View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Minnesota Digital Conservancy

Automated Application Robustification based on Outlier Detection

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Amoghavarsha Suresh

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

John Sartori

August, 2013

© Amoghavarsha Suresh 2013
ALL RIGHTS RESERVED

Acknowledgements

I would like to thank my adviser, Prof. John Sartori. Working with him has been a thoroughly enriching experience, he has patiently introduced me to various nuances of conducting research, especially to succinctly demonstrate the work we do. I am thankful to Prof. Pen Chung Yew and Prof. David Lilja, for their valuable feedback and taking time out of their busy schedule to serve on my committee. I sincerely thank Joseph Sloan for providing various tools and suggestions for my research work.

I have been fortunate to be brought up in a nurturing and endearing family and I am grateful for them. My parents– Suresh and Jwala– have tirelessly encouraged me to pursue my interests and have had amazing influence on my personality. I am grateful to my sister– Dhavala, who has always been a source of constant support and confidence to me. I am also greatly indebted to my uncle and aunt – Mahendra and Jayanathi and my cousins– Raveendra and Soujanya for their unconditional support in my endeavors. I have had the good fortune of studying and growing amongst a peer group who have thoroughly enriched my experiences and have positive influence on me, I would like to thank all of them.

Dedication

To those who held me up over the years!

Abstract

In this thesis, we propose automated algorithmic error resilience based on outlier detection. Our approach employs 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. Thus, for our robust algorithms, error detection becomes equivalent to outlier detection. Our error resilient algorithms use outlier detection not only to detect errors, but also to aid in reducing the amount of redundancy required to produce correct results when errors are detected. 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 two important classes of applications, namely, structured grid and dynamic programming problems, leveraging the flexibility of algorithmic error resilience to achieve improved application robustness and lower overhead compared to previous error resilience approaches.

Contents

List of Tables

List of Figures

Chapter 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. While many hardware and software-based error resilience schemes have been proposed in the past, 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. 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.

1.1 Related work

Previous work on algorithmic error resilience has demonstrated opportunities for algorithmic error tolerance. Application robustification [1] 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 [2, 1]. Algorithmic error resilience has also been demonstrated in the context of specific application classes. Huang and Abraham [2] propose algorithm-based fault tolerance for dense linear algebra, while Sloan et al. [3] 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 [4], 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 [4] 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 significant promise, they also have several limitations. First, the proposed techniques are only applicable for select applications and the transformations are applied uniquely on an application-by-application basis. 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 techniuqes. The techniuqes 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 in this thesis 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 provide a parser-based tool that enhances the error resilience of application automatically. To the best of our knowledge, this is the first work to provide an algorithmic error resilience framework that is both general and automated.

1.2 Outlier detection for error resilience

In this work, we propose algorithmic error resilience techniques based on outlier detection. An outlier is an observation (or subset of observations) in a data set that is inconsistent with the remainder of the data in the set [5]. Our error resilience approach derives from characterizing applications in different classes to identify algorithmic invariants that can be used to validate the 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. Hence, for a well-designed metric function, error detection is akin to outlier detection, and statistically rigorous techniques for outlier detection 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.

1.3 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. Even so, it is non-trivial to determine *which* applications can be robustified using previous approaches and *how* to perform the robustification for a given application. To facilitate the process of robustification for application developers and ease the adoption of robust algorithms, we propose automated techniques for algorithmic error resilience that transform an application into a robust version of the same application with minimal programmer effort.

In our automated approach to algorithmic error resilience, necessary application information is specified through the use of preprocessor directives that can be easily inserted by a programmer. We provide a parser that extracts the relevant information and performs a sort of source-to-source translation on the original application to produce a robust version of the application. We demonstrate automated robust program transformations for structured grid and dynamic programming problems. Note that

while previous approaches for application robustification were applied on a case-by-case basis to individual application. The algorithmic error resilience approaches we describe work for large, general classes of applications (e.g., Berkeley dwarfs [6]) that contain many individual applications.

1.4 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 two important classes of applications (structured grids and dynamic programming) and demonstrate equivalent error resilience at significantly lower overhead compared to conventional software and hardware error resilience techniques. To the best of our knowledge, no previous works have demonstrated error resilient algorithms for these classes of applications.
  - For structured grids, 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 $10E3$ and achieve the same output quality as their error-free counterparts.
  - For monadic dynamic programming problems, our error resilience techniques improve output quality by up to 40%, on average, compared to the original algorithm. Overhead is less than 59% on average – an improvement of at least $3.3\times$ compared to existing redundancy-based techniques that achieve the same output quality.

The rest of this thesis is organized as follows.

- In Chapter 2, we describe outlier detection as an algorithmic error resilience technique and provide guidelines for how and when outlier detection-based algorithmic error resilience can be used.

- In Chapter 3, we describe the structured grid class of applications and how to apply outlier detection-based algorithmic error resilience to robustify applications in this class.

- In Chapter 4, we describe the dynamic programming class of applications and how to apply outlier detection-based algorithmic error resilience to robustify a subset of dynamic programming applications.

- In Chapter 5, we describe the motivation and guidelines to follow for automatic program transformation. We also describe the specific application of automatic program transformation to structured grid and dynamic programming classes of applications.

- In Chapter 6, we describe the methodology, benchmarks, and metrics used to evaluate outlier detection-based algorithmic error resilience of structured grid and dynamic programming applications.

- In Chapter 7, we describe the results obtained for the benchmarks described in Chapter 6.

- In Chapter 8, we present conclusions from this research and suggest future research directions.

- In Appendix A, we provide a programmer's guide for using the automatic program transformation tool to provide error resilience for structured grid and dynamic programming applications.

Chapter 2

Outlier Detection

2.1 Outlier detection

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 [5]. A familiar example of outlier detection is the three sigma rule, based on the principle that 99% of a normal distribution's data fall within three standard deviations from the mean [7]. 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 [5].

Our error resilience approach derives from characterizing applications in different classes to identify algorithmic invariants that can be used to validate the 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 we can consider structured grid applications, where partial differential

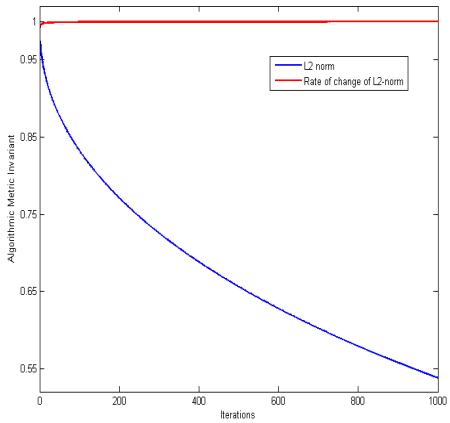

Figure 2.1: Change of an algorithmic invariant: Solving of Poisson equations, a structured grid problem. The monotonically decreasing behavior of the L2 norm error metric function as the grid is updated can be exploited to perform outlier detection.

equations(PDE) are solved using techniques such as, Finite difference method. Algorithmic invariant metrics like L2 norm error are used to perform outlier detection. In applications governed by Laplacian equations, the metric, L2 norm, normally is expected to monotonically decrease. This behaviour of monotonic decrease in L2 norm would be in violation when application is affected by errors, which can be used to detect errors. In Figures 2.2 - 2.5 we 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 2.1) and Laplacian (Figure 2.2) equations, decreases monotonically as the grid is updated. Deviation from this characteristic behavior is used to detect outliers (details in Section 3.2.2). The L2 norm error for parabolic PDEs, such as those used in heat dissipation problems (Figure 2.3), also decreases monotonically as the grid is updated.

- The L2 norm error for hyperbolic PDEs, such as those used in wave propagation

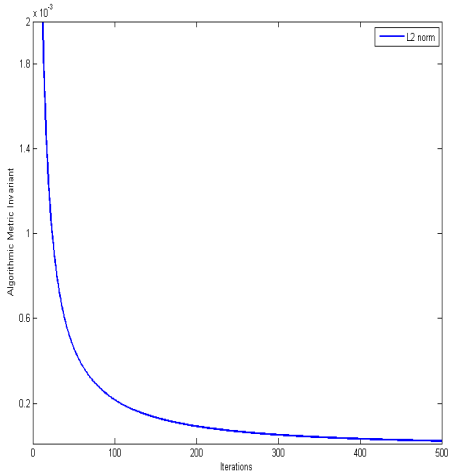

Figure 2.2: Change of an algorithmic invariant:Solving of Laplacian equations, a structured grid problem. The L2 norm error metric function decreases monotonically.

problems, does not behave monotonically. However, it does oscillate (from monotonically increasing to monotonically decreasing) at a deterministic frequency, as shown in Figure 2.4, the overriding behavior is monotonically decreasing, with oscillations conned to a small envelope. In such cases, we can refine the tolerance of our outlier detection threshold to account for the oscillation envelope. Figure 2.5, indicates 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. Alternatively, the metric function may account for the frequency at which the metric oscillates between monotonically increasing and monotonically decreasing (half) periods.

Hence, for a well-designed metric function, error detection is akin to outlier detection, and statistically rigorous techniques for outlier detection can be used to detect the occurrence of errors in an algorithm [5]. . Outlier detection can also be employed to aid in localization and correction of errors that have been detected.

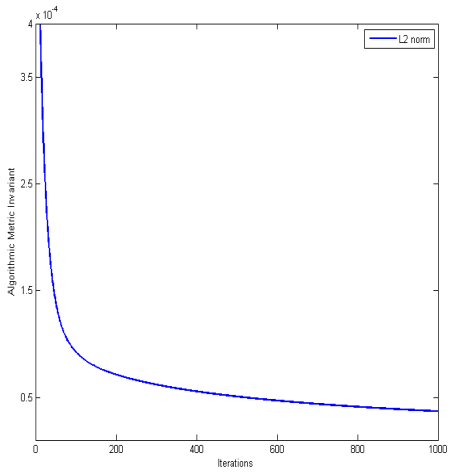

Figure 2.3: Change of an algorithmic invariant: Heat dissipation equations, a structured grid problem. The L2 norm error metric function decreases monotonically.

2.2 General Guidelines

Section 2.1 introduces the motivation for outlier detection and how to apply it in applications. Outlier detection is a generic technique, which is applicable to different types of application. This section provides the guidelines that can be used to design a robust algorithm using outlier detection. The guidelines provided are developed by applying outlier detection to different types of applications, which are discussed in detail in subsequent chapters. However the provided guidelines are just indicative of existing knowledge and is not an exhaustive guideline. Future work in this area should be able to build up on the existing work.

The following guidelines can be used for deciding how and when outlier detection can be applied:

- A metric to follow a particular direction at a given rate. e.g.: Application such as solving for PDE using FDM have characteristic behavior such that the L2 norm would vary, depending on the problem. dimensions. Hence the rate of change of metric (L2 norm) can be used to detect errors.

- A metric to have certain bounds. e.g.: Applications such as Image processing (filtering, compression) involves intelligent manipulation of pixels, which are bound

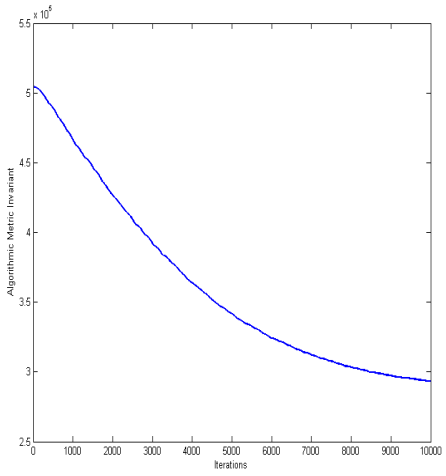

Figure 2.4: Change of an algorithmic invariant: Wave propogation equations, with no time-dependent source at the boundary, a structured grid problem. The L2 norm error metric function oscillates within a small envelope, but the overriding behavior is monotonically decreasing.

to take values in a given range.

- Applications with logical and arithmetic operations can use outlier detection, if logical operations are going to influence a local region in the application and not the entire solution. Eg: Monadic Dynamic Programming problems (explained in Ch 4) have logical operation influencing a local region in the solution. For example, the result of a choice (i,j) in Longest Common Subsequence problem would influence (i,j+1), (i+1,j) and (i+1,j+1) . Hence with the use of outlier detection and redundancy, if error at a point (i,j) occurs, the influence of that problem would be in the local region of the application and the damage (if any) can be undone with minimal effort.

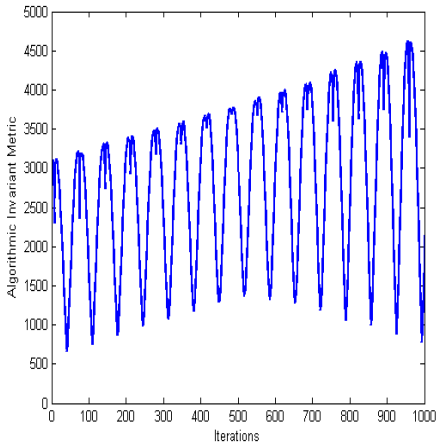

Figure 2.5: Change of an algorithmic invariant: Wave propogation equations, with time-dependent source at the boundary, a structured grid problem. The L2 norm error metric oscillates at a deterministic frequency.

Chapter 3

Outlier Detection in Structured Grids

3.1 Introduction

Structured grid problems represent an important class of algorithmic methods that are prevalent in scientific and engineering problems [6] . They are characterized by a distinct pattern of communication and computation, in which data are represented as a multidimensional 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 classiffied 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.

3.1.1 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 classiffier 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 x n rectangular kernel. The neighborhoods of two adjacent points share either (m - 1) x n or m x (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 3.2.1 how we exploit these two characteristics of non iterative structured grid applications to implement error resilient structured grid algorithms.

3.1.2 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 represents 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 classiffication 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 quantity 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 classiffied as elliptic PDEs [8]. 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 between successive iterations is zero or negligible. As the grid approaches convergence, the L2 norm continues to decrease monotonically. As we will show in Section 3.2.2, we can create a metric function based on 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.

Equation3.1 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.

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + ... + (x_n - y_n)} \tag{3.1}$$

Time-dependent PDEs are classiffied as either hyperbolic or parabolic PDEs 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). Nevertheless, 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, predictable behavior of metric functions like L2 norm cannot be assumed, and a different approach is needed to provide error resilience. Error-resilient algorithms for time-dependent structured grid

```
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
  max_bound[i]  =  quotient  +  max_remainder
  min_bound[i]  =  quotient  −  max_remainder
```

Figure 3.1: The values computed for one column can be bounded by the values computed for another.

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. We propose to define alternative metric functions for such problems in terms of the well-defined frequency domain components of other metric functions. I.e., since the frequency components follow an expected distribution, any observed spectral analysis that does not fit the expected distribution can be classiffied 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).

3.2 Error resilience algorithms

3.2.1 Non-iterative Structured Grid Applications

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

$$Convolution[i][j] = \sum_{k=0}^{2n} \sum_{l=0}^{2m} Kernel[k][l] * Grid[i-n+k][j-m+l] \qquad (3.2)$$

As described in Section 3.1.1, 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 classiffied 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. The process of formulating a column in terms of another column is described in Figure 3.2.1.

$$
\begin{aligned}
krnl &= 1/159 \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} \\
krnl[][3]^T &= \begin{bmatrix} 4 & 9 & 12 & 9 & 4 \end{bmatrix} \\
krnl[][2]^T &= \begin{bmatrix} 5 & 12 & 15 & 12 & 5 \end{bmatrix} \\
krnl[][2] &= krnl[][3] + \begin{bmatrix} (0.25 * krnl[0][3]) \\ (0.33 * krnl[1][3]) \\ (0.25 * krnl[2][3]) \\ (0.33 * krnl[3][3]) \\ (0.25 * krnl[4][3]) \end{bmatrix} \\
min(abs(grid[][k] \cdot krnl[][2])) &= abs(0.66 * grid[][k] \cdot krnl[][3]) \\
max(abs(grid[][k] \cdot krnl[][2])) &= abs(1.33 * grid[][k] \cdot krnl[][3])
\end{aligned}
\tag{3.3}
$$

Equation 3.3 provides an example for Canny edge detection [9] that demonstrates how to derive an expected range of values for the dot product involving column k of the grid and column j 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 $j + 1$ of the kernel is computed before the dot product involving column k and

column j. Thus, we represent column j as a linear combination of column $j+1$. If the maximum coefficient in the linear combination function relating columns j and $j+1$ is $Cmax$, we can bound the range of values for j by $\pm Cmax$ times the result for $j+1$. The bounds, which are 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].

3.2.2 Iterative 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 an expected behavior 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. Observation of an outlier indicates that the update at one or more grid points is faulty. When an outlier is observed, the site of the fault should be localized to allow recovery.

The relaxation factor also captures the tradeoff between computation accuracy and overhead computations for error checking. A larger value of τ results in more false negative errors (since more error 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,

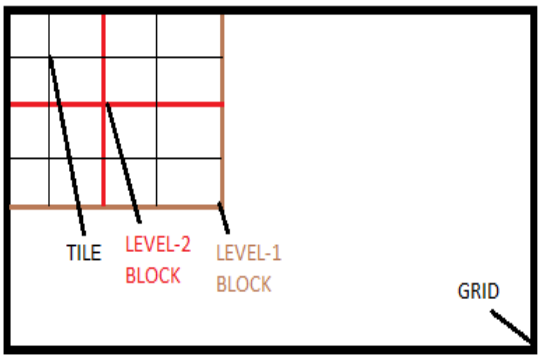

Figure 3.2: The grid is decomposed into multiple levels to reduce the overhead for error detection, localization, and recovery.

and tiles (finest), as shown in Figure 3.2. 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 L2 (L1) blocks 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.

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 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 3.3. For simplicity of exposition, Figure 3.3 assumes a metric value that is expected to decrease after each grid update.

The procedure described in Figure 3.3 is generalized to provide error resilience against many different types of faults (see chapter 6); 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 fault masking, the steps represented by dotted blocks in Figure 3.3 can be forgone, significantly reducing the overhead of error resilience (see chapter 7).

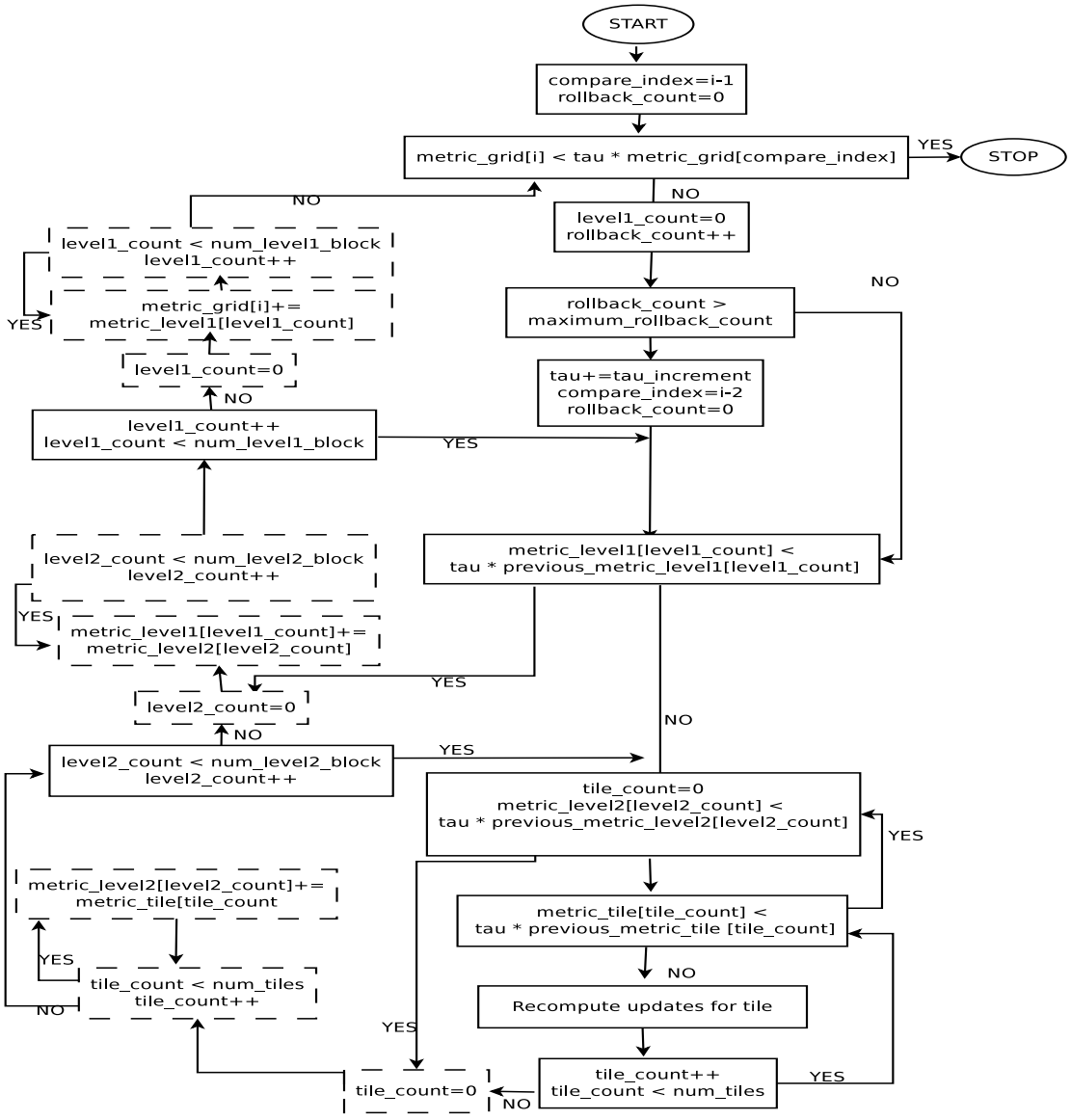

Figure 3.3: Flowchart describing error detection, localization, and recovery for iterative structured grid applications. Steps denoted with dotted lines can be eliminated in certain cases when the fault model is known.

Chapter 4

Outlier Detection in Dynamic Programming Problems

4.1 Introduction

Dynamic programming (DP) is a technique for solving a wide variety of discrete optimization problems such as scheduling, string editing, packaging, and inventory management [10]. Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the subproblems overlap, that is, when subproblems share sub-subproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub-subproblems [10]. A dynamic programming algorithm solves each sub-subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each sub-subproblem. Thus, "programming" in this context refers to a tabular method, not to writing computer code [10].

Typically dynamic programming is applied to optimization problems. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (typically minimum or maximum) value. When developing a dynamic programming algorithm, the following sequence of four steps is followed.

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from computed information.

Example: The longest common subsequence (LCS) problem is stated as follows. Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y. The LCS problem determines the longest of the common subsequences. A DP problem exhibits optimal substructure if optimal solutions to a problem incorporate optimal solutions to related subproblems. The optimal substructure in the LCS problem is described in Equation 4.1.

$$ci,j = \begin{cases} 0 & \text{if } i = 0 \, or \, j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \, and \, X_i = Y_j \\ max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \, and \, X_i \neq Y_j \end{cases} \tag{4.1}$$

In Equation 4.1, c[i,j] is the value of choice (i,j) and is represented using recursive terms – c[i-1,j], c[i,j-1], c[i-1,j-1]. Once the values of all the choices have been evaluated, if needed, an optimal solution is constructed from the table stored in array c[,].

Dynamic Programming problems are classified based on the nature of the problem at hand as either monadic or polyadic [11]. Problems containing a single recursive term are called *monadic* problems and those with multiple recursive terms are called *polyadic* problems. LCS is an example of a monadic problem, since for a choice (i,j) only one of the recursive terms is used. Floyd's all-pairs shortest path problem is an example of a polyadic problem, with recursive solution shown in Equation 4.2.

$$d^k_{i,j} = \begin{cases} c_{i,j} & \text{if } k = 0 \\ min(d^{k-1}_{i,j}, (d^{k-1}_{i,k}, d^{k-1}_{k,j})) & \text{if } \; 0 \geq k \geq (n-1) \end{cases} \tag{4.2}$$

In Equation 4.2, $d^k_{i,j}$ is the shortest path from node i to j using k nodes and $c_{i,j}$ is the weight of the edge between nodes i and j. Floyd's shortest path problem is an example of a polyadic problem because the solution $d^k_{i,j}$ requires both $d^{k-1}_{i,k}$ and $d^{k-1}_{k,j}$. However,

for monadic DP problems such as the LCS problem, the solution $c[i, j]$ requires only one of the $c[i-1, j], c[i, j-1], c[i-1, j-1]$ terms.

A DP problem can also be classified based on the number of preceding levels of recursion required to solve the problem. DP problems that depend on one preceding level are called *serial* problems and those that depend on more than one preceding level are called *non-serial* problems. Floyd's shortest path problem is an example of a serial DP problem, since a given solution in level k requires solutions from level $k-1$. The LCS problem is an example of a non-serial DP problem, since solving for level k requires solutions from levels k and $k-1$.

4.2 Impact of error on Dynamic Programming Problem

Dynamic programming problems are solved in two stages. In stage 1, all the choices are evaluated and the value or benefit of each choice (which represents usefulness to the current problem) is stored in a matrix or grid. In stage 2, the matrix is traversed to determine which of the choices makes up the solution of the problem. In both stages 1 and 2, dynamic programming problems use both logical and arithmetic operations to evaluate the values in the matrix. For robust evaluation of DP problems, both arithmetic and logical operations should be evaluated correctly in order to ensure that all dependent subproblems are evaluated robustly. Incorrect evaluation for a subproblem can result in propagation of faults to subsequent subproblems that depend on the current subproblem. Outlier detection-based algorithmic error resilience for DP problems introduces the following measures to achieve error resilience.

- ***Outlier detection intervals***: In stage 1, while evaluating the options, both logical and arithmetic operations are used and could be susceptible to errors. For a given DP problem, when a set of choices is considered, say *set A*, the set of choices that *set A* depends on, say *set B*, can be known a priori. Also, for a DP problem, the increase in value between a choice in *set A* and *set B* is bounded, and the bound can be determined statically. Since the values of choices in *set B* will be computed before *set A*, and the increase in value for each choice in *set A* is bounded, it is possible to estimate the range of possible values that the *set A* choices can take. Thus, we can perform outlier detection based on the expected

distribution of choice values to check for erroneous arithmetic results. We ensure the correctness of logical operations in subsequent steps.

- ***Redundancy during tracing***: In stage 2, the benefit of each choice has already been computed, and the arithmetic operations used to calculate the benefit have been checked with outlier detection. To ensure correctness of logical operations, redundancy is introduced for only those choices that are considered to be part of the solution. Since only a few choices and their dependent choices are likely to be part of the solution, only such choices are re-evaluated for logical correctness. Each problem uses certain tracing conditions to pick a choice (i,j) to be part of the solution. These tracing conditions are related to conditions used during the initial solving phase when a choice (i,j) is evaluated. Specifically, if a certain solving condition is true when evaluating choice (i,j), then a particular tracing condition should be true when the same choice (i,j) is considered during tracing. As such, the evaluation of a tracing condition k can be used to confirm correct evaluation of the solving condition l that should be true given that the tracing condition k is true. By re-evaluating only those choices that are likely to be part of the solution or influence the choices that are part of the solution, rather than using redundancy for all evaluations, we significantly reduce the overhead of ensuring correctness for logical operations. If a particular choice is found to be erroneous during tracing, then a subset of its neighbors might also be affected by the error, necessitating correction. Error correction is handled in the ***Rollback*** step, described next.

- ***Rollback***: If the logical operations at a particular choice (i,j) were erroneous, choices that depend on the erroneous choice should be re-evaluated to check whether they have been influenced by the error. If so, their values are recalculated. This procedure is repeated until the choices that depend on choice (i,j) are error-free. Once this is done, the tracing is rolled back and resumed at a choice that was not affected by the error at (i,j) or any other choices that depend on choice (i,j).

The three functionalities described above are used to provide error resilience for monadic DP problems. The robust algorithm provided in Section 4.3 is applicable only for monadic DP problems. A choice (i,j) in a monadic DP problem depends on a few

choices (known apriori) and also influences an equal number of choices. For example, in the LCS problem a choice (i,j) depends on only three other choices (i-1,j-1), (i-1,j), and (i,j-1) and can influence only three other choices (i+1,j+1), (i+1,j), (i,j+1). Since a choice (i,j) can only affect a few choices and the dependence relationships are known, implementing rollback is straighforward. In a polyadic problem, a choice (i,j) may be influenced by many choices (not known a priori) and may also influence many choices. Thus, if a logical error occurs at (i,j), then testing all other choices that could depend on (i,j) may require an exhaustive procedure, and a more conventional error resilience approach like TMR may potentially be more efficient.

4.3 Robust Dynamic Programming Algorithm Based on Outlier Detection

Implementation details for the functionalities described in Section 4.2 are described below.

- ***Outlier detection intervals***: Our outlier detection implementation for DP problems uses intervals to bound the expected values of choices and checks for outliers according to the expected value distribution. Outlier detection intervals are defined by unrolling the innermost loop of a program where the benefit of a choice is evaluated, such that the unrolled loop iterations constitute the outlier detection interval. A natural tradeoff exists between the amount of overhead required to perform outlier detection (higher for a shorter interval) and the overhead of recovering from an error if an outlier is detected (higher for a longer interval). We choose the interval length based on the problem size to minimize interval length while ensuring a minimum bound on the number of intervals for a given problem size, observing that a shorter interval usually results in lower overhead, since the overhead of recovery for a longer interval outweighs the overhead of performing outlier detection for a shorter interval.

  The maximum and minimum bounds for choices in a given interval are computed robustly using TMR and compared against the values of choices computed during the interval to check for outliers. In the event of an outlier detection, first, the

choices in the interval are recomputed. If the outlier persists after recomputation, computation is rolled back to the interval containing the first choice that influences the outlier choice. At the close of an interval, the values computed during the interval are assumed to be error-free and safe for use in the computations of future intervals.

Figure 4.1 provides implementation details for outlier detection intervals. In the flowchart, $dim1$ and $dim2$ represent the size of the grid in $dimension - 1$ and $dimension - 2$, respectively. The array max_bound stores the maximum values for the intervals. To calculate the maximum bound for an interval l ($max\_bound[l]$), $f1(l)$ and $f2(l)$ are application-dependent functions used to calculate the indices of choices that are considered in the maximum bound calculation. In each interval, $process(i, j)$ represents the function used to calculate the benefit of choice (i, j) and max_value holds the maximum value computed within the interval. At the end of the interval, max_value is compared against $max\_bound[l]$ and if it is greater than $max\_bound[l]$, an outlier is detected and the interval is recalculated. If the outlier persists after recalculation, it is likely that the error is in one of the choices that influences the current interval l, and benefit calculation is rolled back to a previous index in $dimension - 1$.

- ***Redundancy while tracing***:
  - Redundancy is used to check the correctness of logical operations. The tracing conditions are computed using TMR to ensure the logical correctness of solving conditions. If a tracing condition k is true and the corresponding solving condition l is false, or vice-versa, an error is detected and will be handled by the ***Rollback*** step.

Figure 4.2 describes our approach for providing robustness during tracing. The functions $s\_cond\_m(i, j)$ and $t\_cond\_k(i, j)$ represent the m^{th} and k^{th} conditions for solving and tracing. The tracing conditions are computed using TMR and only one of them will be true. Thus, if the corresponding solving condition is false, then the logical operation at (i, j) is determined to be incorrect and the ***Rollback*** routine will be called.

- ***Rollback***:
  - To check whether a logical error at location (i,j) affects any other dependent choices, each of the dependent choices is re-evaluated and, if necessary, corrected. Since solving conditions are known, the dependent choices of (i,j) can be determined.
  - Upon recomputing / correcting the choices affected by an error at choice (i,j), tracing is rolled back to a point that is guaranteed to be unaffected by the error at (i,j). While tracing, the subset of the choices that are chosen to be part of the solution is maintained as a list and a choice (m,n) that is unaffected by an error at (i,j) can be identified by tracing through the choices, starting at (i,j), to determine the first choice that does not propagate the value of choice (i,j). All choices that are chosen to be part of the optimal solution between choice (i,j) and choice (m,n) may be affected by the logical error at choice (i,j) and hence, they are discarded.

Figure 4.3 describes implementation details for rollback. The indices that need to be checked are stored in queue *check* and functions $f0(i,j)$, $f1(i,j)$, . . . $fn(i,j)$ calculate the indices that will be influenced by choice (i,j) if solving conditions $s\_cond\_0()$, $s\_cond\_1()$, . . . $s\_cond\_n()$ are true. Hence, we check whether each index is influenced by choice (i,j) and if so, the benefit for that choice is recalculated and the choice is inserted in the queue *check*. This process is repeated until the queue *check* is empty.

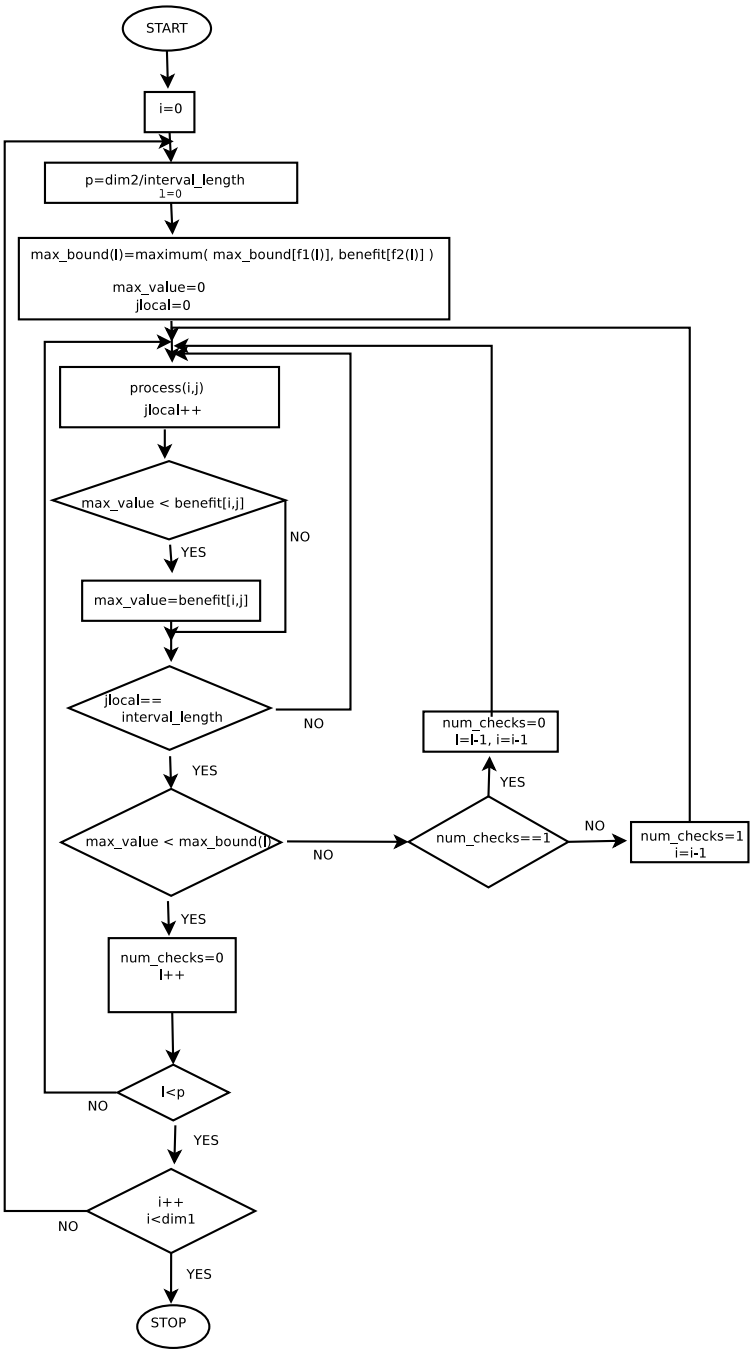

Figure 4.1: This flowchart describes the functionality implemented in the ***Outlier detection intervals*** stage of the robust algorithm. The functions f1(l) and f2(l) represent a generic form of choices that are involved in calculating the maximum bound for an interval l. The routine process(i,j) represents calculation of benefit for choice (i,j).

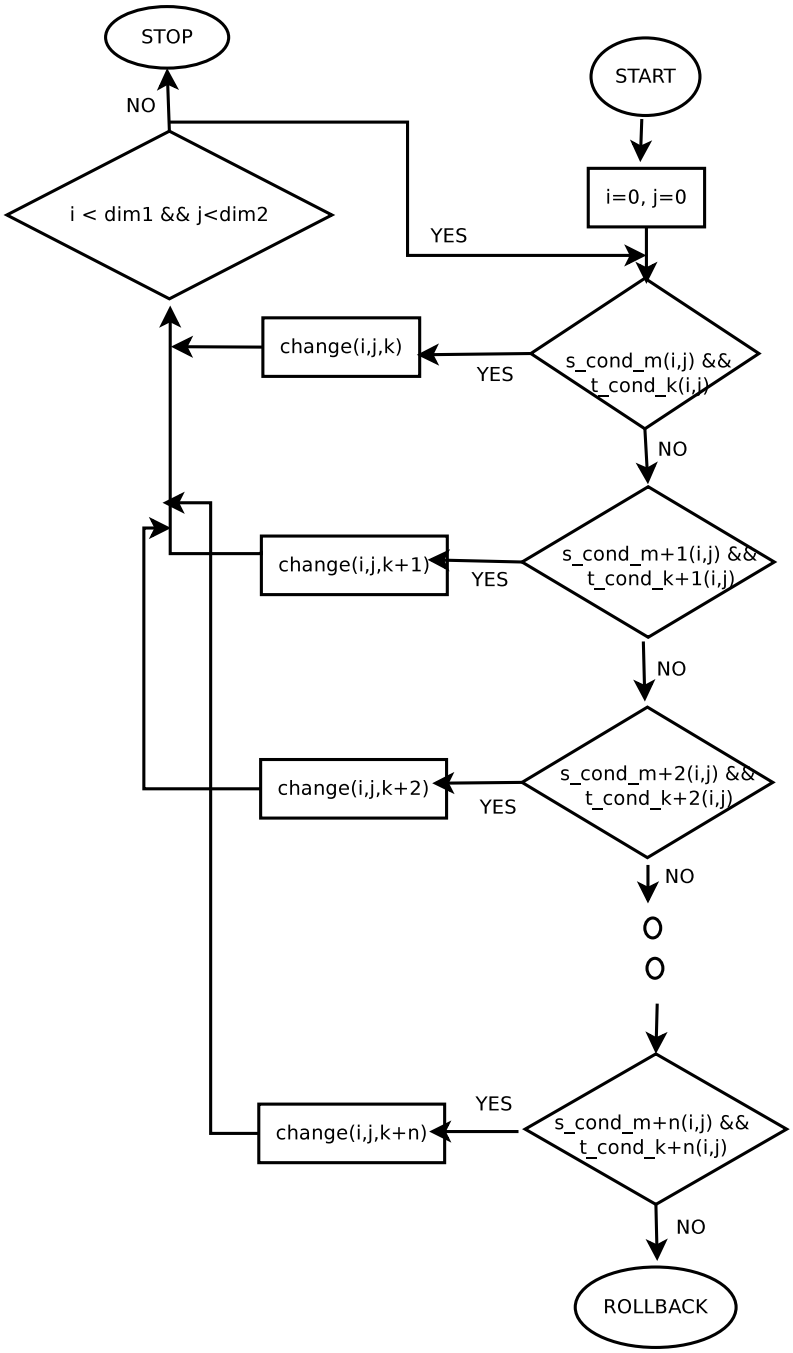

Figure 4.2: This flowchart describes the functionality implemented in the ***Redundancy while tracing*** stage of the robust algorithm. The terms s_cond and t_cond represent solving and tracing conditions, respectively. The flowchart represents the redundancy for a generic problem having n conditions. The routine $change(i, j)$ represents manipulation of indices i and j while tracing.

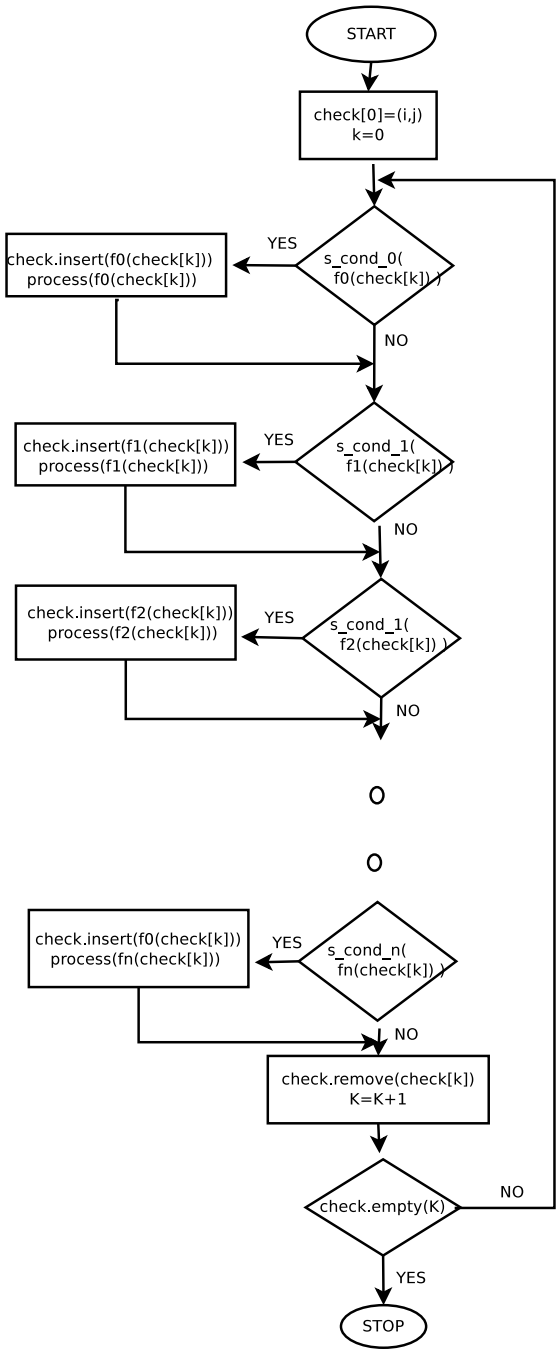

Figure 4.3: This flowchart describes the functionality implemented in the ***Rollback*** stage of the robust algorithm. The term s_cond represents solving conditions and the generic problem has n conditions to re-evaluate. The functions f0, f1, .. fn are used to calculate the indices involved in conditions s_cond_0, s_cond_1, ... s_cond_n. The functions insert, remove, and empty operate on the queue that maintains the list of choices to re-evaluate.

Chapter 5

Automatic Program Transformation

5.1 Motivation

Previous approaches for application robustification [1] 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 steps.

- 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 effort from the programmer. For application robustification to be easily adoptable, however, minimal programmer effort and expertise should be required. One approach to ease the burden on the programmer is to provide automatic program transformation tools that can perform robust transformations for large classes of applications. Another approach to ease the adoption of application robustification is to provide techniques that require relatively unobtrusive changes to the application, rather than a paradigm shift in the coding of the application.

In the spirit of approaches mentioned above, we have designed the error resilience technique, outlier detection, to be based on algorithmic invariants. The algorithmic invariants are used to construct metric functions that are used to monitor for outliers during the course of application execution. The algorithmic invariants are application-specific and are the necessary inputs for application robustification. The algorithmic invariants and the application information can be obtained as preprocessor directives to parsers to automatically implement error resilience for an application. This infrastructure eases the programmer's effort in adopting error resilience. The implementation of automatic program transformation for structured grids and dynamic programming is explained in the following sections.

5.2 Structured Grids

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

- Size – size of the grid in each dimension. This information is required to determine how to decompose the grid to aid in error localization.
- Current and Previous array – the array which is used as input and output in updating in every iteration. This information is needed to calculate the change in the grid in each dimension.
- Start and end of the region where loop construct is present.

This information is required to calculate a metric value (such as L2 norm) for the grid, the grid is decomposed as suggested in the proposed algorithm. A routine call is inserted

into the code, which uses the marked parameters to determine how to decompose the grid and perform error detection and correction.

Necessary preprocessor directives for structured grid application and their syntax are explained in Appendix Section A.1.

5.3 Dynamic Programming

To achieve the functionalities mentioned in Sections 4.2 and 4.3, the following work should be done.

- **Outlier detection intervals:** Mark the region in which the benefit matrix is being filled, i.e., the region in which solving conditions that determine the benefit of a choice are specified. Within this region, mark the conditions that will be evaluated to update the value of a choice (i,j). Marking these locations and conditions provides information necessary to unroll the solving loop, insert logic for calculating the maximum bound on benefit values in an interval, and recalculate benefit values in the interval if necessary.

- **Redundancy while tracing:** Specify the tracing conditions and their correspondence to the solving conditions. Tracing conditions are evaluated robustly (using TMR) and are also used to check the logical correctness of solving conditions.

- **Rollback:** Mark the code in which a choice is chosen to be part of the solution during tracing. This facilitates the maintainance of a list that is used to form the problem solution and that also aids in rolling back, if necessary. When an erroneous choice (i,j) is detected, rollback is performed by, (a) determining the choices that are affected by choice (i,j) and recalculating their benefits, and (b) determining a choice (m,n) that is unaffected by the error at choice (i,j) and discarding the choices chosen after (m,n) and resuming tracing from (m,n).

Necessary preprocessor directives for dynamic programming application and their syntax are explained in Appendix Section A.2.

Chapter 6

Methodology

6.1 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 [12, 13, 14, 3].

The methodology for injecting errors has been adopted from previous work [3]. 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.

- **Bimodal:** Distribution with two Gaussian modes centered at $\pm 1E5$, each with variance $1E2$.
- **Bimodal:** Distribution with two Gaussian modes centered at $\pm 1E10$, each with variance $1E5$.
- **Unimodal:** Gaussian distribution with mean 0 and variance 100.

Memory Faults:

- **Bitflip:** An exponential distribution represents a single bit flip in the binary representation of a number.

Non-Symmetric Faults:

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

6.1.1 Fault Injection

Fault injection has been performed using two different techniques.

Binary instrumentation: Fault injection based on binary instrumentation is performed by instrumenting faults, at the requested rate, in the binary of the application. This type of fault injection is used for applications with primarily floating point operations. Binary instrumentation provides an opportunity to instrument a fault at the most native level of instruction execution. The fault injection was done through the binary instrumentation tool Pin. According to the error rate requested, few of the floating point instructions are chosen, the register contents of the processor is read and one of the operand's register value will be changed according to the fault model sepcified in Section 6.1

Overloaded operators: Fault injection based on overloaded operators is performed at the application level. Data related to the application are declared to belong to a special class, and the operations (basic mathematical operators) involving the data are injected with faults, using the data distributions described above. This type of fault injection is done for applications that primarily involve integer operations. The main reason for injecting faults at the application level is to more easily restrict fault injection to certain critical operations in the code.

6.2 Benchmarks

6.2.1 Structured Grids

Non-iterative Applications

Non-iterative structured grid applications in our test set include Gaussian image blurring [15] and Canny edge detection. Three types of images were used.
Image-1: Synthetic image with Gaussian-distributed data centered at 0 with variance 1, but 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

$$O_{FLOPs} = FLOPs_{error\_injected\_run}/FLOPs_{pristine\_run} \tag{6.1}$$

where *FLOPs* is the number of floating point operations.

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 applications in this classification 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 6.1 , 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 Niter differ required for a non-pristine run to achieve the same output quality (accuracy) as in the pristine run. Iteration overhead

$$O_{iter} = O_{FLOPs} * (N_{iter\_pristine}/(N_{iter\_pristine} - N_{iter\_differ})) \tag{6.2}$$

6.2.2 Dynamic Programming

Dynamic Programming (DP) problems in our test set include the following problems.

- Longest Common Subsequence problem.
- 0/1 Knapsack problem.

Both problems belong are monadic DP problems. The problem sizes considered are represented as tuples (m,n), where the total number of options to be considered is m x n. Different problem sizes considered are (50,50), (500,500), (1000,1000) and (2500,2500). The overhead of using the robust scheme is expressed in Equation 6.3.

$$O_{int} = \frac{Integer\_operations\_error\_injected}{Integer\_operations\_pristine\_run} \tag{6.3}$$

The overloaded operators used for error injection are also added to the total instruction count during execution. Thus, for fairness in comparison, the pristine application is implemented using the overloaded class for error injection with similar error injection rate but without actually injecting any errors. As such, the overhead observed for a robust application is due to the use of algorithmic error resilience.

Qualtiy of results: For all benchmarks, we define quality metrics to compare the results obtained by the robust application to the results obtained by the pristine (no

errors injected) version of the application.

- Knapsack: Inject Result ($Inject\_result$): Final value of the result for the pristine application with and without error injection.
- LCS:
  - Inject Result (IR): Length of the common subsequence computed by the original application when errors are injected.
  - Inject LCS Result (ILR): Length of the common subsequence between the results of the original application with and without error injection.

Chapter 7

Results

7.1 Structured Grids

7.1.1 Noniterative applications

Tables 7.1 and 7.2 demonstrate 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 is 22.75% and average deviation 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 by our techniques.

Table 7.1: 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 7.2: 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

Table 7.3: 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

7.1.2 Iterative Applications

The parameters of the grid decomposition orchestrate a tradeoff between accuracy and overhead for outlier detection, localization, and recovery. Table 7.3 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 7.1 compares overhead (O_{iter}) for different applications with various L2 block sizes. As described in Section 3.2.2, 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 7.2 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. 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.

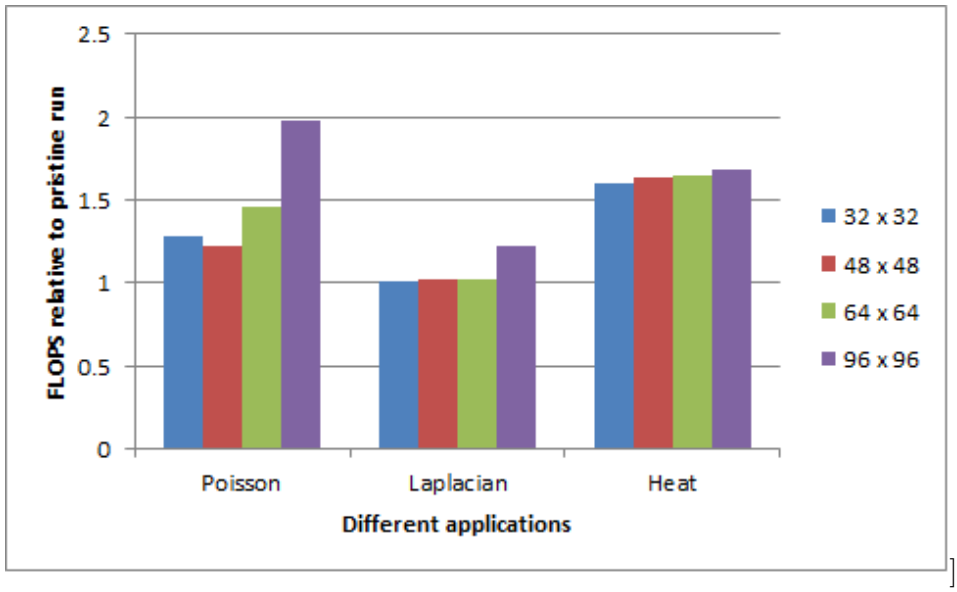

Figure 7.1: Overhead for different applications with different L2 block sizes, $\#L1 = 1$, $\#L2 = 1$ tile-dim=4, bitflip fault-model, fault-rate=1E-3.

Figure 7.3 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 7.4 shows the fault rate that can be tolerated by our error-resilient Laplacian solver for different fault models, as well as the overhead introduced in each case. Overhead of our error resilient algorithm is significantly less than that of traditional hardware and software error resilience schemes.

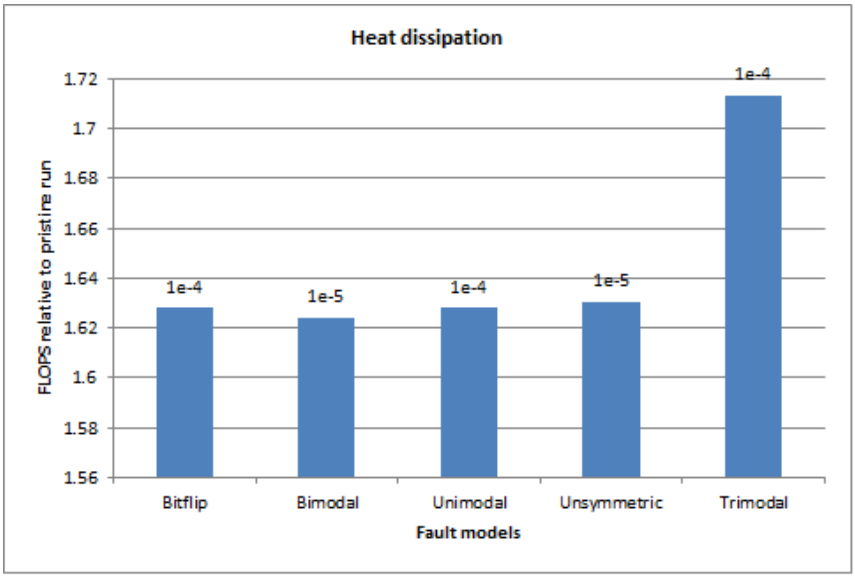

Figure 7.2: This figure shows the fault rate tolerated and overhead introduced by our error resilient heat dissipation algorithm for different fault models on a 192×192 grid.

7.2 Dynamic Programming

Tables 7.4 and 7.5 present the overhead and the quality metrics, respectively, for LCS and Knapsack problems. For the Knapsack problem, we present average results over 10 iterations for different problem sizes ((50,50), (500,500), (1000,1000), (2500,2500)). The average overhead across different fault models is 59%. Overhead is relatively high for the unsymmetric fault model compared to others. A reason for this is that the faults injected in the unsymmetric fault model are of the order $10E4$ to $10E5$, where as in bimodal and trimodal the magnitude is $\pm 10E4$ to $\pm 10E5$. Since the injected faults in other fault models can be either positive or negative, faults can cancel each other, resulting in fewer error detections and corrections. Consequently, the overhead for fault models such as bimodal and trimodal is relatively lower compared to the unsymmetric fault model. In the unimodal fault model, the faults injected are of the order $\pm 10E2$, which is of similar order to the numbers used in the application, and in some instances, the faults do not result in outlier detection or cause any adverse effects. The bitflip fault randomly flips one of the bits and hence the fault injected could be of any order. Hence

the faults injected are similar to that of Unsymmetric fault model and the overhead is similar to that of Unsymmetric fault model.

The quality metric $Pristine\_inject\_result$ – the value of the knapsack provided by the original application when errors are injected – is 15.6% inferior, on average, to the value of the knapsack provided by the pristine application unaffected by errors. The robust application provides an optimal value equal to that of the pristine application.

For the LCS problem, we present average results over 10 iterations for different problem sizes ((50,50), (500,500), (1000,1000), (2500,2500)). The average overhead across different fault models is 46% (excluding the case of (50,50)) and similar behavior as observed for the Knapsack problem is also observed for LCS under various fault models. The quality metrics indicate that, on average, the length of the common subsequence provided by the original application with error injection (IL) is 26.3% inferior (shorter subsequence) to the solution obtained by the pristine application. The length of common subsequence between the results provided by the original application with and without error injection (ILR) is 40.4% shorter than the longest common subsequence provided by the pristine application. The robust application produces the same results as the pristine application. As problem size increases, performance degrades for the original application with error injection, emphasizing the need for error resilience.

Table 7.4: These results compare overhead (O_{int}) for several different grid sizes and fault rates for the Knapsack problem.

Fault_model	Grid-size	Fault Rate	O_{int}	$Pristine\_result$	$Pristine\_inject\_result$
Unsymmetric	(50x50)	$1E-3$	1.44	1371	1090
	(500x500)	$1E-3$	1.80	15538	12320
	(1000x1000)	$1E-3$	1.719	31104	25742
	(2500x2500)	$1E-4$	1.765	77636	65253
Bimodal	(50x50)	$1E-3$	1.43	1371	1157
	(500x500)	$1E-3$	1.68	15538	13182
	(1000x1000)	$1E-3$	1.57	31104	25818
	(2500x2500)	$1E-4$	1.58	77636	66034
Trimodal	(50x50)	$1E-3$	1.43	1371	1307
	(500x500)	$1E-3$	1.60	15538	13627
	(1000x1000)	$1E-3$	1.57	31104	26055
	(2500x2500)	$1E-4$	1.57	77636	68336
Unimodal	(50x50)	$1E-3$	1.42	1371	1371
	(500x500)	$1E-3$	1.58	15538	15493
	(1000x1000)	$1E-3$	1.56	31104	30984
	(2500x2500)	$1E-4$	1.56	77636	77283
Bitflip	(50x50)	$1E-3$	1.23	1371	1123
	(50x50)	$1E-3$	1.71	15338	11854
	(50x50)	$1E-4$	1.57	31104	23814
	(50x50)	$1E-4$	1.59	77636	40042

Table 7.5: These results compare overhead (O_{int}) for several different grid sizes and fault rates for the LCS problem.

Fault_model	Grid-size	Fault Rate	O_{int}	$Pristine\_result$	IL	ILR
Unsymmetric	(50x50)	$1E-3$	1.26	22	21	22
	(500x500)	$1E-3$	1.49	222	156	175
	(1000x1000)	$1E-4$	1.49	436	216	305
	(2500x2500)	$1E-4$	1.44	1081	490	710
Bimodal	(50x50)	$1E-3$	1.26	22	21	22
	(500x500)	$1E-3$	1.49	222	148	162
	(1000x1000)	$1E-4$	1.45	436	223	417
	(2500x2500)	$1E-4$	1.44	1081	702	474
Trimodal	(50x50)	$1E-3$	1.26	22	22	22
	(500x500)	$1E-3$	1.48	222	155	175
	(1000x1000)	$1E-4$	1.45	436	226	320
	(2500x2500)	$1E-4$	1.44	1081	469	694
Unimodal	(50x50)	$1E-3$	1.26	22	22	22
	(500x500)	$1E-3$	1.48	222	216	220
	(1000x1000)	$1E-4$	1.44	436	317	400
	(2500x2500)	$1E-4$	1.44	1081	587	899
Bitflip	(50x50)	$1E-3$	1.21	22	22	22
	(50x50)	$1E-3$	149	222	139	154
	(50x50)	$1E-4$	1.45	436	187	267
	(50x50)	$1E-4$	1.49	1081	432	633

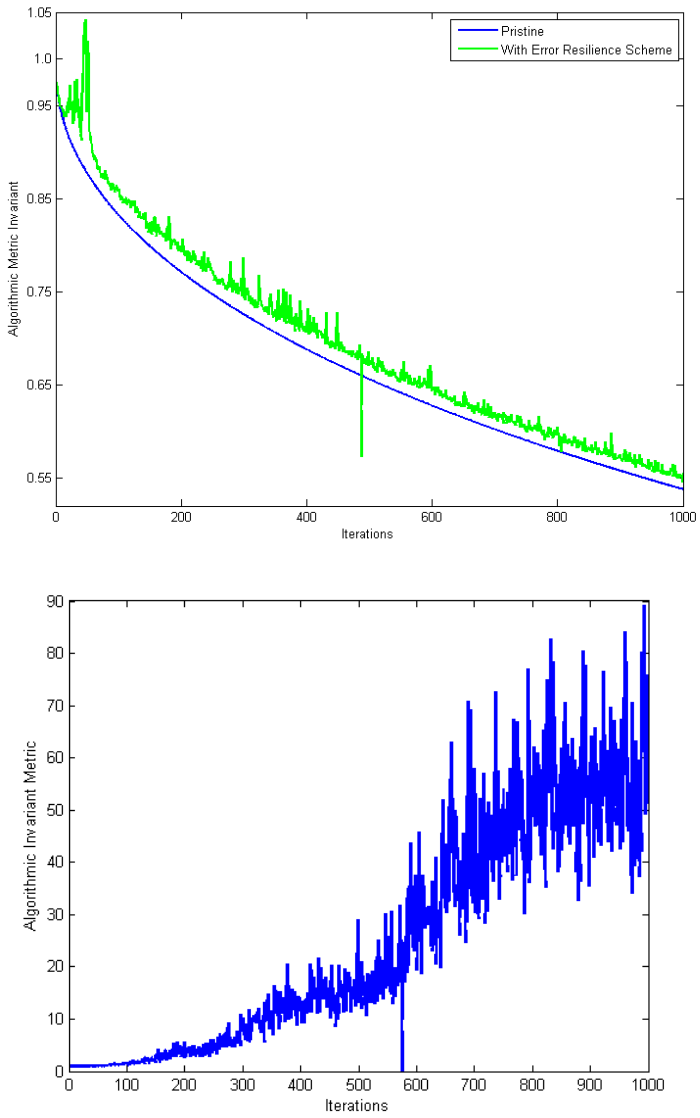

Figure 7.3: These figures show the behavior of the L2 norm error metric for Poisson equation solver on a 192×192 grid with bitflip errors injected at a rate of $1E-3$. **Top**: Our error-resilient algorithm detects outliers and achieves convergence at a similar rate as in the pristine run. **Bottom**: The original algorithm is unable to tolerate errors, and convergence is not achieved.

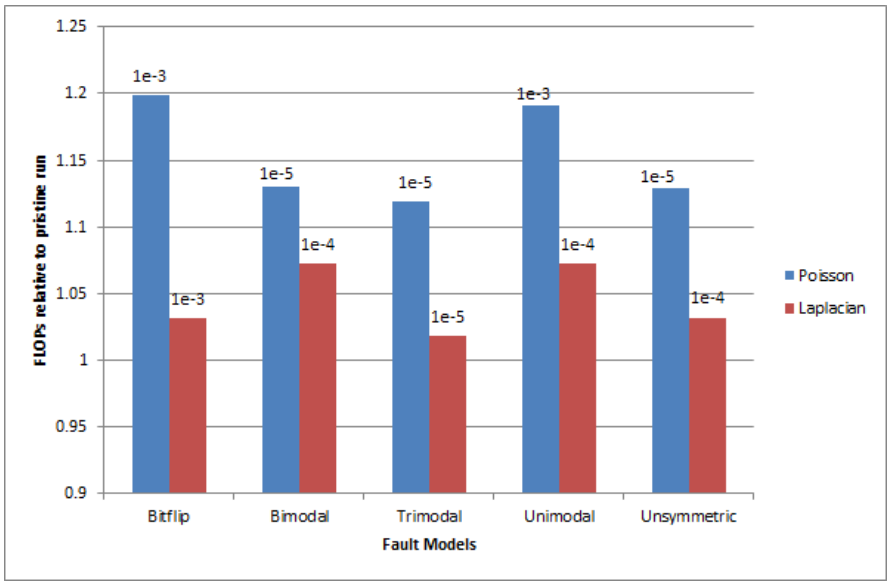

Figure 7.4: This figure shows the fault rate tolerated and overhead introduced by our error resilient Laplacian solver for different fault models on a 192×192 grid.

Chapter 8

Conclusion and Future Work

Future high-performance and energy-efficient computing system will be prone to errors and severely energy constrained. Error resilience may be required to ensure that applications can use these systems productively. In this thesis, 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 (i.e., values that do not conform to the designed distribution or behavior) 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 two important classes of application – structured grids and dynamic programming problems. Our error-resilient algorithms have signficantly lower overhead than traditional hardware and software error resilience techniques.

- For structured grid problems, 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 (across fault models) is 4.5% to 15% for error-resilient iterative

structured grid algorithms that tolerate error rates up to $10E3$ and achieve the same output quality as their error-free counterparts.

- For monadic dynamic programming problems, our error resilience techniques improve output quality by up to 40%, on average, compared to the original algorithm. Overhead is less than 59% on average – an improvement of at least 3.3× compared to existing redundancy-based techniques that achieve the same output quality.

8.1 Outlier Detection

We have demonstrated the application of outlier detection-based algorithmic error resilience for two different classes of applications that involve a combination of arithmetic and logical operations. However, the guidelines for using outlier detection, provided in Chapter 2, are not exhaustive. Additional classes of applications should be surveyed to determine the feasibility of using outlier detection-based algorithmic error resilience 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*, 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) [16]. Dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions [16]. 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 and dynamic programming 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 the 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.

8.1.1 Learning from Outlier Detection

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 or application enables efficient exploration of application knowledge that can result in low-overhead routines for robustificaiton. Using a generic technique that is not application-aware can pose drawbacks such as higher overhead, limited applicability, and difficulty in automation. For example, time-independent structured grid problems like solving of Poisson and Laplacian equations need to calculate the L2 norm between grid updates. An error resilience scheme that uses this native feature to achieve robustifcation has less overhead than a more generic approach. Similarly, in monadic dynamic programming problems the benefit of all the choices are available in table form. Applying redundancy to ensure logical correctness only while tracing the solution path of the matrix / grid, as opposed to using redundancy while calculating benefit of each choice can enable significant reduction of overhead. It is possible to selectively apply redundancy because ensuring logical correctness of choices made only along the optimal solution path (during tracing) will ensure that an optimal solution can be found.

- Using native features of applications also enables robust methods for extracting necessary information from the application, which can be done by a compiler. For example, in our robust structured grid and dynamic programming problems, to incorporate robust algorithms, the programmer only needs to highlight a few 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 outlier detection-based approaches have been shown to be effective for multiple application classes, they may not be applicable for all classes of applications. Future work should draw draw on the highlighted features of outlier detection to develop new algorithmic error resilience strategies that will provide robustness for a wide range of application classes.

8.1.2 Extending Outlier Detection

8.2 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. Even so, it is non-trivial to determine *which* applications can be robustified using previous approaches and *how* to perform the robustification for a given application. To facilitate the process of robustification for application developers and ease the adoption of robust algorithms, we propose 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 grids and monadic dynamic programming problems. Hence through this work we wish to highlight the need for consideration of automatic program robustification, while designing novel techniques. For the same, other potential directions to consider include:

- Providing robust libraries similar to the Standard Template Library of C++. This would provide a convenient method to incorporate robust algorithms and data structures in applications. An advantage of providing robust library is, it can be incorporated using compiler flags to use robust version of STL for the application.
- Transforming existing techniques for robustification to a general format which can be handled by compilers, using a combination of parsers and pre-processor directives.

References

[1] 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.

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

[3] Joseph Sloan, Rakesh Kumar, and Greg 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, Washington, DC, USA, 2012. IEEE Computer Society.

[4] Joseph Sloan, Rakesh Kumar, and Greg 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.

[5] Victoria Hodge and Jim Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, October 2004.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

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

[8] Klaus A Hoffmann. *Computational Fluid Dynamics for Engineers.*

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

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[11] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

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

[13] N. Nakka, Z. Kalbarczyk, R.K. 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.

[14] Keun Soo Yim, Cuong 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.

[15] Mark Nixon and Alberto S. Aguado. *Feature Extraction & Image Processing, Second Edition*. Academic Press, 2nd edition, 2008.

[16] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007.

Appendix A

Programmers' Guide

In this appendix we provide details on all the required preprocessor directives and any other requirement for adopting the automatic program transformation based on outlier detection. The Sections A.1 and A.2 have details for both structured grids and dynamic programming respectively.

A.1 Structured Grids

The iterative applications for structured grid applications can utilize outlier detection-based algorithmic error resilience using the following infrastructure. All the preprocessor directives start with the term #pragma struct_grid. The required preprocessor directives are as follows:

- #pragma struct_grid *size* m, n
  This directive provides the size of the grid used in an application. This information is required to determine how to decompose the grid, to aid in error localization. Currently, only two-dimensional grids are supported, though extending the methodology to higher-dimension grids is trivial.

- #pragma struct_grid *dependency_length* k
  This directive determines the stencil length – the number of elements in each dimension required to update a point i.e, to update a point (i,j) the points ($i \pm 1$,j),($i \pm 2$,j),.. ($i \pm k$,j) and (i,$j \pm 1$), (i,$j \pm 2$),..(i,$j \pm k$) will be required. The

grid is assumed to be zero padded along the boundary in each dimension for this length. I.e, if the dimensions of the padded grid are (m,n) and dependency_length is 1 then the row 0, row m-1, column 0, and column n-1 are expected to be filled with zeros. Likewise, if the unpadded grid dimensions are (m,n), then row 0, row m+1, column 0, and column n+1 in the padded grid are filled with zeros.

- #pragma struct_grid *curr_array my_curr_array*
  This directive indicates the array that stores the grid computed in the current iteration of grid update.

- #pragma struct_grid *prev_array my_previous_array*
  This directive indicates the array that stores the grid computed during the previous iteration of grid update.

- #pragma struct_grid *data_layout n*
  This directive indicates the dimensionality of the grid arrays – $n = 1$ for one-dimensional data layout and $n = 2$ for two-dimensional data layout.

- #pragma struct_grid insert_here
  This directive indicates the location where the method that implements outlier detection-based error resilience will be placed. This pragma should be placed at the end of the outermost loop that specifies the number of grid update iterations that will be performed. This method is called after every iteration to check for outliers (as per the robust algorithm) and if needed corrects any detected errors.

The following program example demonstrates the use of preprocessor directives to mark the key variables that will be used in the automatic robustification of the code.

Example: The code which uses our automatic program transformaiton tool for structured grids will look like in Figure A.1 and the transformed output of the tool is as shown in Figure A.2

```
for (int t = 0; t < num_times; t++)
{
   if(t%2)
   {
      heat_mat1=A;
      heat_mat2=B;
   }
   else
   {
      heat_mat1=B;
      heat_mat2=A;
   }
   #pragma struct_grid size m,n
   #pragma struct_grid curr_array heat_mat1
   #pragma struct_grid prev_array heat_mat2
   #pragma struct_grid data_layout 1
   for (int row_idx = dependency_length ; row_idx < (num_rows - dependency_length); row_
   {
      for (int col_idx = dependency_length ; col_idx < (num_cols-dependency_length); col
      {
         int index= row_idx*num_cols + col_idx;
 heat_mat1[index] =   0.125 * ( heat_mat2[index+num_cols] - 2.0 * (heat_mat2[index]
 + heat_mat2[index-num_cols] ) + 0.125 * (heat_mat2[index+1] - 2.0 * heat_mat2[index]
 + heat_mat2[index-1]) + heat_mat2[index];
      }
   }
   #pragma struct_grid insert_here
}
```

Figure A.1: The pristine code for a typical iterative structured grids application, with necessary preprocessr directives for the automatic program transformation tool.

```
for (int t = 0; t < num_times; t++)
{
        if(t%2)
{
heat_mat1=A;
heat_mat2=B;
}
else
{
heat_mat1=B;
heat_mat2=A;
}
for (int row_idx = dependency_length ; row_idx < (num_rows - dependency_length)
             ; row_idx++)
{
for (int col_idx = dependency_length ; col_idx < (num_cols-dependency_length)
                     ; col_idx++)
{
          int index= row_idx*num_cols + col_idx;
  heat_mat1[index] =   0.125 * ( heat_mat2[index+num_cols] - 2.0 * (heat_mat2[index]
  + heat_mat2[index-num_cols] ) + 0.125 * (heat_mat2[index+1] - 2.0 * heat_mat2[index]
  + heat_mat2[index-1]) + heat_mat2[index];

}
}
 grid_analyze_per_quadrant(heat_mat1, heat_mat2, m,n);
        // This method will take care of error detection and correction.
}
```

Figure A.2: The robust code with necessary changes as per the robustification based on outlier detection for structured grids application.

A.2 Dynamic Programming

In this section we describe the automated robustification based on outlier detection for monadic dynamic programming applications. The framework to adopt automated robustification is presented below. All the preprocessor directives start with the term #pragma dynamic_prog. The required preprocessor directives are as follows:

- The following directives should appear before specifying any directive involving either *solve* or *trace* parameters.
  - #pragma dynamic_prog *mat dimensions* n1 i,j
    This directive indicates the number of indices and the variable names for the the indices, respectively.
  - #pragma dynamic_prog *mat size* $size$1,$size$2
    This directive provides the size of the grid. Currently, only two-dimensional grids are supported.
  - #pragma dynamic_prog *mat array lengths*
    This directive indicates the grid variable.
  - #pragma dynamic_prog *mat declaration int∗ grid*
    This directive indicates the grid variable declaration. This information is required for the rollback method.
- #pragma dynamic_prog *solve num_conditions* n
  This directive indicates the number of conditions that are involved in evaluating the benefit of a choice. All comparisons should be included in this count (both if and else conditions). Also, all conditions should be written in if-else form. For example, the benefit assignment statement $b = max(a, b)$; can be written as $if(a > b)a; elseb;$. We intend to provide macros for common functions (max, min, conditionals, etc.) that are compatible with our automated error resilience framework.
- #pragma dynamic_prog *solve cond* k
  This directive indicates that the following line of code contains condition number k related to evaluating the benefit of a choice. If $k > n$ the following condition will

not be processed by the parser. n directives of this form are expected, indicating the n conditions, as specified. Failure to provide the necessary n conditions will result in a parsing error.

- #pragma dynamic_prog *inner_loop_solve open*
  This directive indicates that the inner loop of the code that calculates the benefit will start on the next line of code.

- #pragma dynamic_prog *inner_loop_solve close*
  This directive indicates that the inner loop of the code that calculates the benefits will end on the next line of code.

- #pragma dynamic_prog *trace num_conditions m*
  This directive indicates the number of conditions that are involved in tracing the grid to determine the optimal solution. All comparisons should be included in this count (both if and else conditions). As described above, all conditions should be written in if-else form. Normally $m = n$.

- #pragma dynamic_prog *trace cond l solve cond k*
  This directive indicates that the next line of code contains condition number l related to tracing the grid and also indicates which *solve* condition should be true for this *trace* condition l to be true. If $l > m$ the following condition will not be processed by the parser. m directives of this form are expected, indicating the m conditions, as specified. Failure to provide the necessary m conditions will result in a parsing error.

- #pragma dynamic_prog *inner_loop_trace open*
  This directive indicates that the inner loop of the code responsible for tracing the grid starts on the next line of code.

- #pragma dynamic_prog *inner_loop_trace close*
  This directive indicates that the inner loop of the code responsible for tracing the grid ends on the next line of code.

- #pragma dynamic_prog *trace accept_choice*
  This directive indicates the code that selects a choice (i, j) as part of the optimal

solution. This directive should be specified immediately before the piece of code where the choice is accepted as part of the optimal solution. See the code below for an example of locating this region of code.

Example: The code which uses our automatic program transformaiton tool for dynamic programming will look like in Figure A.3 and the transformed output of the tool is as shown in Figure A.4 and A.5

```
    #pragma dynamic_prog mat dimensions 2 i,j
    #pragma dynamic_prog mat size 500, 500
    #pragma dynamic_prog mat array lengths
    #pragma dynamic_prog solve num_conditions 3
    for (i=0,i < lena;i++)
    {
         x=a[i];
      #pragma dynamic_prog inner_loop_solve open
         for (j=0; j < lenb ;y++ )
        {
            y=b[j];
           #pragma dynamic_prog solve cond 1
            if (a[i] == b[j])
             lengths[i+1][j+1] = lengths[i][j] + 1;
            #pragma dynamic_prog solve cond 2
            else if (  lengths[i+1][j] > lengths[i][j+1])
             lengths[i+1][j+1] = lengths[i+1][j] ;
    #pragma dynamic_prog solve cond 3
            else
               lengths[i+1][j+1] = lengths[i][j+1];
       }
      #pragma dynamic_prog inner_loop_solve close
    }
     #pragma dynamic_prog solve num_conditions 3
   #pragma dynamic_prog solve cond 4
    result = bufr+bufrlen;
    % *result--='0';
    i = lena-1; j = lenb-1;
  #pragma dynamic_prog trace num_conditions 3
    while ( (i>0) & (j>0) )
    {
#pragma dynamic_prog trace cond 1 solve cond 2
        if (lengths[i][j] == lengths[i-1][j])
         i -= 1;
       #pragma dynamic_prog trace cond 2 solve cond 3
        else if (lengths[i][j] == lengths[i][j-1])
          j-= 1;
        #pragma dynamic_prog trace cond 3 solve cond 1
        else
            *--result = a[i-1];
            #pragma dynamic_prog accept_trace_choice
            i-=1; j-=1;
    }
```

Figure A.3: The pristine code for a typical mondaic dynamic programming applicaiton, with necessary preprocessr directives for the automatic program transformation tool.

```
#include "robust_dynamic_prog.h"
int curr_checkpoint_limit;
curr_checkpoint_limit=checkpoint_length;
int checkpoint_length_plus1=checkpoint_length+1;
int yet_to_rollback_prev_item=1;
for (i=0,i < lena;i++)
{
x=a[i];
for( int checkpoint_zone=0; checkpoint_zone < (num_checkpoints) ; checkpoint_zone++ )
{
int j=checkpoint_zone * checkpoint_length;
int checkpoint_limit= j + checkpoint_length;
if( lenb_minus1< checkpoint_length )
     checkpoint_length=lenb_minus1;
curr_checkpoint_limit=curr_item_1_checkpoints[checkpoint_zone]+1;
int max_in_zone;
max_in_zone=0;
for (;j<checkpoint_limit;j++ )
{
    y=b[j];

    if (a[i] == b[j])
     lengths[i+1][j+1] = lengths[i][j] + 1;

    else if (  lengths[i+1][j] > lengths[i][j+1])
     lengths[i+1][j+1] = lengths[i+1][j] ;

    else
       lengths[i+1][j+1] = lengths[i][j+1];
       }

    if( max_in_zone > curr_checkpoint_limit )
    {
        if( yet_to_rollback_prev_item ==1 )
        {
            yet_to_rollback_prev_item=0;
            checkpoint_zone--;
        }
        else
        {
            yet_to_rollback_prev_item=1;
            checkpoint_zone--;
            i--;
        }
    }
}
}
```

Figure A.4: The transformed code with necessary code changes to adopt outlier detection intervals.

```
result = bufr+bufrlen;
i = lena-1; j = lenb-1;

  bool t1= tmr_char_comparison(a[i],b[j]);
bool t2= tmr_greater(lengths[i+1][j],lengths[i][j+1]);


while ( (i>0) & (j>0) )
{

    if ( (lengths[i][j] == lengths[i-1][j]) && (~t1) && (t2) )
    i -= 1;

    else if (lengths[i][j] == lengths[i][j-1] && (~t1) && (~t2) )
    j-= 1;

    else if( (t2) && (~t1) )
    {
        *--result = a[i-1];
        i-=1; j-=1;// Following variables are defined in the header "robust_dynamic_prog.h"
        top_of_stack=(top_of_stack+1)%(size_stack);
        end_of_stack=((top_of_stack+1)%size_stack);
        accepted_tuple_chars[top_of_stack].my_tuple.i=i;
        accepted_tuple_chars[top_of_stack].my_tuple.j=j;
    }
    else
    {
        xy_tuple fixing_end_tuple=rollback(i,j,accepted_i,accepted_j,lengths[i][j],a,b,error_percent);
        if( !( (fixing_end_tuple.i==0) && (fixing_end_tuple.j==0) && ( top_of_stack <0) ) )
        {
            rollback_tuple_pos rolling_back;//=search_stack(fixing_end_tuple,result_pos);
            last_considered_i=i;last_considered_j=j;
            if( (rolling_back.my_tuple.i) && (rolling_back.my_tuple.j) )
            {
                i=rolling_back.my_tuple.i-1;
                j=rolling_back.my_tuple.j-1;
                result_pos-=rolling_back.result_position_adjust;
            }
            else
            {
                i=lena-1;
                j=lenb-1;
                result_pos=0;
            }

             ij_considered=0;
        }
        else
        {
            i=lena-1;
            j=lenb-1;
            result_pos=0;
        }
    }
}
```

Figure A.5: The transformed code with necessary code changes for *Redundancy while tracing* and *Rollback* steps.