provided, the fault will become a Detectable Unrecoverable Error (DUE) [39], which is followed by a node crash—an observable symptom, which allows commonly used system-level methods in HPC to handle the fault, such as checkpoint/restart.

We target faults that affect the resulting value of hardware instructions and that corrupt the content of registers. This includes instructions that use the functional units, e.g., the ALU, and pointer instructions that compute addresses of loads and stores. As in previous work [32, 33], we assume that data in memory and caches as well as the results of memory operations (i.e., data obtained by loads and stores) are protected with ECC, which is common practice in today's HPC systems. We do not consider faults in control-flow instructions (e.g., branches and function calls) as they can be handled through control-flow checking techniques [5]; however, we do consider faults that affect the values that are returned from function-call instructions.

Workflow of our Approach (see Figure 1):

1. **Verification routine:** users provide a verification routine for the application, or for the kernel within the application, that will be protected by IPAS.
2. **Data collection**: IPAS performs statistical fault injection in random instructions of the target code. For each instruction in which a fault is injected, it collects features of the instruction (e.g., its type and the number of instructions it affects) and saves them in a *feature vector*. Then, it determines if the application suffered from SOC, using the verification routine, and labels the feature vector.
3. **Training**: IPAS uses machine learning to train a classifier $\mathcal{F}(\mathcal{I})$ using the feature vectors as input. The classifier can then be used to classify new instructions $\mathcal{I}$ as either 1 (SOC-generating) or 0 (non-SOC-generating).
4. **Application protection**: IPAS checks every instruction in the code using the learned classifier $\mathcal{F}(\mathcal{I})$ and duplicates it if the classifier predicts it as an SOC-generating instruction. Since compiler optimizations may conflict with instruction duplication, this procedure is performed after all user-level optimizations are performed in the code.

The result of this workflow is a protected scientific code that can be used in production calculations without any need to repeat steps 1–4.

4. Design of IPAS

In this section we describe the main building blocks of our approach: the fault injection methodology to record training data, the features that we use, the machine learning methodology, and the code duplication algorithm.

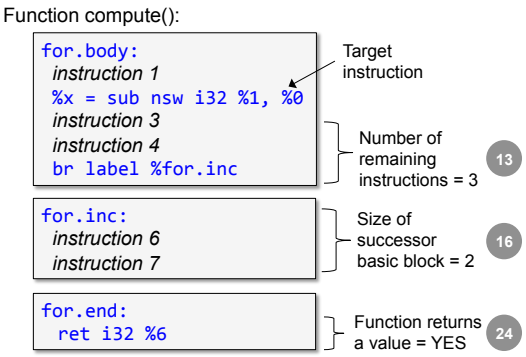


Figure 3: A sample function $compute()$ with three basic blocks. The gray circles denote the ID of the features. This provides three examples of the way features are gathered.

4.1 Fault Injection and Data Labeling

The goal of fault injection (Step 2 in Figure 1) is to create a training set for the machine learning classifier. For each instruction in which a fault is injected, we obtain a feature set that characterizes the instruction (Section 4.2 describes these features). Then we inject a fault and determine whether it yields SOC or not, and label the instruction's feature set as one of the following two classes:

- *Class 1*: SOC-generating instruction
- *Class 2*: non-SOC-generating instruction

Each labeled feature set is added to the training set.

Since a classifier can be built using a subset of the population space, we use statistical fault injection (SFI) to build the training set, in which only a subset of possible randomly selected instructions is subjected to faults. For each test application, we use 2,500 instruction samples, which we found experimentally is sufficient to train our classifier.

We use FlipIt [8], an LLVM-based fault injection tool for HPC applications, for fault injection. FlipIt provides the flexibility to inject faults into random instances of an instruction, bits within a byte, and MPI ranks. In our experiments, we select instruction instances randomly and flip a random bit in the resulting value of the instruction.

4.2 Instruction Features

We record features (explanatory variables) of the instructions in which faults are injected. Our goal is to characterize SOC-generating instructions in terms of these features. We therefore select features that are informative of error propagation (the key phenomenon behind SOC). Features that do not characterize error propagation (e.g., name of functions) or that cannot be accessed from the LLVM IR (e.g., architecture dependent features) are not used.

Table 1 shows a description of the 31 features that we use and their categories. Figure 3 illustrates some of them. The first category covers properties of the instruction itself, such as its type and the size of its return value. The second category covers properties of the basic block the instruction belongs to, such as the size of the basic block and the number of instructions between the target instruction and the last instruction in the basic block. The third category is similar to the second one, but covers properties of the function that the instruction belongs to. The last category corresponds to properties of a program slice that IPAS computes based on the instruction. A slice of a particular instruction x is the set of instructions that affect x (backward slice), or the set of instructions that x influences (forward slice). Since we are interested in characterizing error propagation, between the point in which a fault corrupts an instruction to the point in which it corrupts output, we use a forward slice. We use Weiser's algorithm [40] to compute the forward slice, which is then used to compute the values of features 25–31. Note that Table 1 shows features, and not parameters of our model. Later (Section 4.3.1) we present our classification model and its parameters (Section 4.3.2).

Table 1: Features that are collected for each instruction. We denote boolean features by `[Bool]`, and integer features by `[Int]`

	No.	Description
Instruction	1	`[Bool]` is binary operation
	2	`[Bool]` is add or sub operation
	3	`[Bool]` is multiplication or division operation
	4	`[Bool]` is division remainder operation
	5	`[Bool]` is logical operation (e.g., AND, OR)
	6	`[Bool]` is call instruction
	7	`[Bool]` is comparison instruction
	8	`[Bool]` is atomic read/write instruction
	9	`[Bool]` is get-pointer instruction
	10	`[Bool]` is stack-allocation instruction
	11	`[Bool]` is cast instruction
	12	`[Int]` bytes in the instruction's result (1,8,16,...)
Basic Block	13	`[Int]` number of remaining instructions
	14	`[Int]` size of basic block (BB)
	15	`[Int]` number of successor basic blocks
	16	`[Int]` sum of basic block sizes of successor BBs
	17	`[Bool]` basic blocks is within a loop
	18	`[Bool]` has a PHI instruction
	19	`[Bool]` its terminator is a branch instruction
Function	20	`[Int]` remaining instructions to reach return
	21	`[Int]` number of instructions in the function
	22	`[Int]` number of basic blocks in the function
	23	`[Int]` number of future function calls
	24	`[Bool]` it returns a value
Slice	25	`[Int]` number of instructions in the slice
	26	`[Int]` number of loads in the slice
	27	`[Int]` number of stores in the slice
	28	`[Int]` number of function calls in the slice
	29	`[Int]` number of binary operations in the slice
	30	`[Int]` number of stack-allocation instructions
	31	`[Int]` number of get-pointer instructions

4.3 Machine Learning

The core of IPAS is the use of machine learning and is executed as part of step 3 in Figure 1.

4.3.1 Selecting the Classifier

Our data and problem space has specific requirements, thus we must carefully select the machine learning method for IPAS. In particular, our requirements are as follows. First, the

method must deal efficiently with *class-imbalanced data*, that is, data in which the number of samples in one class is much larger than in the other class. In our test applications, we found that only between 3% and 10% of the data correspond to SOC cases, which clearly creates class imbalance in our data. Second, the method should make good decisions for data points that are outside the training set (since we might be limited to few fault-injection outputs), and should have only few parameters to tune. We found, through experimentation, that support vector machines [12] (SVM) meet all of the above requirements, in comparison to other commonly used classification schemes, such as decision trees and nearest neighbor. In particular, we found that SVMs handle well our class-imbalanced data. Note, though, that IPAS is not restricted to SVMs—other classification methods can be used as long as they meet the above requirements.

Given a set of training examples, each marked as belonging to one of two categories (SOC-generating or non-SOC-generating), an SVM constructs a set of hyperplanes in a high-dimensional space, which can be used for classification. We use the C-SVM algorithm based on Chang et al. [10] to train our classifier. We encourage the reader to examine the literature for a detailed description of the theory behind SVM [7].

4.3.2 Training

Machine learning algorithms can suffer from *overfitting*, which occurs when the trained model describes random error or noise instead of the underlying relationships. An overfitted model would have poor predictive performance. In the context of IPAS, this means that the model would incorrectly classify instructions as SOC-generating, which could lead to two (non mutually exclusive) situations: either it unnecessarily protects too many instructions, which increases runtime overhead, or it fails to protect required instructions to reduce SOC. In addition to avoiding overfitting, one must tune the following parameters of the model to obtain the highest accuracy level possible, given the training data:

- Parameter C: a penalty factor that allows us to trade off training error versus model complexity. A small value increases the number of training errors, whereas a large value promotes overfitting.
- Parameter γ: the coefficient of the *kernel function*, which is a similarity function that is used in SVM (and other kernel methods) to operate on feature vectors. We use a radial basis function kernel, which tends to provide smooth solutions [29].

To tune the C and γ parameters, we perform cross validation [7], a widely used technique to asses the performance of a predictive model. We vary parameters C between 1 and $100,000$ and parameter γ between 0.00001 and 1, and we select 500 different combinations of them. We call each combination a possible *configuration* of IPAS. To select the best

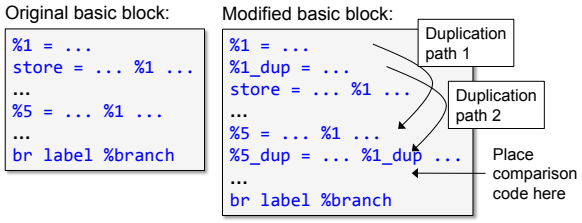


Figure 4: Duplication paths in IPAS.

configuration, we measure the *F-score* metric (also known as *F-measure*) [21]:

$$F\text{-}score = 2 \cdot \frac{Accuracy_1 \cdot Accuracy_2}{Accuracy_1 + Accuracy_2}, \qquad (1)$$

where $Accuracy_1$ is the fraction of SOC-generating training examples correctly classified, and $Accuracy_2$ is the fraction of non-SOC-generating training examples correctly classified, and select the configuration with the highest *F-score*. This metric reaches its best value at 1 and worst score at 0, and provides a single metric to estimate the performance of classifying both SOC-generating and non-SOC-generating instructions—as mentioned earlier, IPAS must do well on classifying both cases to achieve high SOC reduction while maintaining low runtime overhead.

4.4 Code Duplication

The last step in our framework (Step 4 in Figure 1) is code duplication. Previous work studied the mechanics of inserting redundant code into a single thread of execution [30, 32, 33]. The most common approach involves duplicating all computation instructions and inserting comparison instructions at synchronization points (i.e., store and control-flow instructions) to check for faults. We build on these existing techniques in our approach, but use the machine learning results to select where to apply duplication.

In particular, IPAS duplicates all instructions that the machine learning classifier predicts as SOC-generating. It then builds *duplication paths* for these instructions, where a duplication path is a series of instructions $i_1, i_2, \ldots, i_N$ in which for each instruction i_x there is at least one instruction i_y that uses i_x, where $x < y$, except for instruction i_N (the last in the series). Duplication paths are built using use-def and def-use chains—Figure 4 illustrates the idea. Similar to previous work, we limit the span of duplication paths to a single basic block (i.e., the basic block the instructions belong to); thus, two duplication paths from a single basic block always have similar instructions. IPAS then adds a comparison instruction at the end of each duplication path.

Note that, like previous work [32, 33], we do not duplicate loads and stores because the common practice is to assume that data is protected with ECC and because duplicating these instructions substantially increases execution overhead. Also note that, instead of adding comparison instructions before load instructions (as in previous work), we do it at the end of

duplication paths. This implies that an error could propagate slightly further in the execution path, but it would be always caught before a branch instruction.

It is possible to pinpoint instructions in a basic block that do not form a path (i.e., an instruction does not consume or it is not used by other instructions). In this case, IPAS adds a comparison instruction right after the duplicate ones.

4.4.1 Parallel Code

Scientific codes run at large scale as parallel programs, thus resilience solutions need to support parallel programming models. IPAS supports OpenMP and MPI applications, the most common parallel models in HPC. For OpenMP, we assume that a compiler "outlines" sections of code that must run in parallel into separate functions that are then invoked in multiple threads by the OpenMP runtime library. This is a common practice in OpenMP implementations [3]. Since we do not duplicate control-flow instruction (including function-call instructions) the use of OpenMP outlined functions is safe in IPAS. The same principle is applicable in MPI, since applications access the MPI library via function calls. Because MPI applications typically run on large number of nodes, we must ensure that, when a fault is detected (either by IPAS or the system), it is propagated to other nodes. For this, we rely on the default behavior of MPI implementations, in which, if a process fails, the rest of the processes abort, effectively generating a system-level symptom. IPAS does not instrument the OpenMP and MPI libraries themselves since parallel applications are expected to spend a small fraction of the time executing code in them.

5. Experimental Setting

We describe the testbed system, workloads, comparison baseline, and fault-injection methodology to evaluate IPAS.

5.1 Testbed System

For all experiments we use a cluster that has two Intel 6-core Xeon X5660 2.8 GHz CPUs with 24 GB of memory per node. We build IPAS on LLVM v3.6 and use MVAPICH2 MPI v1.9 in all experiments.

Since errors in external libraries can propagate to the application and produce SOC (whether statically or dynamically linked), to obtain maximum protection, libraries must be protected either with IPAS or with other mechanisms [32]. In the current implementation of IPAS, we include static libraries implicitly, but we currently do not support dynamically shared libraries, which typically also includes runtime system libraries, such as libc.

Since our focus in the paper is to show scalabilty, we present results only for the MPI version of the applications. Although OpenMP applications can make use of IPAS, experiments for the paper with OpenMP require a larger fault-injection campaign, since one must take into account the thread dimension, thus we leave this experimentation phase to future work.

5.2 Workloads

We use five workloads that support output verification to evaluate IPAS: two mini applications (CoMD [1] and HPCCG [2]), two commonly used kernels in HPC (algebraic multigrid and FFT), and an HPC benchmark (NPB IS [13]). Although we do not evaluate IPAS in a production-size application, we select kernels and mini applications that are representative of HPC workloads. We use kernels because they are the fundamental building block of HPC applications; thus, programmers could apply IPAS to each kernel independently in large applications. In addition to kernels, we use mini applications because their sensitivity to multiple inputs and execution paths is more realistic as they have a richer code structure. Table 2 describes each code and their verification procedure.

Note that verifications are only done once in the workflow of IPAS (step 2 in figure 1). Once applications are protected with instruction duplication, verifications can be turned off to minimize execution time. However, they can be used to increase fault coverage since our approach, being probabilistic by nature, does not detect all faults.

5.3 Comparison Baseline

Shoestring [17] serves as a valid point of comparison with IPAS since both approaches have the same goal: to avoid protecting symptom-generating instructions (or instructions that are consumed by symptom-generating instructions) since faults that affect these instructions can be covered with symptom- or system-based detection schemes. Shoestring's algorithm to identify instructions requiring protection is based on data-flow analysis and hardware-specific knowledge; for example, it requires knowledge of exception-throwing instructions from the instruction set architecture (ISA). On the other hand, the IPAS classification approach is more generic, and does not require architecture-specific knowledge.

To enable a direct comparison with IPAS, our baseline uses the Shoestring's policy for selecting instructions that require protection. We do this by training our classifier in a different way: instead of using SOC-generating or non-SOC-generating labels for instructions in our data, we label the instructions with symptom-generating and non-symptom-generating labels. In our definition of symptoms, we include fatal ISA-defined exceptions as in [17] (which we assume manifest as application crashes), and application hangs—due to the iterative nature of scientific codes, faults can occasionally result in hangs if, for example, loop-condition variables are affected. Finally, when protecting the code, we only duplicate instructions that are classified as non-symptom-generating. Although there are differences between this and the original Shoestring implementation (which is not publicly available), we expect it provides comparable results with respect to the original version as long as a perfectly-trained classifier is used.

Table 2: Workloads and verification routines used in our evaluation

Code name	Description	Output verification routine
CoMD	It is a mini application for molecular dynamics simulations [1]. It considers the simulation of materials where the interatomic potentials are short range, which requires the evaluation of all forces between atom pairs within a cutoff distance.	We rely on the fact that, in an MD simulation, the total energy of the system is conserved. We check if the total energy has been conserved by determining if it falls within 3 standard deviations of the mean of a Normal distribution, and if not, the routine declares SOC.
HPCCG	It is a mini application that uses the conjugate gradient (CG) to compute the numerical solution of a system of linear equations. CG is commonly used to solve large sparse systems, which often arise when numerically solving partial differential equations or optimization problems.	The verification routine checks that the difference between the known exact and the computed solution is less than a user-defined tolerance level of 1e-06 in a maximum number of iterations. In our experiments we use a $nx = ny = nz = 50$ input with 124 for the iterations limit.
AMG	Algebraic multigrid is an iterative solver for large, sparse linear systems [34]. AMG constructs a hierarchy of sparse operators. These coarse representations of the problem are traversed to iteratively reduce the error in the solution. We use the solve kernel of an AMG code and solve a 2D problem of size 10K x 10K on the finest level of a 4-level hierarchy.	First, the initial inputs to the solver are checked for corruption by reading correct versions from disk. Second, we check if the solver has arrived to a solution using a tolerance of 1e-06 in the allotted number of iterations (i.e., 1,000 in our code). If any of these checks are violated, the routine declares an SOC.
FFT	It computes the discrete Fourier transform (DFT) and its inverse, it is a widely used kernel in scientific codes. We use a kernel that computes the 2D FFT (and the inverse) of a matrix within a 100-iterations loop.	We compute the L-2 norm between the output of an error-free run and the output of a fault-injection run; if the difference is higher than 1e-06, the verification routine declares SOC.
IS	It is part of the NAS Parallel Benchmarks [13], and performs a large integer sort, an operation that is important in particle method codes. It tests both random memory access and communication performance.	We use the verification routine that is provided by the benchmark, which involves iterating over keys in the sorted array and determining if key $i-1$ is less than key i; if this check is not passed, we declare an SOC.

5.4 Fault Injection

Since it is impossible to inject faults, in a reasonable period of time, into all locations of the code and at all clock cycles, we use statistical fault injection to demonstrate IPAS (as we did in Section 4.1 to obtain training samples)—the same approach has been used in previous work [17]. We run 1,024 fault injection runs at random LLVM instruction instances and bits for each combination of approach, configuration, and application, for a total or 51,200 fault injection runs.

We estimate the margin of error for our sampling approach using statistical theory. Under the assumption that soft errors are uniformly distributed across instructions and bits within a byte, and that the characteristic of populations follow a normal distribution—a common assumption in previous work [25]—we have verified that the margin of error for our measurements are low (see Section 6.2).

5.5 Outcome Categories

In each fault injection run, we classify the outcome as:

- *Observable symptom:* the injected fault produced a crash, a hang, or a substantially longer execution time. In these cases, we assume that the system is able to recover the application from the fault using commonly used fault-tolerance techniques in HPC such as checkpoint/restart.
- *Detected by duplication:* the injected fault is detected by the code that is added as part of the instruction-duplication LLVM pass.
- *Masked:* the injected fault did not produced SOC based on the application-dependent verification routine.
- *Silent output corruption:* the fault produced SOC.

Note that in our definition of masking, we do not consider faults that are architecturally masked, since our fault injection is at application level. Architecturally masked errors are benign from the application point of view, so we can safely ignore these errors when reducing SOC.

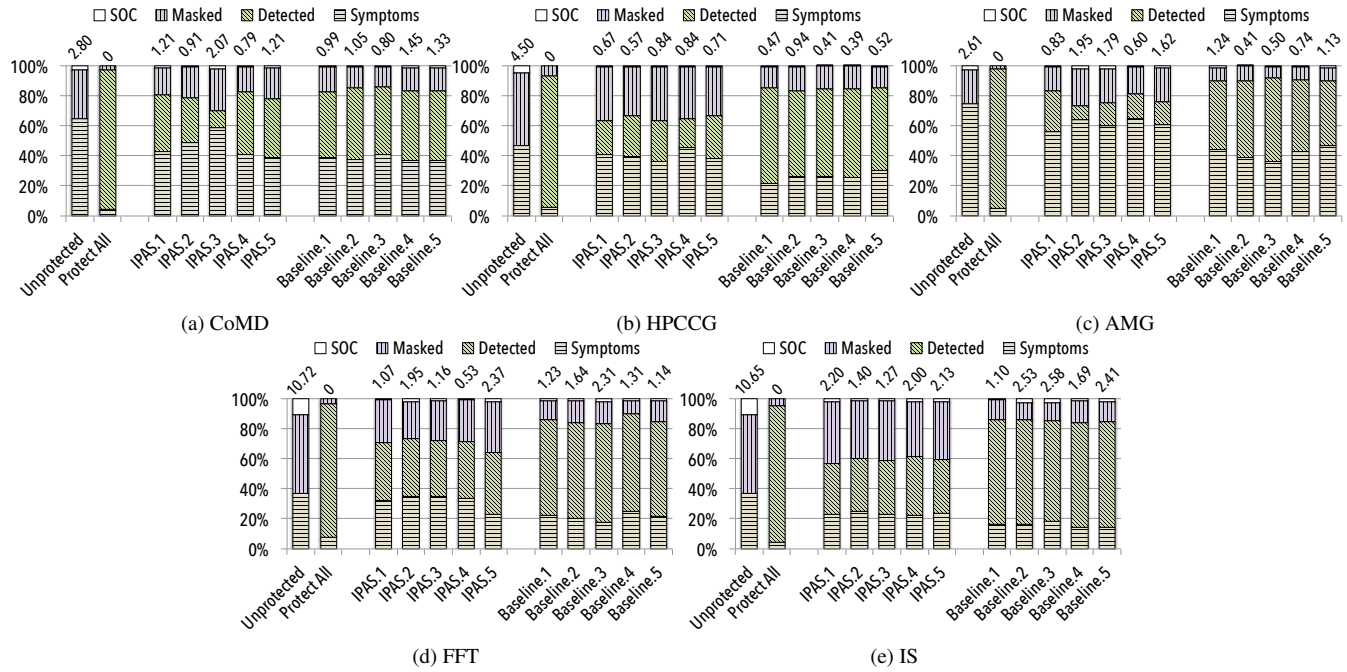
6. Evaluation and Results

We present our evaluation results for IPAS. To quantify fault coverage and SOC reduction, we first use the serial version of the codes with a single MPI process. Then, we demonstrate the scalability of the approach by measuring slowdown, using the best configuration on the parallel MPI versions.

6.1 Best Configurations

The performance of our approach depends on the accuracy of our machine learning classification model: SVM. As we explain in Section 4.3.2, we use the model configuration with the highest *F-score* metric when training our SVM classifier. Although this is a well-accepted approach in the machine learning community, selecting the configuration with the highest *F-score* may not always allow IPAS to perform optimally because of two reasons. First, *F-score* values are taken when doing cross validation—a process that involves only training data. These data may miss important samples (recall that we only use 2,500 training examples), leading to incorrect classifications when these samples are observed in the testing phase. Second, *F-score* does not consider application slowdown (i.e., the overhead that is incurred by duplication), an important metric to weight.

Given the aforementioned limitations, and to make fair comparisons, instead of using only the best configuration from the *F-score* analysis, we use the best N configurations. To do this, we rank each configuration based on its *F-score* and then select the top N configurations in our evaluation. Although there is no uniformly accepted criterion in the literature to select N, previous work has used at least 3 configurations (i.e., $N = 3$) to compare the performance of different SVM configurations [41]. Here we use the top-5 configurations ($N = 5$)—to be less conservative and to analyze a richer set of results—however we expect similar results with $N = 3$ since the top three configuration usually yields the best combination of accuracy and slowdown.



(a) CoMD (b) HPCCG (c) AMG

(d) FFT (e) IS

Figure 5: Coverage results for our test codes. Labels on the top of the bars show the % of SOC.

Table 3: Number of static LLVM instructions and lines of code

	CoMD	HPCCG	AMG	FFT	IS
Static instructions	12240	5107	4478	566	1457
Lines of code	3036	1313	952	249	701

6.2 Coverage Results

Figures 5(a)–(e) shows coverage results for our codes. Labels on the top of the bars show the proportion of SOC. The first bar in each plot shows results for the unprotected case. The second bar shows results for full duplication (i.e., duplicating all instructions). We estimate that the margins of error for the SOC percentages in the unprotected case, at a confidence level of 95%, are as follows: 0.71% (CoMD), 0.89% (HPCCG), 0.68% (AMG), 1.34% (FFT) and 1.33% (IS).

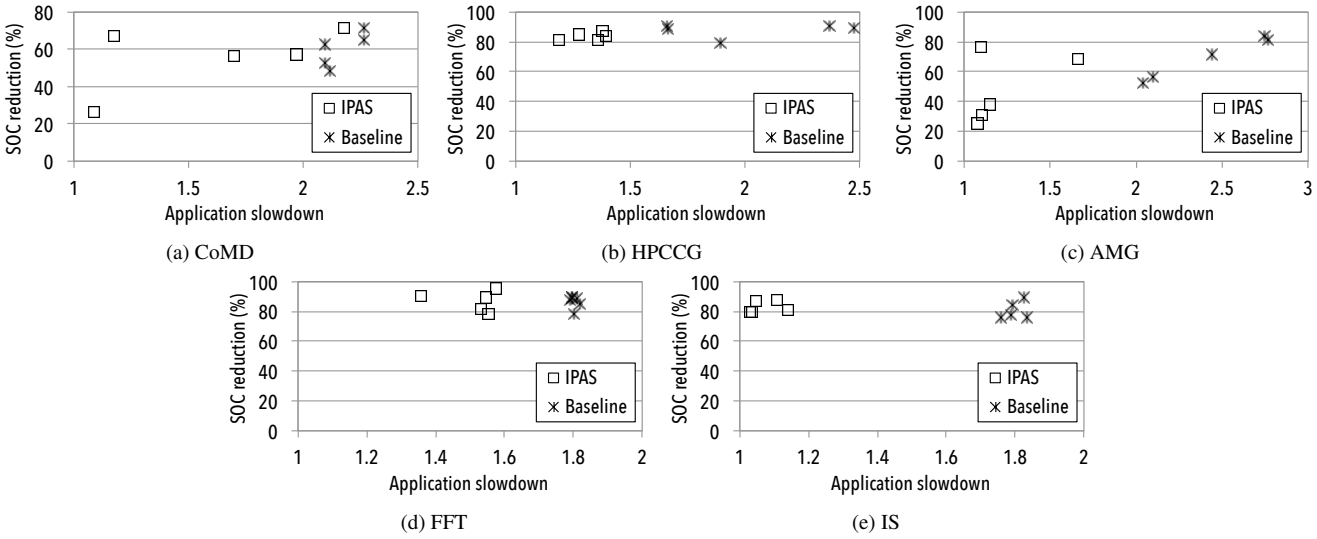
Our first observation is that, based on our verification methodology, the proportion of SOC in these codes is low: between 2.61% and 10.75%. Table 3 gives an overview of the size of the codes in terms of static LLVM instructions and lines of code. We observe that in the largest codes (CoMD, HPCG, and AMG) the percentage of SOC is smaller than that of the smallest codes (FFT and IS). In contrast, we observe that the smallest codes experience higher level of masking. We believe that since FFT and IS execute harder and more deterministic computations than CoMD, HPCG, and AMG, they experience higher levels of SOC.

The rest of the bars in Figures 5(a)–(e) show results for the 5 best configurations of IPAS and Baseline. As expected, the outcome proportions in different configurations of the same technique vary. However, Baseline detects a higher number of faults since it protects more instructions than IPAS. In terms of SOC reduction, both techniques have comparable performance; however, as we show in the next section, existing techniques may overprotect. Figure 7 shows the percentage of duplicated instructions averaged over the 5 configurations for each technique. IPAS protects less instructions than Baseline, which explains our previous observation about differences in fault detection proportions.

6.3 Performance Results

We compare application slowdown (ratio between execution time with duplication and without it) with the percentage of SOC reduction relative to the proportion of SOC of the unprotected case. Figures 6(a)–(e) show the results. We observe varying SOC-reduction-vs-slowdown behavior among the different configurations of IPAS, in contrast to Baseline, which exhibits more stable behavior. We attribute this to the fact that the data for training Baseline is less imbalanced than the data for training IPAS, thus the classifier, after trained, is more stable—the ratio of class1-to-class2 in Baseline is smaller than the same ratio in IPAS since the fraction of SOC-generating samples is smaller in comparison to the fraction of non-symptom-generating samples. We believe that this variance in IPAS can be reduced using a larger, and less imbalanced, training data set. However, this requires a much more extensive fault injection campaign (to obtain more samples of SOC-generating instructions).

In terms of the amount of SOC reduction, both techniques are comparable, except in AMG and HPCCG, in which Baseline achieves slightly higher SOC reduction than Baseline. Nevertheless, we always find a configuration in IPAS that outperforms Baseline in terms of runtime overhead, while



(a) CoMD (b) HPCCG (c) AMG

(d) FFT (e) IS

Figure 6: Percentage of SOC reduction versus slowdown in IPAS and Baseline.

maintaining comparable SOC reduction. For example, in AMG it can reduce SOC by 77% at a slowdown of $1.1\times$, and in IS it can reduce SOC by 86% at a slowdown of $1.04\times$. For CoMD, HPCCG, and FFT, SOC reductions can also be high, but at a higher overhead cost. This confirms that IPAS can intelligently select the instructions that require protection based on the application-specific requirements of SOC tolerance.

Ideally the best configuration for a technique like IPAS must consider both the SOC reduction percentage and the slowdown at the same time. Thus, the *ideal* configuration point would be the closest to the (1, 100) point (i.e., slowdown $= 1.0$ and SOC reduction percentage $= 100$) in the plots of Figures 6(a)–(e). Using this criterion, we find the best configuration point for IPAS and Baseline by computing the point (out of the five configuration points) with the minimum Euclidean distance to the ideal point. The results are shown in Table 4.

6.4 Scalability

We use the best configuration settings for each application (from Table 4) in our scalability experiments. We vary the number of MPI processes using strong scaling and measure application slowdown for IPAS. The slowdown here is the ratio between the job execution time with and without duplication. We present the results in Figure 8. We observe that slowdown stays overall constant as the number of MPI processes increases. This confirms our scalability expectations of IPAS since it instruments computation code only, which does not necessarily incur higher (data- or latency-related) communication overhead. Based on these results, we expect that the slowdown at extreme scales will be small (within the range of noise of the system); thus IPAS has the potential to be used as a low-cost technique to reduce the levels of SOC in scientific codes.

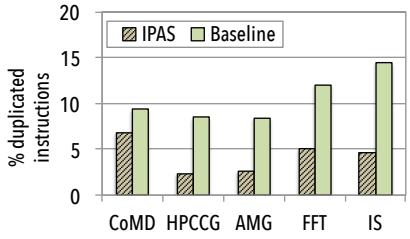


Figure 7: Comparison of the average of duplicated instructions for the top-5 configurations.

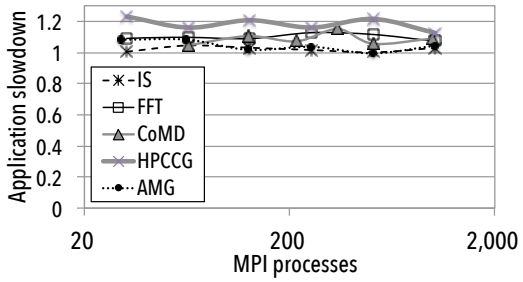


Figure 8: Scalability measurements for the best configuration of IPAS in each code. Slowdown does not vary considerably with scale.

Table 4: Results for best configurations of IPAS and Baseline (based on the ideal point criterion)

Code	SOC reduction (%)		Slowdown	
	IPAS	Baseline	IPAS	Baseline
CoMD	67.58	62.74	1.17	2.09
HPCCG	81.42	90.96	1.18	1.66
AMG	76.89	73.88	1.1	2.1
FFT	90.02	88.49	1.35	1.81
IS	86.88	84.11	1.04	1.79

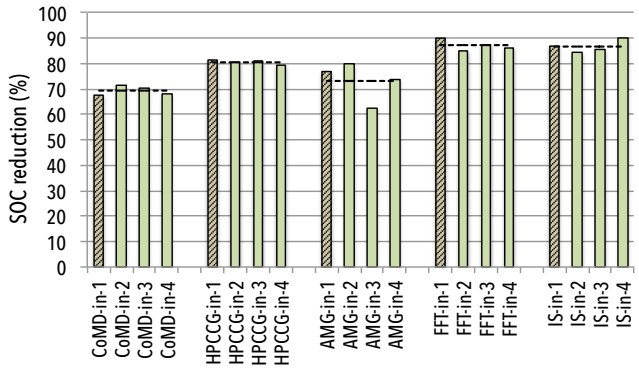


Figure 9: SOC reduction as input is varied. Input 1 is used to train IPAS. The dashed lines shows average values.

Table 5: Application inputs

Code	Input 1 (used for training)	Inputs 2, 3, 4
CoMD	problem size: $nx = ny = nz = 20$	30, 40, 50
HPCCG	problem size: $nx = ny = nz = 50$	75, 100, 125
AMG	problem size: $10K \times 10K$	$15K, 22K, 30K$
FFT	matrix size: 8K	16K, 32K, 64K
IS	Class: S (number of keys: 65K)	W (1M), A (8M), B (33M)

6.5 Input Variation

We now vary the input of the codes to see how it affects IPAS performance. A common practice in HPC applications is to test codes with a small input (usually at small scale) and to execute production runs with larger inputs (usually at large scale). In these experiments, we train IPAS with a base input—the same we used in all previous experiments—and test it on three new larger inputs (see Table 5). We then measure the percentage of SOC reduction using the optimal configuration for each code. The results are shown in Figure 9. We observe that, except for AMG, the percentage of SOC reduction is comparable to the one we get with Input 1. One reason for the variability in AMG is that the weak scaling in this example leads to different sparsity patterns on coarse levels of the hierarchy. As a result, the code execution time varies on different levels of the AMG hierarchy depending on the size of the simulation and the location of the fault. This yields small changes in the level of SOC reduction. As a result, we expect smaller input sensitivity in AMG using a larger SVM training set at the cost of more fault injection runs. However, we leave this investigation to future work.

6.6 Training and Duplication Time

Table 6 shows training time (Step 3 in IPAS), and classification and duplication time (Step 4). The training time is almost constant since we use the same input size in all the codes: 2,500 instruction samples. The data collection time (Step 2) depends on the execution time of the parallel fault-injection jobs, which depend on the system load and scheduling policies. If all fault-injection trials run in parallel, this time is close the application execution time.

Table 6: Training and duplication time

	CoMD	HPCCG	AMG	FFT	IS
Training time (sec)	30.07	30.12	30.39	29.98	30.01
Duplication time (sec)	5.64	6.73	2.33	0.68	1.63
Total time (sec)	35.71	36.85	32.72	30.66	31.64

7. Discussion

Having described our evaluation in detail, we now describe the practical implications and limitations of our approach.

Verifiable computations. We assume that the output of scientific codes can be verified to determine whether SOC occurred or not. This is required *only* to train our machine learning classifier (in Step 3 of Figure 1), and not necessarily required in production runs. While we believe that most scientific algorithms allow verification, which can be done for example by comparing the approximate solution to a known solution, we acknowledge that in some scientific problems this is difficult (if not impossible). To illustrate this, consider the Navier-Stokes equations, which describe the motion of viscous fluids [24]. Although they are widely used in scientific simulations, it has not yet been proven that in three dimensions solutions always exist [16]. This complicates verification since it is impossible to use the exact solution to determine if SOC occurred when a fault is injected. For these cases, IPAS would not be appropriate, unless a relaxed methodology for output verification is used that does not rely on exact solutions.

Training time for large codes. To train our classifier we use examples that are generated by fault injection. For large codes, this could be time consuming. To ameliorate this issue, one can split a large code into small kernels or modules and then use IPAS to independently protect each of them, for example, using parallel jobs to generate training examples.

8. Related Work

Redundancy techniques. Redundantly executing code and comparing the information that the copies generate has been used for decades for fault detection. For single-node computations, two common approaches exist: *Lockstepping* [35], in which redundant copies of a program check the state in every computation cycle, and *Redundant Multithreading (RMT)* [28, 31], in which comparisons are done only for sets of committed instructions in independent execution threads. Although these techniques can reduce SDC (and consequently SOC), they can potentially add significantly hardware and power consumption overheads, which prohibit their use in high-end HPC systems.

Software instruction duplication. A technique that provides redundancy without independent threads of execution is software-based instruction duplication, which has been shown by Reis et al. [32] as SWIFT. It duplicates computation instructions and adds comparison instructions at synchronization points to detect faults. Since SWIFT can potentially incur high runtime overheads, Shoestring [17] has been pro-

posed to probabilistically reduce the overhead of instruction duplication by only protecting non-symptom-generating instructions, which are selected using data-flow analysis and hardware-specific information (e.g., knowledge of exception-generating instructions). A downside of these approaches is that they overprotect since they do not consider SOC errors as a subset of SDC errors.

Reduction of output corruption. Recent work has been proposed to add compiler-level checks in the application to detect SDC that affects the output of computations. For example, Thomas and Pattabiraman [37] proposed heuristics to detect *Egregious Data Corruptions* (EDCs) when outcomes deviate significantly from a user-defined fidelity metric, and Khudia and Mahlke [23] proposed heuristics to detect *Unacceptable Silent Data Corruptions* (USDCs). This work is similar to ours: the concepts of EDCs and USDCs are similar to SOCs and they are both LLVM-based solutions. The main difference between these approaches and ours is that they target soft computations (e.g., those used in multimedia applications), whereas we focus on parallel scientific computations, which can be soft or hard computations.

Placement of checks and detectors. Previous work has also studied the placement of application-level detectors. Hari et al. [19] propose detectors for SDC that are designed by analyzing the assembly code of particular programs. Yu et al. propose ESoftCheck [42], a data-flow analysis technique to remove non-vital checks when detecting errors. Although these techniques can reduce SDC at low cost, they potentially require more manual intervention than techniques like IPAS or Shoestring, which aim at providing automatic compiler-assisted code transformations.

Machine learning. Machine learning (ML) has been used before by the SDCTune technique [26] to improve the performance of instruction duplication. Although SDCTune and IPAS have a similar goal (to selectively protect instructions) there are two main differences between them. First, IPAS uses fault injection in the target code to train a ML model, whereas SDCTune trains a model using fault-injection traces from different codes or benchmarks. Second, and perhaps the main difference between them, ML is used in different ways since SDCTune uses a classifier to classify values of instructions (stored values and comparison results), whereas IPAS uses a classifier to decide whether or not to protect an instruction. Machine learning has also been used to improve the effectiveness of compiler transformations and optimizations [4, 36]. Like IPAS, these techniques use data of static features to train a classifier or a regression function to make predictions at compile time.

9. Conclusions

As larger and more complex supercomputers are built on the path to exascale computing, techniques to mitigate SOC become increasingly important due to a potential increase of soft error rates. This paper proposes a user-guided technique to reduce SOC in scientific codes while maintaining low runtime overheads, high scalability, and portability. Our work demonstrates the use of machine learning as an effective approach to determine the minimal set of instructions that require duplication to avoid SOC in scientific applications. In contrast to previous works that use hardware-specific heuristics and that often overprotect codes, we use an application-focused solution that adapts to specific demands in terms of protection. This, in turn, minimizes runtime overhead. We implement our technique in the IPAS framework using LLVM, and show its benefits in five representative HPC workloads. We demonstrate IPAS scalability in MPI jobs, and found that it can reduce SOC by up to 90% with slowdowns between $1.04\times$ and $1.35\times$.

Acknowledgments

We would like to thank Karthik Pattabiraman and the anonymous reviewers for constructive suggestions and comments. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 and supported by Office of Science, Office of Advanced Scientific Computing Research (LLNL-CONF-669696).

References

[1] CoMD Proxy App. http://www.exmatex.org/comd.html.

[2] HPCCG Mini Application. https://mantevo.org/packages.php.

[3] Intel OpenMP Runtime Library. https://www.openmprtl.org/.

[4] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 295–305, 2006.

[5] Z. Alkhalifa, V. S. Nair, N. Krishnamurthy, and J. A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):627–641, 1999.

[6] R. Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, 2005.

[7] C. M. Bishop et al. *Pattern recognition and machine learning*, volume 4. springer New York, 2006.

[8] J. Calhoun, L. Olson, and M. Snir. FlipIt: An LLVM Based Fault Injector for HPC. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 547–558. 2014.

[9] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 2009.

[10] C.-C. Chang and C.-J. Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[11] C.-L. Chen and M. Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2):124–134, 1984.

[12] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[13] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3): 63–73, 1991.

[14] T. J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. *IBM Microelectronics Division*, pages 1–23, 1997.

[15] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[16] C. L. Fefferman. Existence and smoothness of the navier-stokes equation. *The millennium prize problems*, pages 57–67, 2000.

[17] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 385–396, 2010.

[18] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[19] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2012.

[20] R. Hegde and N. R. Shanbhag. Soft digital signal processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(6):813–823, 2001.

[21] G. Hripcsak and A. S. Rothschild. Agreement, the F-measure, and reliability in information retrieval. *Journal of the American Medical Informatics Association*, 12(3):296–298, 2005.

[22] J. Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, Feb. 2011.

[23] D. Khudia and S. Mahlke. Harnessing soft computations for low-budget fault tolerance. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 319–330, Dec 2014.

[24] O. A. Ladyzhenskaya and R. A. Silverman. *The mathematical theory of viscous incompressible flow*, volume 76. Gordon and Breach New York, 1969.

[25] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: quantified error and confidence. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 502–506. IEEE, 2009.

[26] Q. Lu, K. Pattabiraman, M. S. Gupta, J. Rivers, et al. SDCTune: a model for predicting the SDC proneness of an application for configurable protection. In *2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10. IEEE, 2014.

[27] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, 1979.

[28] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *29th Annual International Symposium on Computer Architecture 2002*, pages 99–110. IEEE, 2002.

[29] M. T. Musavi, W. Ahmed, K. H. Chan, K. B. Faris, and D. M. Hummels. On the training of radial basis function classifiers. *Neural networks*, 5(4):595–603, 1992.

[30] N. Oh, S. Mitra, and E. J. McCluskey. ED 4 I: error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, 51(2):180–199, 2002.

[31] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 25–36, 2000.

[32] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[33] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(4):366–396, 2005.

[34] J. W. Ruge and K. Stüben. Algebraic multigrid. In *Multigrid methods*, volume 3 of *Frontiers in Applied Mathematics*, pages 73–130. SIAM, Philadelphia, PA, 1987.

[35] J. Somers, F. Director, and S. Graham. Stratus ftserver–intel fault tolerant platform. In *Rap. tech. Intel Developer Forum, Fall*, 2002.

[36] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 77–90, 2003.

[37] A. Thomas and K. Pattabiraman. Error detector placement for soft computation. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[38] N. J. Wang and S. J. Patel. Restore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3):188–201, 2006.

[39] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of ISCA '04*, 2004.

[40] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[41] P. Wu and T. G. Dietterich. Improving SVM accuracy by training on auxiliary data sources. In *Proceedings of the Twenty-first International Conference on Machine Learning (ICML)*, page 110. ACM, 2004.

[42] J. Yu, M. J. Garzaran, and M. Snir. ESoftCheck: Removal of Non-vital Checks for Fault Tolerance. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 35–46, 2009.