LADR: Low-cost Application-level Detector for Reducing Silent Output Corruptions

Chao Chen
Georgia Institute of Technology
chao.chen@gatech.edu

Greg Eisenhauer
Georgia Institute of Technology
eisen@gatech.edu

Matthew Wolf
Oak Ridge National Lab
wolfmd@ornl.gov

Santosh Pande
Georgia Institute of Technology
santosh.pande@gmail.com

ABSTRACT

Applications running on future high performance computing (HPC) systems are more likely to experience transient faults due to technology scaling trends with respect to higher circuit density, smaller transistor size and near-threshold voltage (NTV) operations. A transient fault could corrupt application state without warning, possibly leading to incorrect application output. Such errors are called silent data corruptions (SDCs).

In this paper, we present LADR, a low-cost application-level SDC detector for scientific applications. LADR protects scientific applications from SDCs by watching for data anomalies in their state variables (those of scientific interest). It employs compile-time data-flow analysis to minimize the number of monitored variables, thereby reducing runtime and memory overheads while maintaining a high level of fault coverage with low false positive rates. We evaluated LADR with 4 scientific workloads and results show that LADR achieved $> 80\%$ fault coverage with only $\sim 3\%$ runtime overheads and $\sim 1\%$ memory overheads. As compared to prior state-of-the-art anomaly-based detection methods, SDC achieved comparable or improved fault coverage, but reduced runtime overheads by $21\% \sim 75\%$, and memory overheads by $35\% \sim 55\%$ for the evaluated workloads. We believe that such an approach with low memory and runtime overheads coupled with attractive detection precision makes LADR a viable approach for assuring the correct output from large-scale high performance simulations.

HPDC'18, June 11–15, 2018, Tempe, AZ, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5785-2/18/06…$15.00
https://doi.org/10.1145/3208040.3208043

CCS CONCEPTS

• **Computer systems organization → Reliability**;

KEYWORDS

Resiliency, Transient Fault, Soft Error, Silent Data Corruption, SDC, Fault Tolerance, Exascale Computing

1 INTRODUCTION

High reliability is fundamental to the efficient utilization of hardware resources and to user confidence in program results. Unfortunately, while the new processor architectures continue to boost performance with higher circuit density, smaller transistor size and NTV operations, the hardware is projected to be more susceptible to transient faults caused by e.g., particle strikes and heat fluxes.

Transient faults occur with a higher frequency in large scale HPC systems simply because of the sheer number of assembled components [5, 20]. Oliveira et al. [28] projected that a hypothetical exascale machine with 190,000 cutting-edge Xeon Phi processors would experience daily transient errors even though their memory areas are protected with ECC. Some transient faults will be masked by hardware mechanisms or produce obvious failures (e.g., segmentation faults). But more dangerously, others (e.g., those manifested from units unprotected by ECC) could corrupt application states without any warnings (SDCs), and lead to incorrect scientific output [11, 24].

Complete and comprehensive SDC detection requires the duplication of computing through either hardware or software redundancy [19, 23, 26], but the overheads required (~2× resources), and the fact that fault-free operations remain the common case despite the increasing transient fault threat [25], mean that such techniques have seen limited adoption in scientific computing. Fortunately, many scientific applications can tolerate some errors in their outputs [22, 28], as long as the errors don't introduce new data features. This allows the HPC community to trade-off fault coverage for the performance, and has motivated the development of anomaly-based detection methods [14, 18, 21]. These methods seek to exploit characteristics implicit in many scientific applications, such as those which iteratively simulate changes in physical properties, in order to determine when a calculated value has fallen outside its 'expected' range based upon either prior or neighboring values. While applying such techniques to **all** data in an application would be impractical, it is viable to limit its use to only "variables of scientific interest" [14] (e.g., variables that are actually output for further analysis, or that are used in checkpoint/restart. We'll call them the 'crucial variables'.) with relatively low overheads. However, many scientific applications i.e., Parallel Ocean Program (POP), have dozens of crucial variables, and even when these techniques are limited to that set of data, overhead imposed are still significant, perhaps too high for the techniques to gain acceptance by the scientific community.

In this work, we seek to reduce these overheads by exploring correlations among crucial variables and data points. In particular, we present LADR, a light-weight anomaly-based approach to protect scientific applications against SDCs. LADR shares a similar detection model to that in [3, 14]. It is built upon those prior works by introducing compiler techniques to detect and utilize the data-flow of the application in order to limit monitoring overhead. LADR reduces the overheads primarily through minimizing the number of monitored variables. For a given set of crucial variables C, LADR monitors a smaller set of variables D, such that if variables in C are contaminated by SDCs, variables in D will also be contaminated; or put differently, SDCs will be definitely propagated from C to D. We name variables in D as **sink variables** of C. LADR finds D mainly leveraging compile-time data-flow analysis. The similar idea was also presented in [3], but without reference to compile-time analysis. In addition, for each variable in D, LADR also applies a data grouping technique to further reduce runtime and memory overheads, which is not explored in prior studies [3, 14]. LADR is designed to detect SDCs in crucial variables. SDCs in control variables are not covered unless they are propagated as corruptions to crucial variables. The paper makes the following contributions:

(1) we proposed a methodology to minimize runtime and memory overheads for anomaly-based SDC detection techniques.
(2) we designed and implemented the LADR based on the LLVM framework, and supports a majority of scientific applications written in C/C++ and Fortran. Despite some limitations that need to be refined, our prototype of LADR still presents one step towards building application-level SDC mitigation frameworks.
(3) We evaluated LADR with 4 representative scientific workloads including GTC-P, POP, LAMMPs and miniMD. We find that LADR is able to protect them from influential SDCs with as low as $\sim$ 1% memory overheads, and $\sim$ 3% runtime overheads. As compared to the state-of-the-art anomaly-based detection technique, LADR achieved comparable faulty coverage, but reduced runtime overheads by $>$ 20% and memory overheads by up to 55%.

The evaluation results suggest that LADR is a promising solution for scientific applications that can tolerate small numerical fluctuations in their outputs. Certainly, LADR has its constraints for applications in which the accuracy of results is the users' primary concern. We believe, however, LADR is attractive in many situations since many scientific simulations are approximate computing to physical phenomenons and can tolerate some small errors [10, 22, 30] in their output. 2

2 RELATED WORK

The resiliency issue has long been seen as an obstacle to productive exascale systems [1, 5, 15], and has attracted attention in prior work [2, 4, 9, 12]. In this section, we briefly survey prior approaches to the problem, including alternatives to LADR's anomaly-based approach.

2.1 Anomaly-based techniques

Anomaly-based detection methods mainly exploit the characteristics of the application data to detect SDCs. Yim et al. [31] computed the histogram of application data to detect outliers in conjunction with temporal and spatial similarity. Di et al. [14] characterized features of applications' outputs, and exploited the smoothness of their outputs across time dimension for detecting SDCs.

These techniques generally have three phases: 1) predicting the next expected value in the time series for each data point; and 2) determining a bound, e.g. normal value interval, surrounding the predicted value. 3) detecting possible SDCs by observing whether the observed value falls outside the bound. There are 4 possible situations as depicted in Figure 1. Obviously, as compared with complete computational redundancy, these methods trade off fault coverage for performance. They could miss SDCs if the selected bound ρ (shown in Figure 1(a)) or the prediction error ε (shown in Figure 1(b)) is larger than the impact of SDCs. On the other hand, it would incur false positives if ρ is smaller than ε. A case depicting a successful detection is shown in Figure 1(d).

As mentioned previously, our work shares a core similarity to that of Di et al. [14] with respect to the basic approach of detecting SDCs through analysis of predicted values, and

Berrocal et al. [3] introduces the concept of exploiting correlation between variables. LADR extends these prior work by proposing compiler techniques to exploit dataflow-based variable correlation, and in exploring point grouping for additional overhead reduction.

2.2 Redundancy-based approaches

Instruction-level redundancy is an important approach to software-resiliency [8, 27, 29, 32]. EDDI [27] and SWIFT [29] are two well-known studies in this space. EDDI duplicated all instructions and "checking" instructions were inserted right before storing a register value back to memory or determining the branch direction. SWIFT [29] optimized overheads of EDDI through an enhanced control flow mechanism, but it still incurred > 83% overheads. While complete duplication of computation is the only close-to-foolproof defense against SDCs, we believe that is too expensive for widespread use and that the LADR approach, which leverages more of an understanding of application semantics and is amenable to compiler-based techniques with lower overheads has more potential for general use.

On the other hand, process-level replication is also explored in [17, 19, 26]. Here, multiple instances of applications are compared and divergences are taken to be evidence of an SDC. RedMPI [19] ran a shadow process for each MPI rank and redesigned MPI communication system to duplicate message passing for both shadow process and principle process. SDCs were detected through comparing messages between replicas. ACR [26] replicated processes on different nodes and detected SDCs by comparing check-pointing data from each replica. Process-level replication generally doubles the computing resources required for computation, a level of overhead unlikely to be acceptable for many large-scale scientific applications.

2.3 Algorithm-based fault tolerance

There are also several studies [6, 7, 13, 16] focusing on designing resilient data structures and algorithms. Chen[6] examined the block row data partitioning scheme for sparse matrices, which were then utilized to recover critical data without checkpointing. Du et al.[16] constructed a column/row checksum matrix for matrix computations, such that SDCs can be detected by scanning partial product matrix and recovered with the checksum matrix. These algorithm-specific methods are highly specialized and as such focus on specific computational kernels, which are normally a small part of scientific applications. Faults that happen outside the computational kernel but still affect the core computation may be not detectable in such cases. In contrast, we believe the anomaly-based approach to have broad applicability within the space of time-step-based iterative solvers that are our current interest and they can protect applications against SDCs that happened anywhere during the run, as long as they lead to a significant corruption of output variables.

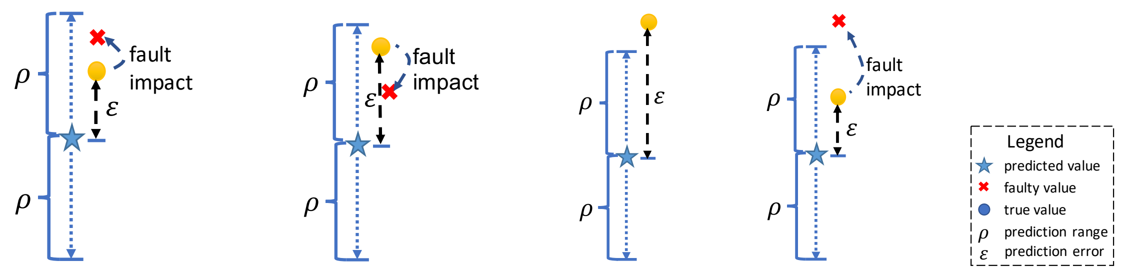
3 DESIGN OF LADR

This section details the design of LADR. We will first introduce two observations that motivated the design, and then present design details.

3.1 Motivation: Propagation of SDCs

Although by no means universal, time-step based iterative solvers are an important component in many scientific simulations, with each time-step implementing a time point in the real world. These applications simulate real-world phenomena by solving a system of differential equations on points (e.g., grids, particles). Each point could be associated with several *states* (i.e., speed, temperature, energy etc.), which are crucial variables of applications. Applications proceed along the temporal dimension to update *states* for each point. The result of each time-step will be taken as the input to the next time-step. Climate models are typical examples. They treat the atmosphere as a cubic box divided into grids, and each grid represents a geometrical area on the earth. Climate parameters, e.g., temperature, are computed by solving atmosphere dynamic equations for each grid. To update a specific state for a grid, the values of other states of other grids would be used. Due to this nature, these applications present two general characteristics that inspired the design of LADR: Firstly, SDCs propagate among crucial variables during the iterative updates. For example, Table 1 shows that, for GTC-P and POP, the contamination of multiple crucial variables could result from a single fault injected into a data element of crucial variable listed in the first column. Therefore, it is possible to detect SDCs by only monitoring a subset of crucial variables. For GTC-P, specifically, it is possible to protect the variables *z0*, *z1*, *z2*, *z3*, *z4*, and *z5* by only monitoring the variable *moments*.[1] In observing this, LADR shares a similarity with [3], which also notes that error correlation between variables can be exploited to reduce SDC monitoring overhead, but does not specially propose a method for establishing this correlation. The variables to be monitored must be carefully selected such that SDCs which occur in unmonitored variables will be captured. LADR leverages data flow analysis to identify these variables since the data-flow among crucial variables imply potential SDC propagation path; Secondly, multiple data points of each contaminated crucial variable might be impacted by a single SDC, as shown in Figure 2. This inspired us to group data points

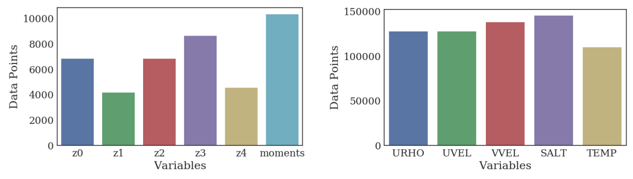
[1] Each of these variables store attributes of particles, e.g., weight and velocity. The names are as they appear in GTCP source code.



(a) false negative due to large ρ (b) false negative due to high ε (c) false positive (d) valid and detectable

Figure 1: Detection model of anomaly-based techniques (figure derived from [14])

to further reduce the overheads. As shown in our evaluation, the grouping could also improve the prediction accuracy for the predictor in Phase I, therefore, making the detector more sensitive to SDCs.



(a) GTC-P **(b) POP**

Figure 2: Average number of contaminated data points within 5 time-steps after a fault injection.

3.2 Overview of LADR

LADR is a lightweight application-level SDC detector for iterative scientific applications. It consists of two components: 1) ***Analyzer***–a static compile-time analysis tool for identifying sink variables for a given set of crucial variables; and 2) ***Protector***–a runtime library for conducting anomaly detection. Figure 5 depicts the process of utilizing LADR to protect scientific applications. First, given the *crucial variables* set, the ***Analyzer*** is used to determine their sink variables. The ***Analyzer*** identifies sink variables by building a ***data-flow graph***, which depicts potential SDC propagation paths among all crucial variables. The ***data-flow graph*** is a directed graph with each node representing a crucial variable, each edge representing a potential propagation direction between two connected crucial variables, and the weight on each edge representing relative execution orders of the statement defining the propagation. Figure 4 gives an example of data-flow graph among 5 variables (f is an alias to `e`) for the code listed in Figure 3. Afterward, original source files are modified to apply ***Protector*** on identified sink variables. ***Protector*** is inserted at the end of the main loop of scientific applications and invoked by every MPI rank on every iteration.

```
#include <stdio.h>
void add(double x[], double y[], double z←
     [], int size) {
  for (int i = 0; i < size; i++)
    z[i] = x[i] + y[i];
}

void mul(double x[], double y[], int factor←
     , int size) {
  for (int i = 0; i < size; i++)
    y[i] = x[i] * factor;
}

int main(int argc, char **argv) {
  double *a, *b, *c, *d, *e, *f, sum = 0;
  int i, size;
  a = (double *)malloc(size * 8);
  ...

  add(a, b, c, size);
  for (i = 0; i < size; i++)
    sum += c[i];
  sum = sqrt(sum);
  mul(b, d, sum, size);
  f = e;
  add(c, d, f, size);
  add(f, d, d, size);
  output(a,b,c,d,e);
}
```

Figure 3: Example code

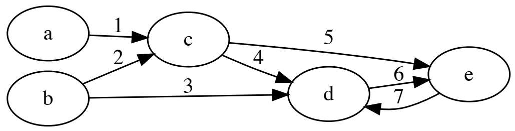


Figure 4: Data-flow graph for the example code in Figure 3

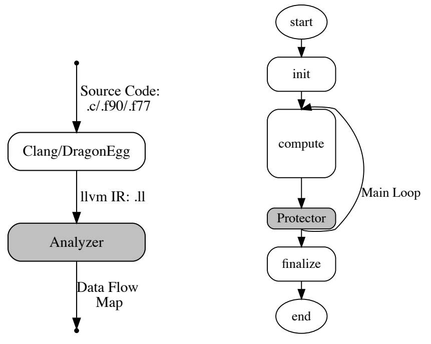
Table 1: Correlation among crucial variables. It is based on controlled fault injection experiments. Faults are injected to variables in the first column and then the outputs of other variables are checked against the output of fault-free run. Each run of the application performs one injection to one variable.

(a) GTC-P Particle Data Variables

injected \ impacted	z0	z1	z2	z3	z4	z5	moments
z0	✓	✓	✓	✓	✓	✓	✓
z1	✓	✓	✓	✓	✓	✗	✓
z2	✗	✗	✓	✗	✓	✗	✓
z3	✓	✓	✓	✓	✓	✓	✓
z4	✗	✗	✗	✗	✓	✗	✓
z5	✓	✗	✗	✓	✓	✓	✓

(b) POP TAVG Output Variables

injected \ impacted	SALT	SALT2	TEMP	UVEL	VVEL
FW	✗	✓	✓	✓	✓
Gradpy	✗	✗	✓	✓	✓
RHO	✗	✗	✓	✓	✓
TFW	✗	✗	✓	✗	✓
STF	✗	✗	✓	✗	✓
SMF	✗	✗	✗	✗	✓



(a) stage I **(b) stage II**

Figure 5: An overview of utilizing LADR

3.3 Analyzer: building data-flow graph for identifying sink variables

The ***Analyzer*** extracts potential SDC propagation paths by leveraging static data-flow analysis based on the observation that SDCs would propagate from one variable – saying b, to another – saying a, only if the calculation for a directly or indirectly uses the value of b. ***Analyzer*** focuses on the main computation codes – the main loop – of scientific applications. For simplicity, the ***Analyzer*** used the following observations for scientific applications:

(1) crucial variables of these applications are life-time-long arrays (a memory space). At the program level, they are either defined as global arrays or allocated during the initialization phase of the applications but not deallocated until the end of execution.
(2) Parallelization techniques (MPI and OpenMP) have no/limited impact on data-flows among crucial variables, because each process/thread conducts the same computation but on a different portion of data.[2] Hence, ***Analyzer*** ignores all MPI functions calls and OpenMP primitives.
(3) crucial variables are updated inside loops, and related loop bodies will not be skipped because of false initial loop conditions.

Analyzer works on LLVM IR code, a light-weight and low-level intermediate representation of programs. This makes LADR relatively independent of programing languages utilized by scientific applications, and it can support a majority of existing scientific applications (if not all) that are normally written in either C/C++ or Fortran. In LLVM IR code, each memory access is explicitly issued through either a load instruction (*LoadInst*) to read data from a memory location, or a store instruction (*StoreInst*) to update a memory location. Therefore, each assignment operation in source codes would correspond to several *LoadInst* instructions to read data for RHS operands, a set of related computation instructions, and one *StoreInst* instruction to update the memory with final result, as shown in Figure 6.

3.3.1 Extracting data-flow among variables. Based on the above observation, ***Analyzer*** extracts data-flow among crucial variables by analyzing *StoreInst* instructions, which have two operands named as *source* and *destination*, simply assuming that the array in *destination* (write to) covers arrays involved in *source*. For each operand of *StoreInst*, LADR leverages def-use chain to backwardly extract involved variables through looking for *LoadInst*. There are 5 possible situations:

(1) **the source operand is a pointer**, as shown in line 23 in Figure 3. In this case, ***Analyzer*** considers the *destination* operand as an alias to the *source* operand,

[2] There are two principal questions WRT how MPI and OpenMP might impact LADR: 1) do they add additional dependencies for the ***Analyzer*** to track, and 2) how they impact SDC propagation. We find that for the scientific simulations we have studied, which are largely spacially decomposed, communication abstractions don't add dependencies between state variables that are not already present in the code. Second, while it is possible for SDCs to be communicated between processes, they will generally be picked up wherever they cause a significant disruption to expectations since the **protector** is embodied in every MPI rank. So these primitives get no special handling in LADR.

```
%2 = load i32* %i, align 4
%idxprom = sext i32 %2 to i64
%3 = load double** %x.addr, align 8
%arrayidx = getelementptr inbounds double* ←-
    %3, i64 %idxprom
%4 = load double* %arrayidx,align 8
%5 = load i32* %i, align 4
%idxprom1 = sext i32 %5 to i64
%6 = load double** %y.addr, align 8
%arrayidx2 = getelementptr inbounds double*←-
     %6, i64 %idxprom1
%7 = load double* %arrayidx2,align 8
%add = fadd double %4, %7
%8 = load i32* %i, align 4
%idxprom3 = sext i32 %8 to i64
%9 = load double** %z.addr, align 8
%arrayidx4 = getelementptr inbounds double*←-
     %9, i64 %idxprom3
store double %add, double* %arrayidx4, ←-
    align 8
```

Figure 6: LLVM IR Code for the loop of add

and maintains an pointer-to-pointer aliasing map for the ongoing analyzed function for future reference.

(2) **the source operand is an array element and the destination variable is a scalar variable**, as shown in line 20 in Figure 3. In this situation, ***Analyzer*** will maintain a scalar map for the ongoing analyzed function to temporary record propagation path information among scalar variables and arrays for future reference, since it could define an indirect propagation among crucial variables.
(3) **the source operand is a scalar variable and the destination variable is an array element**, as shown in line 9 (*factor*) in Figure 3. For this case, ***Analyzer*** will first check the scalar map using the source operand. If there exists an entry, it will retrieve related *source* arrays from the scalar map, and record the propagation information between the *destination* and each *source* array into propagation map.
(4) **both source and destination operands are scalar variables**, as shown in line 13 in Figure 3. This is similar to case 3. LADR will first check whether the *source* operand has an entry in the scalar map; if yes, it will register the destination operand into the map with the same content. Otherwise ***Analyzer*** will simply skip the instruction.
(5) **both source and destination operands are array elements**, as shown in line 4 in Figure 3. This case defines a direct propagation among crucial variables. The ***Analyzer*** will simply register the propagation information in the propagation map (aliasing map would be checked to retrieve the actual variables before the actual registration).

3.3.2 Function Calls. To build the data-flow graph for whole application, LADR needs to handle function calls. For each *CallInst*, ***Analyzer*** takes following actions depending on the type of the callee function:

(1) **memory allocations**, e.g., malloc/alloc. For each memory allocation, ***Analyzer*** assigns an id internally to virtually represent the allocated memory region and registers it in global variable tables.
(2) **basic math computation**, e.g., *sqrt* and *max*. For these functions, ***Analyzer*** extracts data-flow information among their arguments and return values leveraging knowledge of library APIs.
(3) **application defined functions**. For these functions, ***Analyzer*** will dive into function bodies to analyze each *StoreInst*, essentially *inlining* the subroutine to handle parameters and return values.

3.3.3 Conditional Control/Data Flow. Generally the ***Analyzer*** takes a conservative approach to conditional or runtime determined control and data flow, e.g. evaluating both paths in an *if* expression and taking the superset of those contributions as the overall data flow. This is most effectively done by simply ignoring the branch operators in the LLVM IR code.

Overall. To elaborate ***Analyzer*** clearly, we take the example code in Figure 3 to illustrate the entire analyzing process. ***Analyzer*** starts analysis in the main function. It considers all malloc statements as defining variables. For the first *add* function call statement (line 18), it inlines *add* by aliasing *x* to *a*, *y* to *b*, *z* to *c*, and *s* to *size*. All references to *x*, *y*, *z* and *s* are replaced with *a*, *b*, *c*, *size*, so we will get propagation paths among *c*, *a* and *b* instead of *z*, *x* and *y*. For line 20, we will get *sum* covers *c*. Since *sum* is scalar, it will be registered in scalar map. In line 21, since *sqrt* is a standard mathematical function, we can easily get that *sum* covers itself. Then for *mul* function call, it will do the inline procedure as did to *add* in above. In line 4, we will get propagation paths among *d*, *b* and *sum*. Since *sum* is now in scalar map with a propagation path from *c*, we will update the entry for *d* in the propagation map with an edge from *c*. The above procedures will be repeated for the following function calls.

3.4 Protector: anomaly-based detection

LADR ***Protector*** shares a similar detection approach to extant anomaly-based detection methods[3, 14] (see Section 2). However, it distinguishes itself in two aspects: 1) it groups data points for each monitored variable, and works on the feature array constructed by extracting a data feature from each group. 2) it detects SDCs with two metrics from the perspectives of the number of contaminated data points and error magnitudes. LADR currently supports three data features for grouping, including mean, standard deviation, and

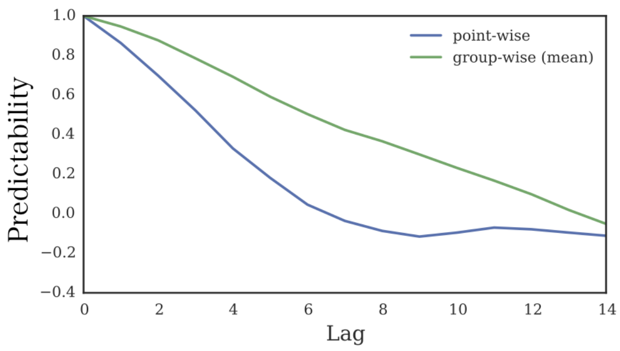


Figure 7: Predictability (measured with pacf) of a point and the group (constructed with 8-neighbor points) in which the point resides. The mean value of the group is used.

entropy, as well as three predictors: Linear Curve Fit (LF), Quadratic Curve Fit (QF) (from [14]), and AutoRegression (AR). LADR seeks to select the best predictor, feature, and grouping strategy based on a learning window, such as the first few time steps of a run assuming there are no SDCs in these steps.

3.4.1 Data points grouping. Crucial variables of scientific applications are typically huge arrays with millions or even billions of data points, therefore, it would still incur non-negligible runtime and memory overheads with current point-wise predictions, since they need to preserve the history data and predict the value for each data point at each time-step. LADR proposes to mitigate this issue by grouping data points, which is motivated by the observation that a single SDC could contaminate multiple data points (see Section 3). Intuitively, data points grouping could increase the potential of diluting SDCs if the group size is too large, therefore reducing fault coverage. Based on the intuition that the global/regional state could be more stable and predictable than the point-wise state (as shown in Figure 7), LADR employs a heuristic-based grouping algorithm to achieve a balance between fault coverage and resulting overheads. In this heuristic-based grouping algorithm, the predictability for each data point(C_p) and the global array (C_∂) are first calculated with partial correlation function (pacf), and data points are now roughly divided into two groups based their predictability: 1) points whose C_p are larger than C_∂; and 2) points whose C_p are less than C_∂. For points whose C_p are larger than C_∂, they are simply divided into groups with minimum group size, which is a user-specified parameter (we set it as 8 in our experiments). For points whose C_p are smaller than C_∂, they are merged with their neighbors recursively until the predictability of the group is around C_∂ or group size is larger than a user-specified maximal group size. For each group, the feature leading to higher predictability is selected as representative of the group. In addition, domain-specific knowledge can be leveraged for grouping too. For example, in MD codes, the velocity information (V) of each atom is encoded with three data points, representing velocities in the x, y, and z directions. Thus, a grouping strategy that calculating the absolute velocity ($\sqrt{x^2 + y^2 + z^2}$) can be applied, which could reduce the overhead by 3×. Such domain-specific grouping can replace the default heuristic-grouping algorithm or be applied together with the heuristic grouping algorithm.

3.4.2 SDC detection. As in [14], LADR detects SDCs by checking the value of monitored variables at the end of each time-step. It works on prediction errors calculated with Eq. 1 (below) for each data point, where P is prediction value, O is observation value and i refers to a data point. Since we have no knowledge about the correctness of observation value, historic mean values of data points are used in our estimates.

$$E(i) = (P(i) - O(i))/mean(hist(i)) \tag{1}$$

Afterward, for each data point, the prediction error range, $[mean(E(i) - \theta \times std(E(i)), mean(E(i) + \theta \times std(E(i))]$, is constructed based on its history prediction errors. θ is initialized with a constant value (we set it to 8 in our experiments), and is updated at runtime with the following equations, where γ is the actual up bound of history prediction errors:

$$\theta = \begin{cases} \frac{\theta+\gamma}{2}, & if \ \ \gamma < 0.5 * \theta \\ 1.5\gamma, & if \ \ \gamma > \theta \end{cases} \tag{2}$$

If the prediction error of a specific data point is out of its range, an SDC is assumed for the point. It is, however, too strict to use the result of a single point to indicate the existence of SDCs, and would incur significant false positives due to prediction variance. To mitigate this issue, LADR detects SDCs based on two metrics. First, for each time-step, we calculate the ratio of data points with out-of-bound prediction errors. An SDC is reported only when the ratio is larger than a threshold. Currently, the threshold was derived statistically from the learning window. During the learning phase for selecting best predictor, feature and grouping strategy, the out-of-bound ratio R for each time step was also recorded. And the threshold was set as $mean(R) + 5 * std(R)$, or 0.08% in our experiments. This metric is designed based on our second observation, and it is to detect SDCs that could contaminate a large number of data points. Secondly, LADR also checks the statistics (mean and standard deviation) of prediction errors (error magnitudes across all data points) for each time-step. In normal cases, we observe that these statistics are within small bounds, which can be constructed based on the historic data using the same method for constructing prediction error bounds. If a statistic associated with the current

```
GE_Init(MPI_COMM);
GE_Protect(char varname, void *var, int ←
     data_type, size_t size, int buf_size, ←
     float ratio, float impaction);
GE_Snapshot();
GE_PrintResult();
GE_Finalize();
```

Figure 8: GE Protector API

step is out of the normal bound, an SDC is reported also. This metric could help to detect SDCs causing significant errors but affecting too few data points. The recovery procedure will be invoked if an SDC is reported by either one of these two metrics.[3]

4 IMPLEMENTATION

We implemented LADR with two separate components: a LLVM-based compiler-framework—***Analyzer***, and a runtime protector library—***GE***. In particular we implemented ***Anlyzer*** as an independent LLVM pass (~ 1688 LOC) based on LLVM-3.5.2. It analyzes unmodified source code of an application to build data-flow graph among crucial variables. We leverage Clang or DragonEgg to compile C/C++ or Fortran codes into LLVM IR codes (.ll file). Our current prototype of ***Anlyzer*** only generates the data-flow graph, and the sink variables are manually/visually selected by simply picking destination node in the generated graph, e.g., d in Figure 4. We hope to fully automate the entire process in future work.

The GE runtime is developed based on GSL library. Similar to [14], it exposes users 5 API routines as shown in Figure 8. *GE_init* is inserted to the beginning of the application right after *MPI_Init*. *GE_Protect* is inserted before the main loop for each protected sink variable. Each sink variable can be assigned with separate parameters according their data features. *GE_Snapshot* is inserted at the end of main loop, right before output routine. Finally, *GE_PrintResult* and *GE_Finalize* are expected to be inserted right before *MPI_Finalize*.

5 EVALUATION

We evaluated LADR through fault-injection experiments. In this section, we will first introduce the evaluation methodology, and then present evaluation results.

5.1 Evaluation Methodology

We evaluated LADR on a cluster with 32 nodes. Each node is equipped with a 12-core Intel(R) Xeon(R) X5660 (2.80GHz) CPU, 24GB of memory and Mellanox Technologies MT26438 InfiniBand card. Similar to the work in [14], we focused on unexpected data changes (SDCs) caused by transient faults, e.g., bit-flips of the data. We simulated SDCs through application-level fault injections as did in study [11]. We implemented a simple fault injection tool based on GDB's MI interface for this purpose. For sake of easy analysis, the tool injects faults to one MPI rank during each run of applications. In this tool, a fault is identified by a tuple with 4 elements: (***iteration***, ***execution point***, ***target***, ***fault***):

- ***iteration*** and ***execution point*** together determine when the fault will be injected.
- ***Execution point*** is represented in form of *file:line*.
- ***target*** determines where the fault will be injected. It is a specific memory location of applications.
- ***fault*** determines how to corrupt the value of the target. Random bit-flips are used in our evaluation.

We injected faults directly into applications' memory space since it behaves closer to SDCs. We performed one injection per run, and contaminated one data point per injection. ~6000 injections in total were performed. We compared LADR to the "Reference" scheme, in which all of crucial variables in our setups were monitored and point-wise predictor was employed. In contrast, LADR only monitored the identified sink variables (shown in Table 4) and also applied the data point grouping technique. For the "Reference" scheme, the tool developed in [14] was used. We compared them from 3 aspects:

1. *fault coverage (FC)*. It measures how many SDCs are detected by an SDC detector. It is defined by number of detected SDCs over the total injections that contaminated values of crucial variables.
2. *false positive (FP)*. A FP happens when an SDC is mistakenly reported for a fault-free time-step. It is defined as the number of mistakenly reported time-steps over the total number of iterations under the evaluation.
3. *overhead*. It contains runtime overhead incurred by predictions, and memory overheads incurred for storing history data sets.

5.2 Evaluated Workloads and Baselines

We evaluated LADR with the four scientific workloads in Table 2. In our evaluation, we set up the baselines for GTC-P and POP with 6 crucial variables, and for miniMD and LAMMPS with 3 crucial variables. The data size of each crucial variable is shown in Table 3. LADR's ***Analyzer*** identified 1 sink variable for each evaluated workload, as shown in Table 4. After sink variables were determined, application codes were modified to monitor the variable using the GE library. The modification effort involved 25 lines of code changes for GTC-P, miniMD and LAMMPS, as well as 45 lines of code changes for POP, which was minor and acceptable.

[3]This paper doesn't propose new recovery methods, and we assume checkpoint/restart can be used here.

Table 2: Evaluated scientific workloads

Workloads	Descriptions	crucial variables
Gyrokinetic Toroidal Code–Princeton (GTC-P)	C program. A 2D domain decomposition version of the GTC global gyrokinetic PIC code for studying micro-turbulent core transport. It solves the global, nonlinear gyrokinetic equation using the particle-in-cell method.	6 particle data variables: zion0, zion1, zion2, zion3, zion4, zion5
Parallel Ocean Program (POP)	Fortran program. A 3D ocean circulation model designed primarily for studying the ocean climate system. It was used to perform high resolution global ocean simulations to resolve meso-scale eddies that play an important role in the dynamics of the ocean.	6 TAVG output variables: uvel, vvel, salt, temp, rho, salt2
LAMMPS	C++ program. a classical molecular dynamics code, and an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator developed at Sandia National Laboratories	3 molecular/atom property variables: X, F, V
MiniMD	C++ program. A simple, parallel molecular dynamics (MD) code developed at Sandia National Laboratories	3 molecular/atom property variables: X, F, V

Table 3: Per time-step size of evaluated variables

GTC-P		POP		miniMD		LAMMPS	
name	size	name	size	name	size	name	size
zion0	25280 KB	TEMP	5750 KB	V	16384 KB	V	16384 KB
zion1	25280 KB	SALT	5750 KB	F	16384 KB	F	16384 KB
zion2	25280 KB	SALT2	5750 KB	X	16384 KB	X	16384 KB
zion3	25280 KB	UVEL	5750 KB	–	–	–	–
zion4	25280 KB	VVEL	5750 KB	–	–	–	–
zion5	25280 KB	RHO	5750 KB	–	–	–	–

5.3 LADR Analyzer cost

As shown in Table 4, ***Analyzer*** worked more efficiently on GTC-P, miniMD and LAMMPS than on POP. It roughly took 2 ∼ 3 minutes for analyzing GTC-P and miniMD, ∼ 18 minutes for analyzing LAMMPS, but nearly 3 hours for POP. We attribute this issue to two reasons: 1) a larger code base for POP (around 3× of GTC-P and 10× of miniMD), and 2) inefficient IR code generation for Fortran. During our evaluation, We found that the IR code generated from the Fortran program was not as succinct as the IR generated from C/C++. It introduced significantly more branches and virtual functions, which are not shown in the original source code. These features, especially branches, complicate the analysis of ***Analyzer*** because it needs to evaluate each potential execution path. With the ongoing project FLANG, a native Fortran front-end for the LLVM framework, we expect this issue would be mitigated.

Table 4: Analyzer cost

Apps	Source code (LOC)	LLVM IR (LOC)	Sink variables	Analysis time
GTC-P	18,453	61,049	moments(1775KB)	3 mins.
miniMD	4,167	28,028	V(16384 KB)	2 mins.
LAMMPS	415,084	1,655,405	V(16384 KB)	18 mins.
POP	59,678	478,311	TRACER(575KB)	189 mins.

5.4 Fault coverage

Fault coverage is a major metric for measuring the effectiveness of SDC detectors. While LADR aims to optimize the runtime and memory overheads, it should not significantly sacrifice fault coverage.

Figure 9 compares the fault coverage of LADR to the "Reference" scheme for the evaluated workloads. Results of using different grouping strategies are also reported. "LADR-grouping(H)" shows grouping data points leveraging the proposed heuristic grouping algorithm, "LADR-grouping(S)" divided data points evenly using the same number of groups as in "LADR-grouping(H)," and the "LADR" label shows simple point-wise monitoring of the sink variable. As shown in the figure, "LADR" achieved comparable fault coverage as compared to the "Reference" scheme, reducing overhead by monitoring fewer variables at a cost of just a 1% ∼ 4% decrease in fault coverage. Meanwhile, the heuristic grouping algorithm boosts its performance significantly, since it improved the predictability of the feature data, therefore allowing a more accurate detection model. As an example, Figure 10 presents the impact of our heuristic grouping algorithm on the predictability for miniMD. It shows that, the heuristic grouping algorithm effectively removes the data points with less predictability (comparing Figure 10(a) and Figure 10(b)), but didn't blindly increase the group size 10(c). As shown in Figure 1(b), low predictability would lead to lower fault coverage since a large bound was required for tolerating false positives. These results suggest that, by leveraging data-flow information, it's unnecessary to monitor all crucial variables to protect scientific applications from SDCs, and heuristic grouping algorithm can achieve a better balance between group size and prediction accuracy than static grouping.

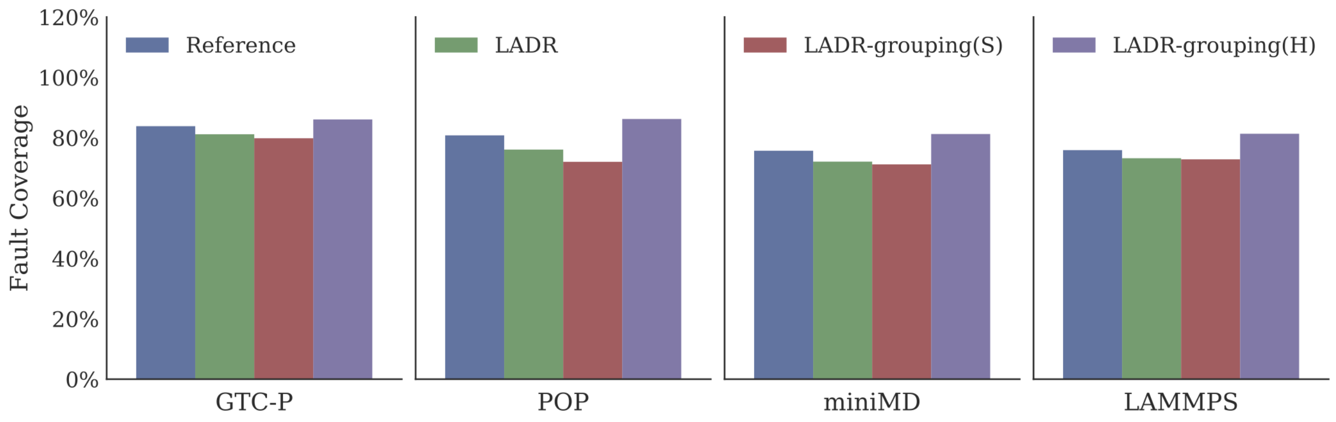
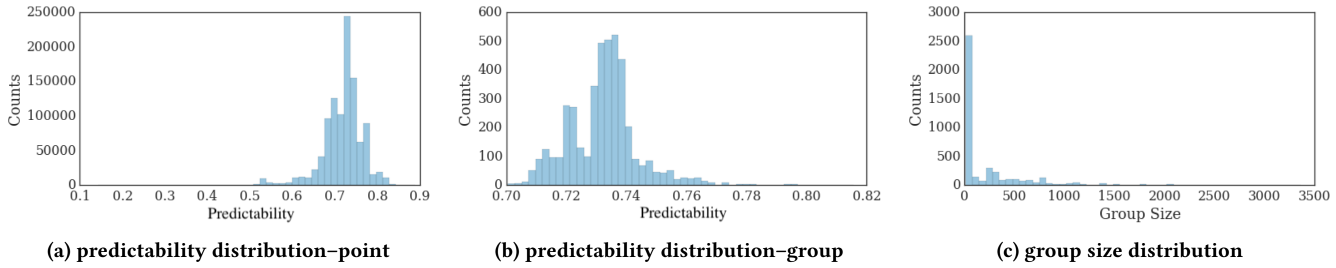


Figure 9: Fault coverage comparison. H – heuristic grouping; S – static grouping



(a) predictability distribution–point **(b) predictability distribution–group** **(c) group size distribution**

Figure 10: Impacts of heuristic grouping. The constructed feature data is more predictable among time-steps. Most groups have a few data points, while static grouping has 245 points for each group. Due to limited space, only the data for miniMD is plotted. Other workloads share a similar observation.

5.5 False positives

The false positive rate is another important metric for measuring the effectiveness of SDC detectors. A false positive would invoke the unnecessary recovery scheme, potentially incurring significant overheads. Therefore, a low false positive rate is required from SDC detectors. Table 5 compares the false positive rates of LADR against the "Reference" scheme. We measured false positive ratio basing on first 1000 time-steps of workloads. LADR achieved slightly lower false positive rate mainly because it monitored fewer variables.

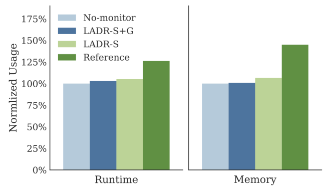
Table 5: False positive rate

	LADR	Baseline
GTC-P	0.2%	0.5%
POP	0.3%	0.7%
miniMD	0.8%	1.1%
LAMMPS	0.7%	0.9%

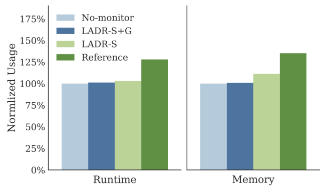
5.6 Runtime and memory overheads

In this section, we evaluated the overheads of LADR. We compared it to the "Reference" scheme and to baseline runs in which no protection is applied to the workloads. Figure 11 presents their overheads normalized to the baseline for each evaluated workload. By monitoring sink variables, LADR reduced runtime overheads by 21% for GTC-P, 25% for POP, 22% for miniMD and 57% for LAMMPS. It also reduced memory overheads for them respectively by 38% for GTC-P, 39% for POP, 24% for miniMD and 28% for LAMMPS. This is mainly because it monitored fewer variables. For GTC-P and POP, it is also because the selected sink variable is smaller than the crucial variables. In addition, the grouping algorithm divided GTC-P into 5230 groups, POP into 3270 groups, miniMD into 4289 groups, and LAMMPS into 10367 groups for data points in each node. This further reduced memory overheads to around 1% for these workloads and runtime overheads to 2.96% for GTC-P, 1.22% for POP, and 9.16% for miniMD, and 11.8% for LAMMPS. These results show that monitoring sink variables can significantly reduce overheads of anomaly-based SDC detectors, and data points grouping can further reduce the overheads to a negligible level.

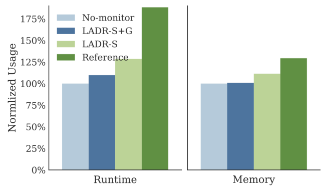
In conclusion, the evaluation results show that LADR significantly reduced runtime and memory overheads as compared with prior methods without sacrificing fault coverage. It would be a promising method for scientific applications



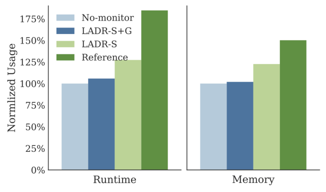
(a) GTC-P



(b) POP



(c) miniMD



(d) LAMMPS

Figure 11: Runtime and memory overheads of LADR with sink variable (S) and data points grouping (G)

that can tolerate some numerical disturbance while being protected from larger SDCs in the majority of their data.

6 CONCLUSION

SDC is an increasingly important concern for the reliability of exascale systems. It is harmful to scientific applications, since it could lead to incorrect scientific insights. Validating a given simulation run to be free of SDCs is a very challenging problem and an effective solution must achieve high fault coverage with limited overheads. In the absence of such validation, the integrity of high fidelity simulations remains questionable on very large scale systems that are prone to such errors.

In this paper, we presented and evaluated LADR, a lightweight application-level approach to protect applications from SDCs using techniques based on compiler analysis. It improves extant anomaly-based techniques by focusing on minimizing runtime and memory overheads primarily through exploiting correlation among crucial variables and data points. We evaluated LADR using application-level fault injection experiments. Results suggest that LADR can protect application from influential SDCs with no more than 8% overheads. For two of evaluated workloads, it only incurs around 2% overheads. LADR demonstrates that it is unnecessary to apply anomaly detection techniques on all crucial variables. Instead, a subset of crucial variables can be identified employing compile-time data-flow analysis.

In future work, we plan to improve LADR by undertaking a more detailed array data-flow analysis and by leveraging the flow properties of amplification of errors.

REFERENCES

[1] Steve Ashby, Pete Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina, and others. 2010. The opportunities and challenges of exascale computing. (2010).

[2] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. 2011. FTI: High performance Fault Tolerance Interface for hybrid systems. In *Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis.*

[3] Eduardo Berrocal, Leonardo Bautista Gomez, Di Sheng, Zhiling Lan, and Franck Cappello. 2015. Lightweight silent data corruption detection based on runtime data analysis for HPC applications. In *Proceedings of International Symposium on High-Performance Parallel and Distributed Computing.*

[4] Robert Birke, Ioana Giurgiu, Lydia Y Chen, Dorothea Wiesmann, and Ton Engbersen. 2014. Failure Analysis of Virtual and Physical Machines : Patterns , Causes and Characteristics. In *Proceedings of International Conference on Dependable Systems and Networks.*

[5] Franck Cappello, Al Geist, B. Gropp, L. Kale, Bill Kramer, and M. Snir. 2014. *Toward Exascale Resilience:2014 Update.* Technical Report.

[6] Zizhong Chen. 2011. Algorithm-based Recovery for Iterative Methods Without Checkpointing. In *Proceedings of International Symposium on High-Performance Parallel and Distributed Computing.*

[7] Zhizhong Chen. 2013. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. In *Proceedings of International Symposium on Principles and Practice of Parallel Programming.*

[8] Z. Chen, R. Inagaki, A. Nicolau, and A. V. Veidenbaum. 2015. Software fault tolerance for FPUs via vectorization. In *Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation.*

[9] Zhi Chen, Alexandru Nicolau, and Alexander V Veidenbaum. 2016. SIMD-based soft error detection. In *Proceedings of International Conference on Computing Frontiers.*

[10] Z. Chen, S. W. Son, W. Hendrix, A. Agrawal, W. K. Liao, and A. Choudhary. 2014. NUMARCK: Machine Learning Algorithm for Resiliency and Checkpointing. In *Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis.*

[11] Chen-Yong Cher, Meeta S. Gupta, Pradip Bose, and K. Paul Muller. 2014. Understanding Soft Error Resiliency of Blue Gene/Q Compute Chip through Hardware Proton Irradiation and Software Fault Injection. In *Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis.*

[12] Majid Dadashi, Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. 2014. Hardware-Software Integrated Diagnosis for Intermittent Hardware Faults. In *Proceedings of International Conference on Dependable Systems and Networks.*

[13] Teresa Davies and Zizhong Chen. 2013. Correcting Soft Errors Online in LU Factorization. In *International Symposium on High-Performance Parallel and Distributed Computing.*

[14] Sheng Di and Franck Cappello. 2016. Adaptive Impact-Driven Detection of Silent Data Corruption for HPC Applications. *IEEE Transactions on Parallel and Distributed Systems* 27, 10 (Oct. 2016), 2809–2823.

[15] Jack Dongarra, Pete Beckman, Terry Moore, and Patrick etc. Aerts. 2011. The International Exascale Software Project Roadmap. *High Perform. Comput. Appl.* (2011).

[16] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. 2012. Algorithm-based Fault Tolerance for Dense Matrix Factorizations. In *Proceedings of International Symposium on Principles and Practice of Parallel Programming.*

[17] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. 2012. Combining partial redundancy and checkpointing for HPC. In *Proceedings of International Conference on Distributed Computing Systems*.

[18] Christian Engelmann, Hong H. Ong, and Stephen L. Scott. 2009. The Case for Modular Redundancy in Large-Scale High Performance Computing Systems. In *the IASTED International Conference*.

[19] Kurt Ferreira, Jon Stearley, James H. Laros, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G. Bridges, and Dorian Arnold. 2011. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis*.

[20] M Gomaa, C Scarbrough, T N Vijaykumar, and I Pomeranz. 2003. Transient-fault recovery for chip multiprocessors. In *Proceedings of International Symposium on Computer Architecture*.

[21] L. A. B. Gomez and F. Cappello. 2015. Detecting and Correcting Data Corruption in Stencil Applications through Multivariate Interpolation. In *Proceedings of International Conference on Cluster Computing*.

[22] Nathanael Hübbe, Al Wegener, Julian Martin Kunkel, Yi Ling, and Thomas Ludwig. 2013. Evaluating Lossy Compression on Climate Data.. In *Proceedings of International Supercomputing Conference*.

[23] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2016. HAFT - hardware-assisted fault tolerance.. In *Proceedings of European Conference on Computer Systems*.

[24] Dong Li, Jeffrey S. Vetter, and Weikuan Yu. 2012. Classifying soft error vulnerabilities in extreme-Scale scientific applications using a binary instrumentation tool. In *Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis*.

[25] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V Adve, Vikram S Adve, and Yuanyuan Zhou. 2008. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*.

[26] Xiang Ni, Esteban Meneses, Nikhil Jain, and Laxmikant V Kalé. 2013. ACR : Automatic Checkpoint / Restart for Soft and Hard Error Protection. In *Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis*.

[27] Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. 2002. EDDI: Error Detection by Diverse Data and Duplicated Instructions. *IEEE Trans. Comput.* 51, 2 (2002).

[28] Daniel Oliveira, Laércio Pilla, Nathan DeBardeleben, Sean Blanchard, Heather Quinn, Israel Koren, Philippe Navaux, and Paolo Rech. 2017. Experimental and analytical study of Xeon Phi reliability. In *Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis*.

[29] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. 2005. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of International Symposium on Code Generation and Optimization*.

[30] S. Di and F. Cappello. 2016. Fast Error-Bounded Lossy HPC Data Compression with SZ. In *Proceedings of International Symposium on Parallel and Distributed Processing*. 730–739.

[31] Keun Soo Yim. 2014. Characterization of Impact of Transient Faults and Detection of Data Corruption Errors in Large-Scale N-Body Programs Using Graphics Processing Units. In *Proceedings of International Symposium on Parallel and Distributed Processing*.

[32] Keun Soo Yim, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2011. Hauberk: Lightweight Silent Data Corruption Error Detector for GPGPU. In *Proceedings of International Symposium on Parallel and Distributed Processing*.