Automated Algorithmic Error Resilience for Structured Grid Problems Based on Outlier Detection

Amoghavarsha Suresh and John Sartori
University of Minnesota
{sure0043,jsartori}@umn.edu

ABSTRACT

In this paper, we propose automated algorithmic error resilience based on outlier detection. Our approach exploits the characteristic behavior of a class of applications to create metric functions that normally produce metric values according to a designed distribution or behavior and produce outlier values (i.e., values that do not conform to the designed distribution or behavior) when computations are affected by errors. For a robust algorithm that employs such an approach, error detection becomes equivalent to outlier detection. As such, we can make use of well-established, statistically rigorous techniques for outlier detection to effectively and efficiently detect errors, and subsequently correct them. Our error-resilient algorithms incur significantly lower overhead than traditional hardware and software error resilience techniques. Also, compared to previous approaches to application-based error resilience, our approaches parameterize the robustification process, making it easy to automatically transform large classes of applications into robust applications with the use of parser-based tools and minimal programmer effort. We demonstrate the use of automated error resilience based on outlier detection for structured grid problems, leveraging the flexibility of algorithmic error resilience to achieve improved application robustness and lower overhead compared to previous error resilience approaches. We demonstrate $2\times - 3\times$ improvement in output quality compared to the original algorithm with only 22% overhead, on average, for non-iterative structured grid problems. Average overhead is as low as 4.5% for error-resilient iterative structured grid algorithms that tolerate error rates up to 10E-3 and achieve the same output quality as their error-free counterparts.

Categories and Subject Descriptors

D [Software]: Miscellaneous

Keywords:

algorithmic error resilience, application robustification, outlier detection, structured grids

1. INTRODUCTION

As technology scaling continues, variability increases with every process generation, leading to significant reliability challenges for current and future computing systems. These reliability challenges are compounded by the ever-increasing device counts in modern processors and processor counts in high-performance computing systems. Ensuring reliability is already a challenge in current-generation computer systems [11, 5] and will pose an even greater challenge in future-generation systems [10, 2]. While many hardware and software-based error resilience schemes have been proposed in the past [15], their heavy reliance on redundancy, worst case design, and conservative correctness guarantees becomes increasingly impractical, given the extreme energy and performance constraints targeted by current- and future-generation systems [10, 2]. Compared to more conventional approaches to error resilience, algorithmic error resilience, wherein an algorithm is replaced by a robust version of the same algorithm, offers the potential for greater flexibility, reduced overheads, and application- and algorithm-aware approaches to error resilience [17, 16, 8, 18].

Previous work on algorithmic error resilience has demonstrated opportunities and potential benefits. Application robustification [16] proposes to transform applications into numerical optimization problems that can be solved with stochastic optimization techniques. Since these optimization techniques converge to the correct result even when computations are noisy, robustified applications are naturally error tolerant. As static and dynamic non-determinism become more common and more prominent in current and future technologies, application robustification may be a useful means of achieving acceptable results on hardware that is necessarily stochastic by nature [8, 16]. Algorithmic error resilience has also been demonstrated in the context of specific application classes. Huang and Abraham [8] propose algorithm-based fault tolerance for dense linear algebra, while Sloan et al. [17] propose algorithmic techniques that reduce the overhead of fault detection for sparse linear algebra, based on the well-structured (e.g. diagonal, banded diagonal, etc.) nature of many sparse problems.

In [18], algorithm-based error localization and recomputation has been proposed as an alternative to a checkpoint and rollback scheme. The localization and recomputation approach is shown to be effective for iterative linear solvers for parallel experiments involving multi-node processors. The work in [18] also shows the viability of scaling algorithmic error resilience techniques for parallel systems.

While the application-based error resilience approaches proposed in previous work show promise, they also have several limitations. First, the proposed techniques are only applicable for select applications and must be applied on an application-by-application basis, since the transformation process is different for every application. Also, it is unclear how to perform robustification for a given application using the previously-proposed techniques, and furthermore, it is uncertain which applications can be robustified by the techniques. The techniques also must be applied manually by a programmer who is an expert in application-based error resilience, making the implementation and adoption of robust applications a difficult process.

In contrast to previously-proposed approaches for application robustification, the automated algorithmic error resilience techniques we propose are generally applicable to large classes of applications and are designed in a parameterized fashion such that any application that fits into a covered application class can be robustified by our error-resilient algorithms. In addition to being more general, our error resilience approaches are easier to apply to programs, due to their parameterized nature, and can be applied automatically with minimal programmer effort and expertise. We have developed a parser-based tool that enhances the error resilience of applications automatically.[1] To the best of our knowledge, this is the first work to provide an algorithmic error resilience framework that is both general and automated. In this paper, we make the following contributions.

- We introduce novel approaches for algorithmic error resilience based on outlier detection.
- We demonstrate how application robustification can be automated for large classes of applications and provide a parser-based tool that performs application robustification.
- We apply outlier detection-based algorithmic error resilience to an important class of applications – structured grids – and demonstrate equivalent error resilience at significantly lower overhead compared to conventional software and hardware error resilience techniques (e.g., TMR [15]). To the best of our knowledge, no previous works have demonstrated error-resilient algorithms for this class of applications.
- We show $2\times - 3\times$ improvement in output quality compared to the original algorithm with only 22% overhead, on average, for non-iterative structured grid problems. Average overhead is as low as 4.5% for error-resilient iterative structured grid algorithms that tolerate error rates up to $10E-3$ and achieve the same output quality as their error-free counterparts. Performance overhead is lower for lower fault rates.

2. CHARACTERIZING STRUCTURED GRID ALGORITHMS

Structured grid problems represent an important class of algorithmic methods that are prevalent in scientific and engineering problems [1]. They are characterized by a distinct pattern of communication and computation, in which data are represented as a multi-dimensional grid with regular relationships between grid points, and computation consists of making updates to the grid. During updates, each data point in the grid is influenced by a neighborhood of surrounding data points. Structured grid applications can be classified into two categories – iterative and non-iterative – based on the nature of updates to the grid. Below, we describe this dichotomy of structured grid problems.

[1] Our automated algorithmic error resilience tool is available for download at the following link: http://www.ece.umn.edu/users/jsartori/tools.html

2.1 Non-iterative Structured Grid Applications

In non-iterative structured grid applications, each data point in the grid is updated only once. Grid updates are performed using a kernel or characteristic function that describes how each point is influenced by its neighbors in the grid. A common form of computation in non-iterative structured grid applications is convolution, in which a kernel is applied at each grid point to overlap the neighborhood of interest around the grid point. Kernel coefficients weight the influence of each neighboring point on the update to the current grid point, and a scalar product is computed between a kernel function and the neighborhood around a point to obtain the output for a location in the grid. For example, the grid in a non-iterative structured grid problem might represent image or video data, and the kernel might represent some transformation or classifier applied to the grid, such as sharpening, blurring, or feature detection.

We identify and exploit two important characteristics of non-iterative structured grid problems in our outlier-based error resilience approaches.

Significant data reuse: Since each grid point is influenced by its neighbors, the neighborhood influencing a grid point overlaps with the neighborhoods influencing neighboring grid points. For example, consider a $m \times n$ rectangular kernel. The neighborhoods of two adjacent points share either $(m-1) \times n$ or $m \times (n-1)$ common grid locations. Thus, many common data points are used when updating neighboring grid points.

Constant kernel / characteristic function: The coefficients of the kernel or characteristic function are constant and are used in the computation at every grid point. Often, kernel values are also distributed symmetrically or according to a deterministic pattern. We will explain in Section 3.1.1 how we exploit these two characteristics of non-iterative structured grid algorithms to implement error-resilient structured grid algorithms.

2.2 Iterative Structured Grid Applications

Iterative structured grid applications are those that make repeated updates to the grid over a number of iterations. The data points in the grid can represent physical quantities, function coefficients, or values over a surface or volume. Computations frequently take the form of a numerical optimization that iterates over the data points, updating the solution of a system of differential equations until convergence is reached. Further classification of iterative structured grid applications is possible, based on whether the problem is time-dependent or time-independent.

Time-independent applications describe physical phenomena that have no dependence on time. The phenomena modeled by these problems are represented by systems of equations. For example, the Laplacian equation can be used to describe a physical relationship over a spatial region, such as the relationship between electric potential and charge density in a conductor. In this case, the partial differential equations (PDEs) describing a phenomenon are classified as elliptic PDEs [7]. Obtaining a solution to the PDEs provides a measurement of the physical quantity in the region of interest.

To solve a representative system of PDEs for a particular application, the structured grid that represents the region of interest is first initialized with a probable solution. The initial solution is iteratively updated, according to the finite difference method (FDM) to advance the region of interest toward convergence. The grid is said to converge if the L2 norm error, which describes the difference between successive iterations, is zero or negligible. Equation 1 shows the calculation of L2 norm error between two vectors x and y. The same relation can be applied to points in a grid to quantify the difference between grid values in successive iterations. As the grid approaches convergence, the L2 norm continues to decrease monotonically. As we will show in Section 3.1.2, we can create a metric function that characterizes the expected behavior of L2 norm for iterative structured grid problems for the purpose of outlier detection. Since the L2 norm is already computed by the original (non-error-resilient) algorithm, the overhead required to implement outlier detection-based error resilience in this manner can be kept low.

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + ... + (x_n - y_n)} \tag{1}$$

Time-dependent PDEs are classified as either hyperbolic or parabolic and like time-independent PDEs, are solved by initializing the grid with a probable solution and applying FDM. Unlike time-independent problems however, an appropriate number of grid updates for these problems can be decided without measuring a metric that tracks convergence (e.g., L2 norm error). For problems in which the boundaries of the grid do not have active time-dependent sources (e.g., an active voltage source in a wave propagation problem), the grid updates are only influenced by the initial state of the grid, and the result is expected to converge to some steady state. For such time-dependent problems, the L2 norm error is also expected to decrease monotonically and as such can be used to design a metric function with predictable behavior for the purpose of outlier detection. For problems involving active sources, monotonic behavior of metric functions like L2 norm cannot be assumed, and a different approach is needed to allow outlier detection. Error-resilient algorithms for time-dependent structured grid problems with active sources are a subject of ongoing work. Hyperbolic PDEs exhibit metric function behavior that oscillates at a well-defined and predictable frequency, but does not necessarily converge to any particular steady state value. For such problems, alternative metric functions could be defined in terms of the well-defined frequency domain components of other metric functions, such as L2 norm. Since the frequency components follow an expected distribution, any observed spectral analysis that does not fit the expected distribution can be classified as an outlier. We expect this technique to work much the same as the metric-based approaches described above, with the addition of a frequency domain transform (e.g., FFT).

3. OUTLIER DETECTION IN STRUCTURED GRIDS

An outlier is an observation (or subset of observations) that deviates markedly from the rest of the data in its sample set. Outliers naturally arise in sample sets due to changes in system behavior, fraudulent behavior, errors, or simply through natural but unusual deviations in populations. Outlier detection can identify system faults, fraud, etc. before they escalate with potentially catastrophic consequences [6]. A familiar example of outlier detection is the 3σ rule, based

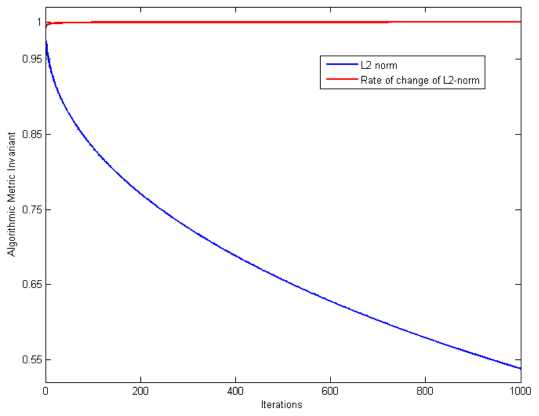


Figure 1: For solving of Poisson equations, the monotonically decreasing behavior of the L2 norm error metric function as the grid is updated can be exploited to perform outlier detection.

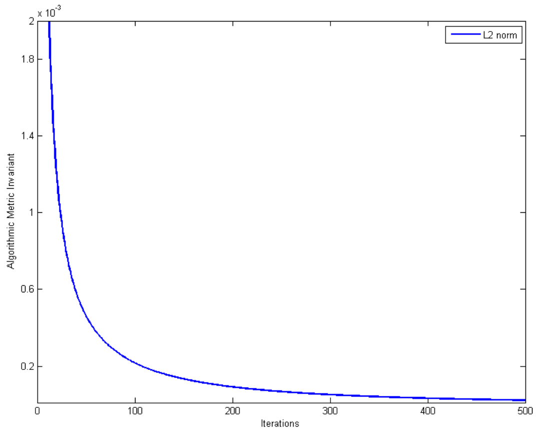


Figure 2: For solving of Laplacian equations, the L2 norm error metric function decreases monotonically.

on the principle that 99% of a normal distribution's data fall within three standard deviations from the mean [14]. To detect outliers accurately and efficiently, a designer should select an outlier detection scheme that is suitable for the sampled data set in terms of the correct distribution model, the correct attribute types, scalability and speed of the approach, any incremental capabilities to allow exemplars to be updated dynamically, and model accuracy. Outlier detection is frequently performed by using statistical, neural, and machine learning techniques [6].

Our error resilience approach derives from characterizing applications in different classes to identify algorithmic invariants that can be used to validate their computations. The algorithmic invariants are used to design metric functions that produce metric values according to a designed distribution or behavioral pattern. The metric functions are also designed to produce outliers when the underlying algorithm is affected by errors. As an example, consider structured grid applications where partial differential equations (PDEs) are solved using techniques such as FDM. In applications governed by Laplacian equations, the metric L2 norm is expected to monotonically decrease. This invariant behavior is violated when an application is affected by errors. Thus, algorithmic invariant metrics like L2 norm error can be used to perform outlier detection. In Figures 1- 5 we

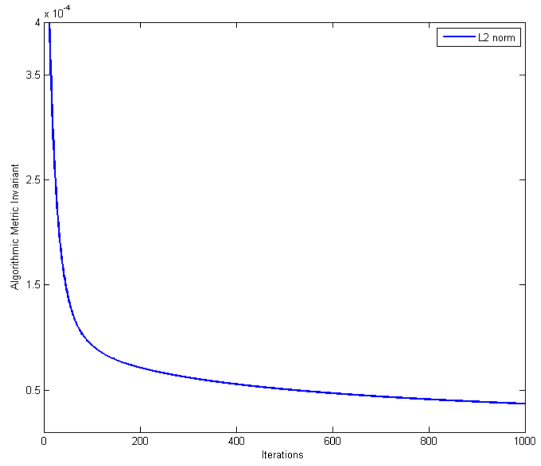


Figure 3: For heat dissipation equations, the L2 norm error metric function decreases monotonically.

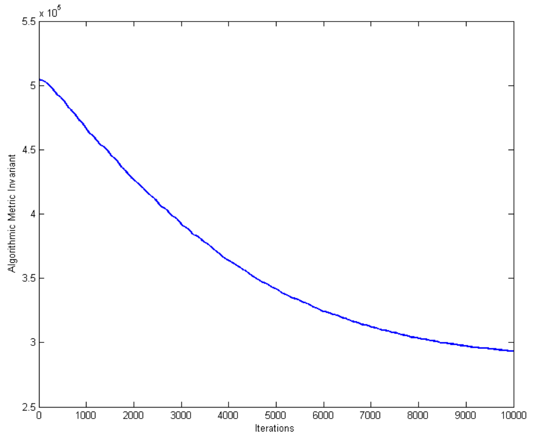


Figure 4: For wave propagation equations with no time-dependent source at the boundary, the L2 norm error metric function oscillates within a small envelope, but the overriding behavior is monotonically decreasing.

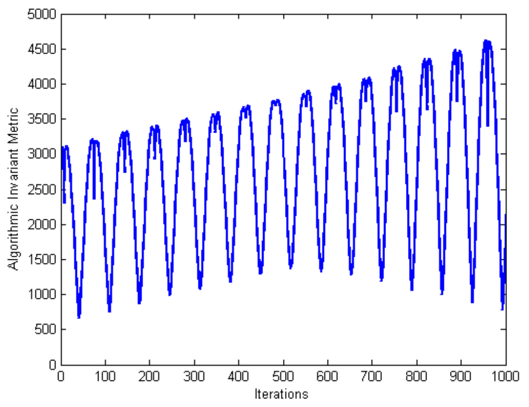


Figure 5: For wave propagation equations with a time-dependent source at the boundary, the L2 norm error metric oscillates at a deterministic frequency. The frequency distribution of the L2 norm can be used to create a metric function for outlier detection.

```
for($i = 0,\ \ i < 2m$)
  $quotient\ =\ (int)\ kernel[0][i]\ /\ kernel[0][i+1]$
  $max_{remainder} = 0$
  for($j = 0, j < (2n+1)$)
    $remainder = (((int)kernel[j][i]/kernel[j][i+1])$
                $-quotient)$
    if($remainder\ >\ max_{remainder}$)
      $max_{remainder}\ =\ remainder$
  $C_{max}[i]\ =\ quotient\ +\ max_{remainder}$
  $C_{min}[i]\ =\ quotient\ -\ max_{remainder}$
```

Figure 6: For kernel-based non-iterative structured grid problems, the values computed for one column in the grid can be bounded by the values computed for another.

illustrate the behavior of an L2 norm error metric for several different types of structured grid applications governed by PDEs. Results are shown for a grid size of 192 x 192. Depending on the underlying modeling characteristics of the application, the expected behavior of the metric is different. Below, we describe examples of metrics and how they are used to perform outlier detection.

The L2 norm error for applications based on elliptic PDEs, such as Poisson (Figure 1) and Laplacian (Figure 2) equations, decreases monotonically as the grid is updated. The L2 norm error for parabolic PDEs, such as those used in heat dissipation problems (Figure 3), also decreases monotonically as the grid is updated. Deviation from this characteristic behavior is used to detect outliers (details in Section 3.1.2).

The L2 norm error for hyperbolic PDEs, such as those used in wave propagation problems, does not behave monotonically. However, it does oscillate (from monotonically increasing to monotonically decreasing) at a deterministic frequency. As shown in Figure 4, the overriding behavior is monotonically decreasing, with oscillations confined to a small envelope. In such cases, we can refine the tolerance of our outlier detection threshold to account for the oscillation envelope. Figure 5, demonstrates a different case of wave propagation problems where there may be no overriding behavior. However, we can potentially use the frequency distribution of L2 norm error as a metric function to detect outliers in such cases, since deviation from the characteristic frequency indicates an error.

For a well-designed metric function, error detection is akin to outlier detection, and statistically rigorous techniques for outlier detection [6] can be used to detect the occurrence of errors in an algorithm. Outlier detection can also be employed to aid in localization and correction of errors that have been detected (Section 3.1).

3.1 Error-Resilient Structured Grid Algorithms

3.1.1 Non-iterative Structured Grid Applications

Non-iterative applications can be represented as some form of convolution or stencil operation over the grid.

$$Convolution[i][j] = \sum_{k=0}^{2n}\sum_{l=0}^{2m} Kernel[k][l]*Grid[i-n+k][j-m+l] \tag{2}$$

As described in Section 2.1, we observe characteristics of non-iterative structured grid applications that can be exploited to develop outlier-based error detection, and subsequently, error correction schemes. Given the large amount of data reuse between computations for neighboring grid points (both in the kernel and the grid), the range of possible values for a neighboring computation can be constrained in terms of the computed value for the current grid point. These constraints can be used to perform outlier detection. I.e., an error may cause a neighboring computation to be classified as an outlier, according to the expected range of values it might assume. We have developed an error resilience approach wherein a column (row) of a kernel is written as a linear combination of another column (row). From this formulation, we can place a range on possible values computed in a column (row) based on the value computed in another column (row) and identify outliers based on their location in (or outside of) the expected value distribution. We have automated the process of formulating a column in terms of another column, as described in Figure 6.

$$krnl = 1/159 \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} \qquad (3)$$

$$krnl[][3]^T = \begin{bmatrix} 4 & 9 & 12 & 9 & 4 \end{bmatrix}$$

$$krnl[][2]^T = \begin{bmatrix} 5 & 12 & 15 & 12 & 5 \end{bmatrix}$$

$$krnl[][2] = 1 \cdot krnl[][3] + \begin{bmatrix} (0.25 * krnl[0][3]) \\ (0.33 * krnl[1][3]) \\ (0.25 * krnl[2][3]) \\ (0.33 * krnl[3][3]) \\ (0.25 * krnl[4][3]) \end{bmatrix}$$

$$C_{max}[2] = 1 + 0.33$$

$$C_{min}[2] = 1 - 0.33$$

$$min(abs(grid[][k] \cdot krnl[][2])) = abs(0.66 * grid[][k] \cdot krnl[][3])$$

$$max(abs(grid[][k] \cdot krnl[][2])) = abs(1.33 * grid[][k] \cdot krnl[][3])$$

Equation 3 provides an example for Canny edge detection [3] that demonstrates how to derive an expected range of values for the dot product involving column k of the grid and column i of the kernel, based on the computation for a neighboring column. In a horizontal sweep of the grid, the dot product involving column k of the grid and column $i+1$ of the kernel is computed before the dot product involving columns k and i. Thus, we represent column i as a linear combination of column $i+1$. If the maximum coefficient in the linear combination function relating columns i and $i+1$ is $C_{max}[i]$, we can bound the range of values produced by i by $\pm C_{min}[i]$ to $\pm C_{max}[i]$ times the result for $i+1$. The bounds used in outlier detection are tighter when all coefficients in the same column (or row) of a kernel are of the same order of magnitude. Absolute value is used in Equation 3 to accommodate both positive and negative numbers. The range can be cut in half if all grid or kernel values are positive (or negative), as may be the case in several structured grid problems. For example, the values in a grid representing a grayscale image must fall within the range [0,255].

3.1.2 Iterative Structured Grid Applications

To create error-resilient algorithms for iterative structured grid applications, we rely on metric functions that are calculated during grid updates. We design metric functions with expected behaviors or output distributions, based on application invariants. For example, in solving Poisson or Laplacian equations we can define a metric function based on the L2 norm (that is already computed by the applications) that is expected to always increase or decrease at a certain rate α as the grid is updated. The metric is compared between successive grid updates to check whether the rate of change of the metric conforms to the expected distribution. A relaxation factor τ is used to distinguish the threshold between acceptable metric values and outliers. Observed metric values outside the range of $\alpha \pm \tau$ are classified as outliers. The relaxation factor can be determined based on some established rule (e.g., 3σ) or empirically. Observation of an outlier indicates that the update at one or more grid points was faulty. When an outlier is observed, the site of the fault should be localized to allow recovery.

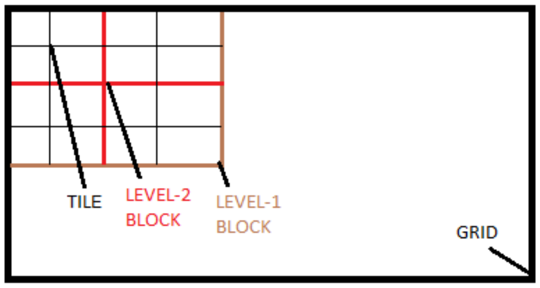


Figure 7: The grid is decomposed into multiple levels to reduce the overheads for error detection, localization, and recovery.

The relaxation factor also negotiates a tradeoff between computation accuracy and overhead computations for error checking. A larger value of τ results in more false negative errors (since more errors are deemed too insignificant to be detected as outliers) and thus, less accuracy but also less overhead, since some small errors are ignored. For many of the applications we study, small errors can be tolerated (e.g., perhaps resulting in more iterations to convergence), while larger errors cannot (e.g., preventing convergence).

In order to reduce the overhead of error detection, localization, and recovery, we decompose the grid into three levels – level-1 (L1) blocks (coarsest), level-2 (L2) blocks, and tiles (finest), as shown in Figure 7. The metric function for the grid at a given iteration i is calculated across all grid points, and the metric value of a tile is the sum of metric values at each point within the tile. Likewise, the metric value for an L2 (L1) block is the sum of metric values for each tile (L2 block) within the L2 (L1) block. The metric value for the entire grid is the sum of metric values of the L1 blocks. Our automated algorithmic error resilience tool performs grid decomposition automatically based on grid dimensions. The user only needs to use a preprocessor directive to identify the variables that define the grid size (see Section 4.2 and Appendix A).

To reduce the overhead of outlier detection, we do not perform checks for every location in the grid. Instead, initial comparisons for outlier detection take place at the coarsest granularity of the grid (L1). Metric values are compared to the metric values from the previous iteration to detect outliers, localize them at the current level, and trace them to the finest level of the grid (tile) for correction. Grid decomposition also reduces the overhead of localization and recovery, since only a subset of tiles need to be checked and recovered when an outlier is detected. After localizing an outlier to a tile, the tile is re-computed and re-checked up to two additional times in case an error occurs during recovery. If an outlier is still detected after two rollbacks, the relaxation factor is increased and the metric is compared against the value in iteration $i-2$ rather than the value in $i-1$. This allows forward progress in case a slightly erroneous value

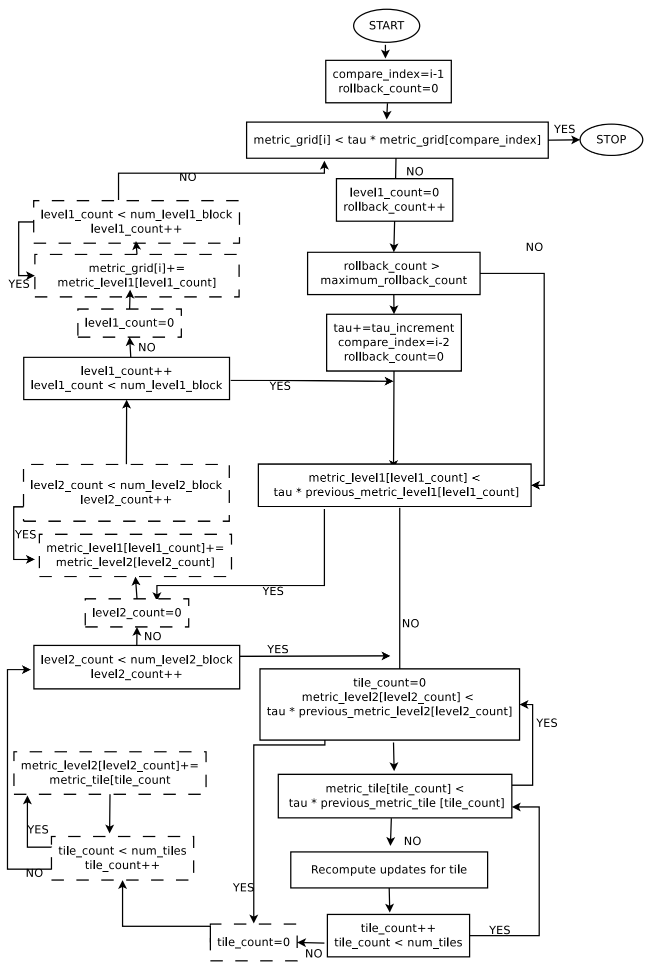

START
compare_index=i-1
rollback_count=0
metric_grid[i] < tau * metric_grid[compare_index]
YES
STOP
NO
level1_count=0
rollback_count++
rollback_count > maximum_rollback_count
tau+=tau_increment
compare_index=i-2
rollback_count=0
level1_count < num_level1_block
level1_count++
metric_grid[i]+= metric_level1[level1_count]
level1_count=0
level1_count++
level1_count < num_level1_block
metric_level1[level1_count] < tau * previous_metric_level1[level1_count]
level2_count < num_level2_block
level2_count++
metric_level1[level1_count]+= metric_level2[level2_count]
level2_count=0
level2_count < num_level2_block
level2_count++
tile_count=0
metric_level2[level2_count] < tau * previous_metric_level2[level2_count]
metric_level2[level2_count]+= metric_tile[tile_count
metric_tile[tile_count] < tau * previous_metric_tile [tile_count]
tile_count < num_tiles
tile_count++
Recompute updates for tile
tile_count++
tile_count < num_tiles
tile_count=0

Figure 8: Flowchart describing error detection, localization, and recovery for iterative structured grid applications. Steps denoted with dotted lines can be eliminated in certain cases when the fault model is known.

was accepted in $i-1$, resulting in false positive detections in iteration i. The algorithm used for fault detection, localization, and recovery is shown in Figure 8. For simplicity of exposition, Figure 8 assumes a metric value that is expected to decrease after each grid update. Our automatic error resilience tool instantiates and inserts a function that implements these error resilience functionalities into the code of a structured grid application. Instantiation and placement are based on key variable names identified by preprocessor directives (see Section 4.2 and Appendix A). This is akin to a library-based approach for algorithmic error resilience.

The procedure described in Figure 8 is generalized to provide error resilience against many different types of faults (see Section 5); however, the procedure can be optimized to reduce overhead in certain scenarios when the fault model is known. Certain fault distributions, such as bimodal faults, can result in two faults of nearly equal and opposite magnitudes masking in a coarse-level metric (e.g., L1 block). Additional fine-grain checks are necessary in the general case to ensure that such faults are detected. However, if knowledge of the fault model precludes the possibility of such fault masking, the steps represented by dotted blocks in Figure 8 can be forgone, significantly reducing the overhead of error resilience (see Section 6).

4. AUTOMATIC PROGRAM TRANSFORMATION

4.1 Motivation

Previous approaches for application robustification [16] consist of reformulating applications as stochastic optimization problems. An application must first be expressed as a constrained optimization problem, manually transformed to an unconstrained exact penalty form, and then solved using stochastic gradient descent and conjugate gradient algorithms. Transforming an application into a stochastic optimization problem is non-trivial, and must be performed for each application individually. The transformation involves the following considerations.

- The application must be reformulated into a stochastic optimization problem. Only certain types of applications are amenable to being transformed into stochastic optimization problems, and even the process of identifying such applications is performed manually on a case-by-case basis.
- Manual robustification requires intimate knowledge of algorithms such as stochastic gradient descent or conjugate gradient, as well as the application being transformed.
- Each application is transformed manually and individually, and the approach for transforming an application is different for each application.

Manual robustification of applications requires significant programmer effort. For application robustification to be easily adoptable, it should involve minimal programmer effort and expertise. To ease the burden on the programmer, we provide automatic program transformation tools that can perform robust transformations for large classes of applications. Our techniques require relatively unobtrusive changes to the application, rather than a paradigm shift in the coding of the application. We have parameterized our error resilience algorithms and metric functions so that required application information can be easily extracted by a parser through preprocessor directives. Other than marking a few key variables in the original code, the entire process of application robustification is automated. The only reason variables must be marked by the programmer is that different programmers use different variable names. E.g., one programmer may call the grid "grid" while another calls it "matrix". Alternatively, a library-based approach could be used to standardize the naming of key variables and eliminate the need for any variable marking. This infrastructure eases the programmer's effort in adopting error resilient algorithms. The implementation of automatic program transformation for structured grids is explained in the following section.

4.2 Automated Algorithmic Error Resilience for Structured Grid Applications

Our automated algorithmic error resilience tool requires the following information to be marked with preprocessor directives in structured grid applications.

- *Size* and *Layout* – These directives identify the variables that define the size of the grid in each dimension and the dimensionality of the grid data structure. This information is required to determine how to decompose the grid to improve the efficiency of error localization.

- *Current* and *Previous* arrays – These directives identify the variable names for the arrays that are used as input and output during grid updates. Grid data are used during outlier detection to calculate the change in the grid between grid updates.

- *End of grid update code.* A template function is inserted into the code, immediately following the grid update code. The function is responsible for grid decomposition and error detection and correction (see Figure 8 for details). The template function is instantiated with the values of other variables marked by preprocessor directives (i.e., size, layout, current, previous), and this directive is used to determine where to insert the function call.

Appendix A provides a code example showing how to mark key variables in a program in preparation for automatic robustification by our tool.

5. METHODOLOGY

5.1 Fault Model

Our evaluation focuses on transient faults that affect the outputs of numerical computations. Other manifestations of transient faults, such as memory corruption, deviations of control flow, or memory access errors are assumed to be accounted for using simple low-overhead techniques, unless they manifest as numerical data errors that the proposed techniques cover. This is a widely used fault model [9, 12, 19, 17]. Note that our error resilience techniques can detect and correct silent data corruptions that result in outliers.

The methodology for injecting errors has been adopted from previous work [17]. When a fault occurs, it is modeled by drawing a value from one of the fault distributions below and adding it to the target operation. These distributions are selected to model the arithmetic effects of circuit-level faults at a high level, making it possible to parametrize them to represent multiple low-level fault models.

Symmetric Faults: The following distributions model faults that affect the outputs of circuits and that have equal probability of being positive or negative.
1: Bimodal: Distribution with two Gaussian modes centered at $\pm 1E5$, each with variance $1E2$.
2: Bimodal: Distribution with two Gaussian modes centered at $\pm 1E10$, each with variance $1E5$.
3: Unimodal: Gaussian distribution with mean 0 and variance 100.

Memory Faults:
4: Bitflip: An exponential distribution represents a single bit flip in the binary representation of a number.

Non-Symmetric Faults:
5: Unsymmetric: Gaussian distribution centered at $1E5$ and with variance 100: represents a one sided error distribution (e.g. unsigned representation).
6: Trimodal: Mixture of models 1 and 2, each sampled half the time: models timing errors in functional circuit units, which are biased toward most and least significant digits.

5.2 Fault Injection

Fault injection is performed at a selected rate using Pin tool for binary instrumentation. Binary instrumentation provides an opportunity to instrument a fault at the most native level of instruction execution. To inject faults, floating point instructions are perturbed at the selected rate by reading the register contents of the processor and changing one of the operand register values according to the selected fault model.

5.3 Benchmarks

5.3.1 Non-iterative Applications

Non-iterative structured grid applications in our test set include Gaussian image blurring [13] and Canny edge detection. Three types of images were used.
Image-1: Synthetic image with Gaussian-distributed data centered at 0 with variance 1. Data points are scaled from [0,1] to the range [0,255].
Image-2: Synthetic image with Gaussian-distributed data centered at 0 with variance 1.
Real world example images: We use a collection of sample images of size 32x32, 64x64, 128x128, and 256x256.

Output quality is represented in terms of the average deviation between the computed (X_i) and pristine (X'_i) pixel values: $(1/N)\sum_{i=1}^{N} |X_i - X'_i|/X_i$. The overhead in terms of extra operations performed for both iterative and non-iterative applications is defined as
$O_{FLOPs} = FLOPs_{error\_injected\_run}/FLOPs_{pristine\_run}$, where $FLOPs$ is the number of floating point operations.

5.3.2 Iterative Applications

Iterative applications in our test set include the following time-independent (†) and time-dependent (‡) structured grid problems.
Poisson equation†, solved using Jacobi method
Laplacian equation†, solved using explicit finite difference method
Heat dissipation‡, solved using explicit finite difference method

Iterative structured grid applications may use grids with sizes on the order of $10E6$ or greater and are often implemented for distributed systems, where the grid is distributed between multiple processing nodes. Since we tested our approach on a single processor system, we used grid sizes that can fit in a single processor's memory, equivalent to a chunk that would be distributed to a node in a distributed system. We show the error resilience afforded by our techniques by quantifying, for the fault models in Section 5.1, the error rate that can be tolerated by our error-resilient structured grid algorithms. All the applications have been run for 1000 iterations and the value of the output quality metric at the end of 1000 iterations is compared between the pristine, error injection, and error injection + error resilience runs. Since errors may increase the number of iterations required to reach convergence, we also count the number of iterations (N_{iter_differ}) required for a non-pristine run to achieve the same output quality (accuracy) as in the pristine run. Iteration overhead is defined as $O_{iter} = O_{FLOPs} * (N_{iter\_pristine}/(N_{iter\_pristine} - N_{iter\_differ}))$.

6. RESULTS

6.1 Non-iterative applications

Tables 1 and 2 show the output quality, error resilience, and performance of our error resilient algorithms for image blurring and edge detection. We show detailed evaluations for bitflip and unimodal faults, since error magnitude may be comparable to signal magnitude, making outlier detection more challenging. Compared to the pristine run, average overhead (O_{FLOPs}) is 22.75% and average deviation

Table 1: Average deviation for non-iterative applications for different grid dimensions ($GD \times GD$), test images (Img), and fault models (Unimodal=3, Bitflip=4).

GD	Img	Canny (3)	Canny (4)	Gauss (3)	Gauss (4)
64	1	0.001990	0.001692	0.001701	0.001794
64	2	0.001962	0.001962	0.001829	0.001648
128	1	0.002215	0.001688	0.002083	0.001638
128	2	0.002139	0.002139	0.002118	0.002109

Table 2: Comparison of output quality (average deviation (AD)) and overhead (O_{FLOPs}) with respect to pristine run for Canny edge detection with and without error resilience for different grid sizes; Unimodal fault model, fault rate=1E-2.

Metric	Error Resilience?	64x64	128x128	256x256
AD	No	0.00468	0.00461	0.00451
	Yes	0.00199	0.00221	0.00206
O_{FLOPs}	No	1.245	1.350	1.345
	Yes	1.187	1.191	1.303

is 0.0019064. Compared to the error injected run, output quality is improved by 2× on average. For bimodal, trimodal, and unsymmetric faults, error magnitudes are of the order $\pm 10E5$ or $\pm 10E6$ and produce large outliers that are easily detected and corrected by our techniques.

6.2 Iterative Applications

The parameters of the grid decomposition orchestrate a tradeoff between accuracy and overhead for outlier detection, localization, and recovery. Table 3 compares overhead and error resilience for several grid decompositions. We find that a smaller L2 block size results in lower overhead, due to reduced error recovery overhead. Figure 9 compares overhead (O_{iter}) for different applications with various L2 block sizes. As described in Section 3.1.2, knowledge of the fault model can enable use of a lower-overhead error resilience approach in some cases. On average, overhead is reduced by 21% for the fault model-aware error resilience approach.

Figure 10 shows the fault rate that can be tolerated by our error-resilient heat dissipation algorithm for different fault models, as well as the overhead introduced in each case. Average overhead is higher than for other iterative applications because the L2 norm error is not used by the original algorithm but must be computed by the error-resilient algorithm. Even so, overhead of our error-resilient algorithm is significantly less than that of traditional hardware and software error resilience schemes, which employ spatial and/or temporal redundancy to achieve error resilience [15].

Table 3: We compare overhead (O_{iter}) for several different grid decompositions and fault rates. Results are shown for Poisson equation solver with bitflip fault model.

(Grid-size,#$L1$,#$L2$,L2-Size)	Fault Rate	O_{iter}
(320x320,1,25,64x64)	$1E-3$	1.83787
(384x384,1,36,64x64)	$1E-3$	2.00794
(384x384,1,16,96x96)	$1E-3$	1.53335
(480x480,1,25,96x96)	$1E-3$	1.96161
(576x576,3,9,64x64)	$1E-3$	1.48059
(576x576,3,9,64x64)	$1E-4$	1.24327
(576x576,2,9,96x96)	$1E-4$	1.23814
(768x768,4,9,64x64)	$1E-4$	1.27950
(768x768,3,9,96x96)	$1E-4$	1.27810

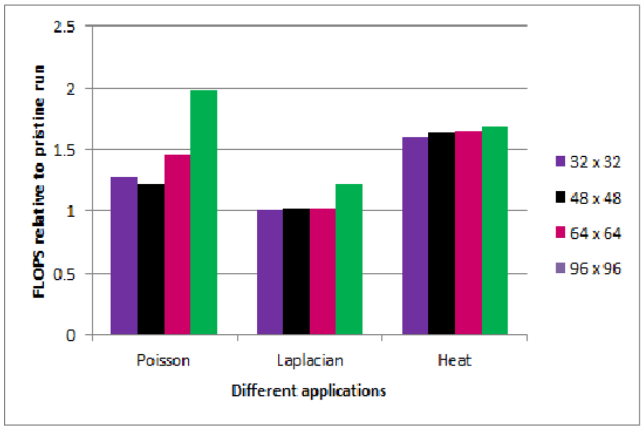


Figure 9: Overhead for different applications with different L2 block sizes, $\#L1 = 1$, $\#L2 = 1$ tile-dim=4, bitflip fault-model, fault-rate=1E-3.

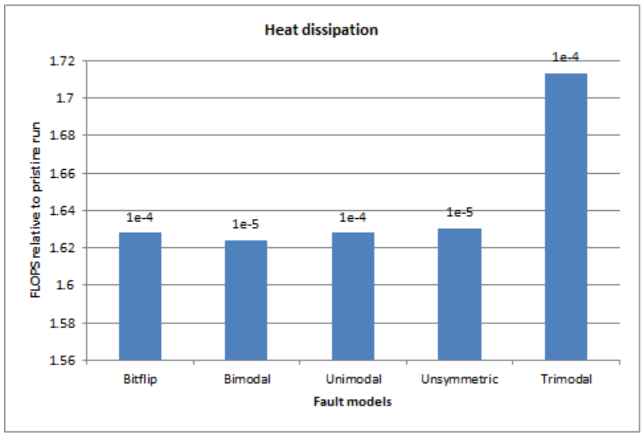
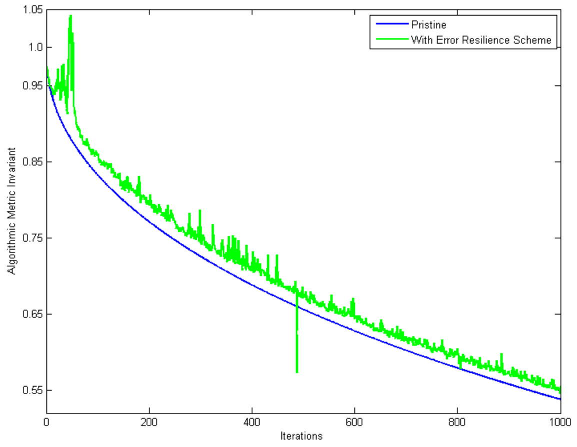


Figure 10: This figure shows the fault rate tolerated and overhead introduced by our error-resilient heat dissipation algorithm for different fault models on a 192×192 **grid.**

Figure 11 compares the L2 norm error metric for Poisson equation solving between the pristine run (top blue), the error-injected run with the original algorithm (bottom blue), and the error-injected run with our error-tolerant algorithm (top green). Our error-resilient algorithm detects outliers, tolerates errors, and achieves a similar convergence rate as in the pristine run. Without our error resilience techniques, however, the algorithm is unable to achieve convergence, and error magnitude blows up as the grid is updated. Figure 12 shows the fault rate that can be tolerated by our error-resilient Laplacian and Poisson solver for different fault models, as well as the overhead introduced in each case. Our error-resilient algorithm has significantly lower overhead than traditional hardware and software error resilience schemes [15].

7. CONCLUSION AND FUTURE WORK

Future high-performance and energy-efficient computing systems will likely be prone to errors and severely energy constrained. Error resilience may be required to ensure that



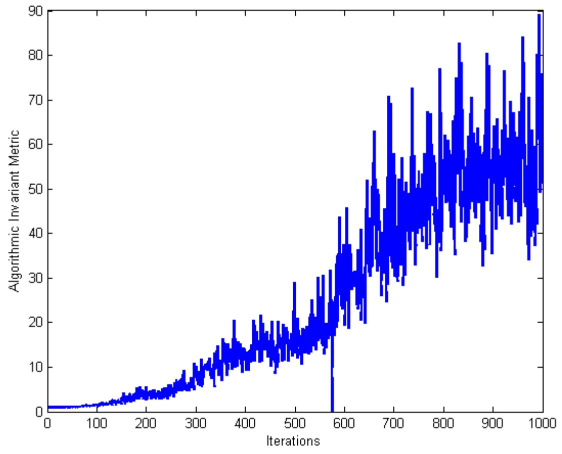


Figure 11: These figures show the behavior of the L2 norm error metric for Poisson equation solver on a 192×192 grid with bitflip errors injected at a rate of $1E-3$. Top: Our error-resilient algorithm detects outliers and achieves convergence at a similar rate as in the pristine run. Bottom: The original algorithm is unable to tolerate errors, and convergence is not achieved.

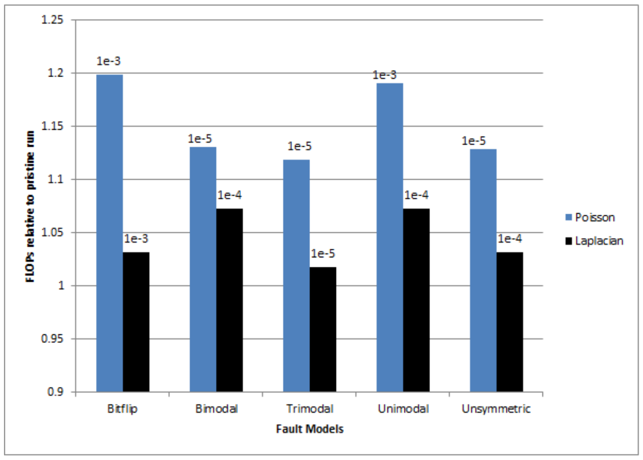


Figure 12: This figure shows the fault rate tolerated and overhead introduced by our error-resilient Laplacian and Poisson solvers for different fault models on a 192×192 grid.

applications can use these systems productively. In this paper, we propose automated algorithmic error resilience based on outlier detection. Our approach employs metric functions that exploit the characteristic behavior of algorithms – normally producing metric values according to a designed distribution or behavior and producing outlier values when computations are affected by errors, causing uncharacteristic behavior. Thus, for a robust algorithm that employs such an approach, error detection becomes equivalent to outlier detection. Compared to previous approaches to application-based error resilience, our approaches parameterize the robustification process, facilitating the automatic transformation of large classes of applications into robust applications with the use of parser-based tools and minimal programmer effort. We demonstrate automated algorithmic error resilience for structured grid problems. Our error-resilient algorithms have significantly lower overhead than traditional hardware and software error resilience techniques [15].

- Our error-resilient algorithms show $2\times-3\times$ improvement in output quality compared to the original algorithm with only 22% overhead, on average, for non-iterative structured grid problems.
- Average overhead (across fault models) is 4.5% to 15% for error-resilient iterative structured grid algorithms that tolerate error rates up to $10E-3$ and achieve the same output quality as their error-free counterparts. Performance overhead is lower for lower fault rates.

7.1 Learning from Outlier Detection-based Error Resilience

Through our study of outlier detection-based algorithmic error resilience, we have identified the following approaches as being particularly amenable to efficient algorithmic error resilience.

- *Using native features of an algorithm* can reduce the overhead of providing error resilience. For example, time-independent structured grid problems, such as solving Poisson and Laplacian equations, calculate the L2 norm between grid updates. An error resilience scheme that exploits this native feature may have less overhead than a more generic approach.
- Utilizing native features of algorithms also enables simple methods for extracting necessary information from applications, facilitating automated robustification. For example, in our robust structured grid algorithms, the programmer only needs to identify a few key features of the application (using preprocessor directives), rather than manually creating an entirely new application with a new algorithmic structure. This eases the adoption of error-resilient applications by minimizing programmer effort and required expertise.

While we have demonstrated the effectiveness of outlier detection-based algorithmic error resilience for structured grid applications, such approaches may not be applicable for all classes of applications. Future work should draw draw upon the design principles highlighted above to develop new automated algorithmic error resilience strategies that will provide robustness for a wide range of application classes.

7.2 General Guidelines for Invariant Selection

Outlier detection is a generic technique, which is applicable to different types of applications. This section provides guidelines that can aid in designing a robust algorithm based on outlier detection. The guidelines, which are based on our experience in applying outlier detection to different types of applications, can aid in identifying suitable invariant metrics that can be used for outlier detection. The guidelines are indicative of existing knowledge but are not exhaustive. Future work in this area should be able to utilize these guidelines to build upon existing work.

- Variables (either natural or contrived) that characterize the majority of data and/or computations in an application or algorithm are good candidates to be exploited for metric creation. Such variables are even more suitable for metric creation when they naturally conform to a certain behavioral pattern. For example, computing the L2 norm error metric used in applications that solve PDEs using FDM involves data for the entire grid and is affected by all grid update computations. Thus, the metric is sensitive to errors in grid data and computations. Expected monotonic behavior at a calculable rate makes the metric particularly useful for outlier detection.

- Data structures comprised of members that have bounded values are good candidates to be exploited for metric creation. Such structures are even more suitable for metric creation when their values are further constrained due to algorithmic dependencies. For examples, pixel values computed in many image processing applications are constrained to fall within a bounded range. Features such as data reuse (e.g., the constant kernel in convolution-based applications) can further constrain the range of possible values a pixel may take, improving the precision of outlier detection.

7.3 Extending Outlier Detection-based Error Resilience

The principles and guidelines for using outlier detection provided in Sections 7.1 and 7.2 are not exhaustive. We have demonstrated the application of outlier detection-based algorithmic error resilience for a class of application that primarily involves arithmetic operations. Additional classes of applications should be surveyed to determine the generality of outlier detection-based algorithmic error resilience and to provide a more comprehensive guide for leveraging outlier detection-based approaches for error resilience.

One potential direction in which outlier detection can be applied to new classes of applications is to use *dynamic invariant detection tools* [4], that provide invariants for a program based on dynamic analysis. An invariant is a property that holds at a certain point or points in a program; examples include constant variables ($x = a$), non-zero ($x \neq 0$), value range ($a \leq x \leq b$), linear relationships ($y = ax + b$), ordering ($x \leq y$), and sortedness (x is sorted) [4]. Dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions [4]. Hence, dynamic invariant detection tools can be used to determine algorithmic invariants for an application that could potentially be converted into metric functions and exploited for outlier detection-based error resilience.

Dynamic invariant detection tools must profile an application through multiple runs in order to infer invariants. This process can be time consuming. The techniques we propose for structured grids are based on static analysis of invariants and do not require profiling to determine invariants. Static invariant detection is also more reliable than dynamic invariant detection, since static analysis is based on program characteristics and not necessarily on dynamic execution behavior, which can be different for different input cases. Invariants determined through dynamic invariant detection will hold for the set of profiled inputs but may not be generally applicable. Thus, before relying on dynamic invariant detection, these requirements and concerns should be evaluated.

7.4 Automatic Program Transformation

While algorithmic error resilience may prove to be an efficient means of tolerating non-determinism in computing systems, one drawback of previous approaches for application "robustification" is that they have been applied manually to applications on a case-by-case basis. Consequently, previous approaches require significant expertise in application robustification as well as substantial manual effort. Also, it is non-trivial to determine *which* applications can be robustified using previous approaches and *how* to perform the robustification for a given application [16]. To facilitate the process of robustification for application developers and ease the adoption of robust algorithms, we have proposed automated techniques for application robustification that transform an application into a robust version of the same application with minimal programmer effort.

We have demonstrated how outlier detection is amenable to automatic program transformation by providing automatic program transformation tools for structured grid problems. Through this work, we wish to highlight the need for automatic program robustification. Along this vein, other future work directions include the following.

- *Providing robust libraries*, similar to the Standard Template Library of C++, would provide a convenient method to incorporate robust algorithms and data structures in applications. With a library-based approach, incorporating robust algorithms into applications could be as simple as specifying a compiler flag that directs the compiler to use the robust version of a library, rather than the standard version.

- Transforming existing application robustification techniques into a format that supports compiler-based automation. The adoption of existing application robustification approaches could be facilitated by creating tools, similar to our a tools that use parsers and preprocessor directives, to perform robust program transformations automatically.

8. REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[2] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[3] J. Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679–698, 1986.

[4] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[5] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz. Extending stability beyond cpu millennium: a micron-scale atomistic simulation of kelvin-helmholtz instability. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 58:1–58:11, 2007.

[6] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[7] K. A. Hoffmann. *Computational Fluid Dynamics for Engineers.*

[8] K. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computing*, 33(6):518–528, 1984.

[9] K.-H. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, C-33(6):518–528, 1984.

[10] ITRS. International technology roadmap for semiconductors, 2010.

[11] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender. Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer. *Device and Materials Reliability, IEEE Transactions on*, 5(3):329–335, 2005.

[12] N. Nakka, Z. Kalbarczyk, R. Iyer, and J. Xu. An architectural framework for providing reliability and security support. In *Dependable Systems and Networks, 2004 International Conference on*, pages 585–594, 2004.

[13] M. Nixon and A. S. Aguado. *Feature Extraction & Image Processing, Second Edition.* Academic Press, 2nd edition, 2008.

[14] F. Pukelsheim. The three sigma rule. *The American Statistician*, 48(2):88–91, 1994.

[15] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems (3rd ed.): design and evaluation.* A. K. Peters, Ltd., 1998.

[16] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 161–170, 2010.

[17] J. Sloan, R. Kumar, and G. Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, 2012.

[18] J. Sloan, R. Kumar, and G. Bronevetsky. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12, 2013.

[19] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer. Hauberk: Lightweight silent data corruption error detector for gpgpu. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 287–300, 2011.

APPENDIX

A. PROGRAMMER'S GUIDE

The following code example shows how to mark key variables with preprocessor directives in the code for a non-robust iterative structured grid application (Figure 13) and how the marked variables are used as inputs to a function that implements error resilience in the robust version of the code produced by our tool (Figure 14). Due to space limitations, several code details have been omitted in Figures 13 and 14. Figure 8 describes the code for grid decomposition and error detection and correction in greater detail, and Section 4.2 describes the preprocessor directives that mark key variable names in the code. For more details on our automated algorithmic error resilience tool and example usage, please download the tool and documentation (see footnote 1).

```
#define ROWS 1000000
#define COLS 1000000
#define STENCIL_LENGTH 1
#define PADDED_ROWS ROWS+2*STENCIL_LENGTH
#define PADDED_COLS COLS+2*STENCIL_LENGTH
#define DATA_LAYOUT 1

// Variable declarations
int m = ROWS;
int n = COLS;
int sl = STENCIL_LENGTH;
int dl = DATA_LAYOUT;
float heat_mat1[PADDED_ROWS*PADDED_COLS];
float heat_mat2[PADDED_ROWS*PADDED_COLS];

// Programmer-inserted preprocessor directives
#pragma struct_grid size m,n
#pragma struct_grid curr_array heat_mat1
#pragma struct_grid prev_array heat_mat2
#pragma struct_grid stencil_length sl
#pragma struct_grid data_layout dl
#pragma struct_grid iterator t
for (int t = 0; t < num_iters; t++)
{
   ...
   // Beginning of grid update code
   for(int row_idx = STENCIL_LENGTH;
   row_idx < (PADDED_ROWS - STENCIL_LENGTH);
   row_idx++){
      for(int col_idx = STENCIL_LENGTH;
      col_idx < (PADDED_COLS - STENCIL_LENGTH);
      col_idx++){
         int index = row_idx*PADDED_COLS + col_idx;
         #pragma struct_grid equation
 heat_mat1[index] =
         0.125*(heat_mat2[index+PADDED_COLS]
         - 2.0*heat_mat2[index]
         + heat_mat2[index-PADDED_COLS])
         + 0.125*(heat_mat2[index+1]
         - 2.0*heat_mat2[index]
         + heat_mat2[index-1])+heat_mat2[index];
      }
   }
   // End of grid update code
   #pragma struct_grid end_grid_update
}
```

Figure 13: The original code for a typical iterative structured grid application with preprocessor directives inserted to identify key variables.

```
#include "RobustSG.h"
...
for (int t = 0; t < num_iters; t++)
{
   ...
   // Beginning of grid update code
   for (int row_idx = STENCIL_LENGTH;
   row_idx < (PADDED_ROWS - STENCIL_LENGTH);
   row_idx++){
      for(int col_idx = STENCIL_LENGTH;
      col_idx < (PADDED_COLS-STENCIL_LENGTH);
      col_idx++){
         // ...grid update equation...
      }
   }
   // End of grid update code
   // Error detection and correction function
   grid_outlier_based_resilience(heat_mat1,
                                 heat_mat2,
                                 m, n, sl, dl, t);
}
```

Figure 14: Our automated tool transforms an application into a robust version that employs error-resilient algorithms.