Automatic Algorithm-Based Fault Detection (AABFD) of Stencil Computations

Louis Narmour
Univ Rennes, Inria, CNRS, IRISA,
Colorado State University
France/USA
louis.narmour@irisa.fr

Steven Derrien
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
steven.derrien@irisa.fr

Sanjay Rajopadhye
Colorado State University
Fort Collins, USA
sanjay.rajopadhye@colostate.edu

Abstract

In this work, we study fault tolerance of transient errors, such as those occurring due to cosmic radiation or hardware component aging and degradation, using Algorithm-Based Fault Tolerance (ABFT). ABFT methods typically work by adding some additional computation in the form of invariant checksums which, by definition, should not change as the program executes. By computing and monitoring checksums, it is possible to detect errors by observing differences in the checksum values. However, this is challenging for two key reasons: (1) it requires careful manual analysis of the input program, and (2) care must be taken to subsequently carry out the checksum computations efficiently enough for it to be worth it. Prior work has shown how to apply ABFT schemes with low overhead for a variety of input programs. Here, we focus on a subclass of programs called stencil applications, an important class of computations found widely in various scientific computing domains. We propose a new compilation scheme to automatically analyze and generate the checksum computations. To the best of our knowledge, this is the first work to do such a thing in a compiler. We show that low overhead code can be easily generated and provide a preliminary evaluation of the tradeoff between performance and effectiveness.

Keywords: fault-tolerance, program transformations, polyhedral compilation, stencils

ACM Reference Format:
Louis Narmour, Steven Derrien, and Sanjay Rajopadhye. . Automatic Algorithm-Based Fault Detection (AABFD) of Stencil Computations. In *Proceedings of IMPACT '23: 12th International Workshop on Polyhedral Compilation Techniques (IMPACT '23).* ACM, New York, NY, USA, 11 pages.

1 Introduction

Computing technology continues to move in the direction of *more* capability and *more* complexity in *less* space with *less* power. Transient silent errors that originate in the hardware and manifest as silent data corruption pose a serious concern for software reliability. Such errors occur due to phenomena such as cosmic radiation [1, 2] and hardware component aging and degradation [3, 4]. This is a problem of both size and scale. At one end of the spectrum, as trends push the development of smaller and lower power systems, the likelihood of encountering such errors increases [5, 6]. On the other end, silent data corruption errors have even been observed in large-scale infrastructure services running at the global scale as recently as 2021 [7]. This also has implications in the realm of High-Performance Computing (HPC) [8–10] where workloads can run for weeks and even months. Waiting months for the computation to finish only to realize that the result was incorrect has a significant impact given the time wasted. The need for robust fault tolerance today is becoming more and more prevalent as computing platforms grow in capability and complexity.

One approach to enable fault tolerance involves duplication either in the software [11, 12] or in the hardware [13]. Duplication has the highest coverage, but also the highest overhead. Other approaches avoid duplication and employ compile-time analysis to augment the software with checksums to detect errors in the memory [14]. While this is less expensive, it also has lower coverage. Silent errors that happen inside the floating-point arithmetic units, for example, go undetected. Detection of silent transient errors is difficult because the errors manifest *in the data* and doing so requires running some analysis *on the computed data*, which most hardware-based fault tolerance schemes ignore.

Algorithm-based fault tolerance (ABFT) [15] has been widely studied since it was first proposed in 1984 and provides a relatively cheap way to detect, and correct, such errors *in the data*. The main idea is to augment the computation with extra work in the form of invariant checksums by exploiting algebraic identities, which by definition should remain constant valued as the program executes. By comparing checksums evaluated periodically as the program executes, we can detect errors if their difference is above some threshold. ABFT schemes have been studied on a variety of different computational kernels from dense linear algebra such as Fast-Fourier Transform networks, matrix multiplication, and more recently convolutional neural networks and stencil computations [16–22].

In this work, we focus on ABFT in the context of scientific computing. Specifically, we focus on stencil applications which are commonly used to compute approximate solutions

IMPACT '23, January 16, 2023, Toulouse, France
2023.

to partial differential equations modeling physical phenomena such as (electro)dynamic wave propagation and heat flow. Even among just stencil computations, there is a wide range of variability. The structure of a particular stencil depends on the specific properties of the particular phenomenon under study, such as the rate at which heat flows through (potentially multiple and different) physical media, just to give an example. These flow rates may be the same in all physical directions, referred to as *isotropic* diffusion in the literature, or direction-dependent, *anisotropic* [23]. To complicate things even further, these rates need not be the same everywhere, they may have different *magnitudes* at different physical points in space. All of these factors influence the particular form of the input stencil program, which directly affects the analysis required to carry out ABFT.

There are several challenges that make the application of algorithm-based fault tolerance difficult:

1. identification of the ABFT-applicable regions of the input program (and there may be multiple independent regions)
2. construction of the invariant checksums
3. computing the checksums efficiently enough to be worth it (i.e., with low overhead)

Each of these is largely dependent on the input program and requires very careful and manual analysis. However, there is no such *ABFT-compiler* (yet) that can perform all of the above from the input program alone. In this work, we propose a new methodology to carry out this analysis at compile time, embodied as a sequence of program transformations, which can be automated thanks to polyhedral compilation techniques. To this end, we make the following contributions:

- We show how to carry out an ABFT analysis automatically in a compiler.
- We present some preliminary data on the tradeoff between overhead and error detection effectiveness of the generated code.

The rest of this paper is organized as follows. In Section 2, we provide several motivating examples showing how ABFT works on stencils illustrating the challenges mentioned above. The framework and scope of our analysis are reviewed in Section 3 followed by the description of our compiler passes in Section 4 and preliminary performance data and a discussion with respect to execution time (i.e., overhead) and effectiveness (i.e., ability to reliably detect errors) in Sections 5. Finally, we discuss related work and open questions in Sections 6 and 7.

2 Motivating Examples

Algorithm-based fault tolerance works fundamentally by defining invariant checksums over subsets of the domains of the computed variables and then asserting that the checksums do not change as the program evolves. For each checksum (in general, there may be multiple) this involves constructing two algebraically equivalent expressions that compute the same value, each taken from different program slices, and then asserting that their difference is zero[1].

Stencils are typically implemented as a series of weighted convolutions. The properties mentioned above manifest in the code as different weight values and potentially different convolution domains at different points in the domain. This directly influences where it is possible to construct the invariant checksum expressions. We illustrate this variability with several examples of increasing complexity in the following sections. One quickly appreciates the difficulty of doing this manually and can see why it is desirable to leave such analysis to a polyhedral compiler.

2.1 1D Jacobi stencil with constant weights

The Jacobi 1D stencil updates an $(N+1)$-element array over a series of T time steps. The primary computation of each non-boundary point comes from the weighted sum of three neighboring points from the previous time step.

$$B_{t,i} = \begin{cases} A_i & \text{if } t = 0 \\ B_{t-1,i} & \text{elif } 0 < t \leq T \text{ and } i = 0 \text{ or } i = N \\ w_0 B_{t-1,i-1} + w_1 B_{t-1,i} + w_2 B_{t-1,i+1} & \text{else} \end{cases} \tag{1}$$

In stencils, deriving the invariant checksum is achieved by building two expressions such that no two points with the same time step (i.e., the t index value of $B_{t,i}$ in Equation 1) appear in both expressions.

2.1.1 Construct invariant checksums. Let us introduce a new variable, $C_{t,l,m}$, to denote the checksum at time step t defined as the sum over the window of values of $B_{t,i}$ for $l \leq i \leq m$,

$$C_{t,l,m} = \sum_{i=l}^{m} B_{t,i} \qquad 1 \leq l \leq m \leq N-1 \tag{2}$$

This is illustrated as the solid boxed region in Figure 1. By substituting the definition of $B_{t,i}$ where $1 \leq l \leq m \leq N-1$ from Equation 1 into Equation 2 we obtain,

$$C_{t,l,m} = \sum_{i=l}^{m} \left(w_0 B_{t-1,i-1} + w_1 B_{t-1,i} + w_2 B_{t-1,i+1} \right) \tag{3}$$

Notice that the points of B can be grouped based on the combination of w_0, w_1, and w_2 through which they contribute, shown by the dashed boxes in Figure 1. The middle dashed rectangle represents the points at time step $t-1$ that contribute to C_t by all three weights. After some algebra we

[1]In reality, we deal with floating point arithmetic which is not associative so the difference will never truly be zero. Instead, we can assert that it is below some sufficiently small threshold.

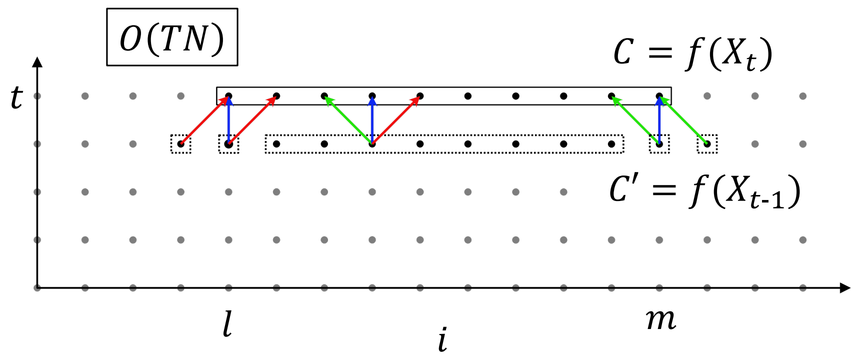

Figure 1. Single checksum illustration. Points contributing to $C_{t,l,m}$ in Equation 2 shown in the solid boxed region and points contributing to $C'_{t,l,m}$ in Equation 4 in dashed boxed regions. Repeating this each time step at every $(m-l)$'th position along i leads to an overall cost of $O(TN)$.

obtain,

$$\begin{aligned} C'_{t,l,m} \equiv\ & (w_0)B_{t\text{-}1,l\text{-}1} + (w_0 + w_1)B_{t\text{-}1,l} \\ & + (w_0 + w_1 + w_2) \sum_{i=l+1}^{m-1} B_{t\text{-}1,i} \\ & + (w_1 + w_2)B_{t\text{-}1,m} + (w_2)B_{t\text{-}1,m+1} \qquad (4) \end{aligned}$$

which we will denote as $C'_{t,l,m}$. The right-hand sides of Equations 2 and 4 both compute the *same numerical value* using *different elements* in the domain of the variable B. Equation 2 uses elements solely from time step t, and Equation 4 from time step $t-1$.

The only way for $C_{t,l,m}$ and $C'_{t,l,m}$ to evaluate to different values is if some error were to occur between their computation. To use this, we assert that their difference,

$$\Delta C_{t,l,m} \equiv C'_{t,l,m} - C_{t,l,m}$$

is below some threshold, large enough to be distinguishable from floating-point round-off errors, and small enough to actually detect most errors. This is discussed further in Section 5.

2.1.2 Achieve low overhead with interpolation.

For any fault tolerance scheme to be feasible, the cost of implementing it needs to be sufficiently low. However, computing and comparing checksums as shown in the previous section has a significant amount of overhead. This is because the work required to compute $\Delta C_{t,l,m}$ from Equation 2 in Figure 1 is asymptotically the same as the main stencil computation. The overhead here is too high, however, it is possible to do better.

Instead of comparing expressions across *adjacent* time steps where,

$$C' = f(B_{t-1,i}) \qquad (5)$$

we could compare them across *several*, say τ, time steps such that C' is a function of only the points in B at time step $t-\tau$,

$$C' = f(B_{t-\tau,i}) \qquad (6)$$

To achieve this, we can repeatedly substitute B in C'. After two substitutions we will have $C' = f(B_{t-2,i})$, after three substitutions $C' = f(B_{t-3,i})$, and so on as illustrated in Figure 2. At each step, we need to account for the combinations of weights. As we show in Section 4.1.5 since the checksum accumulates points computed by a convolution, this leads to a very regular pattern, but still, this is not something one would want to do by hand.

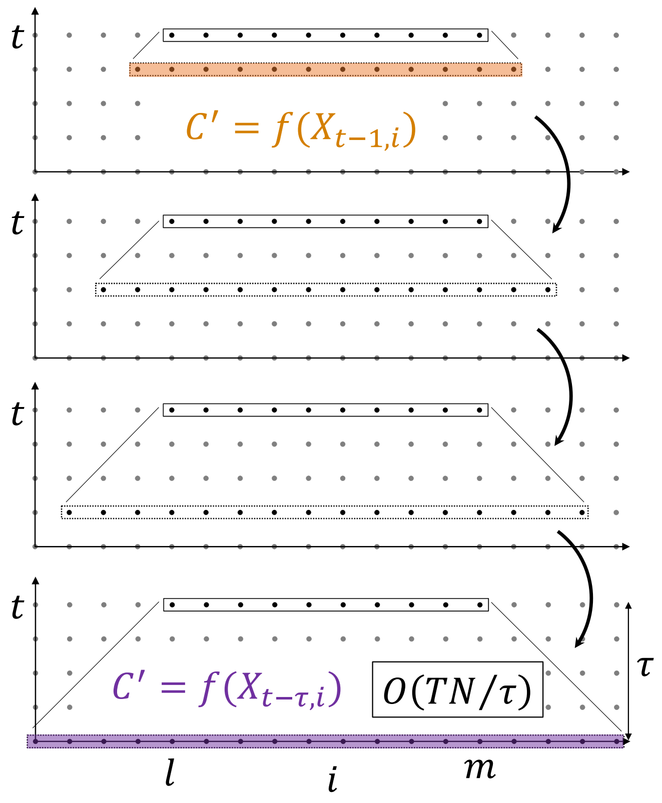

Figure 2. Interpolated checksum illustration after $\tau = 4$ repeated substitutions. Interpolation is more efficient than Figure 1.

Expressing C' as a function of $B_{t-\tau,i}$ has the same effect as what prior work [22] refers to as "interpolating" checksums solely from other checksums at previous iterations. However, we have shown the interpolation here in the opposite order (i.e., from t to $t-1$). Regardless of the order in which it is done, this process of interpolation is critically necessary to be able to carry out the checksum computations with low overhead. The intuition behind this can be understood by comparing the complexities of the non-interpolated scheme (Figure 1) and the interpolated scheme (Figure 2). The non-interpolated version is $O(TN)$ which has the same complexity up to a constant order as the stencil computation itself. The interpolated version, on the other hand, has the complexity $O(TN/\tau)$. This tunable parameter τ directly influences the complexity of checksum computation and consequently the overhead.

2.2 Stencils with multiple time dependencies

The interpolation process as described above is straightforward when there are only stencil dependencies from t to $t-1$, but consider the same example now with one additional dependency on $t-2$,

$$B_{t,i} = \begin{cases} A_i & \text{if } 0 \le t < 2 \\ B_{t-1,i} & \text{elif } 2 \le t \le T \text{ and } i = 0 \text{ or } i = N \\ w_0 B_{t-1,i-1} + w_1 B_{t-1,i} + w_2 B_{t-1,i+1} & \\ \qquad + w_3 B_{t-2,i} & \text{else} \end{cases} \tag{7}$$

adapted from finite-difference time-domain (FDTD) acoustic wave simulation code [24].

In this example, the process of repeated substitution results in an alternate expression of the form,

$$C' = f(B_{t-\tau,i}) + g(B_{t-\tau-1,i}) \tag{8}$$

for two different functions f and g. This is because each substitution introduces terms across the previous *two* time steps. The difficulty of identifying the correct combinations of weights here is magnified. Again, this is not something that we would want to do by hand. We show how to handle this example in Section 4.1.5 but the reader is encouraged to try to work out the equivalent version of Equation 4 and perform τ repeated substitutions obtaining the expressions for f and g to fully appreciate this for themselves.

2.3 Stencils with variable weights

Not all stencil programs are directly amenable to ABFT because it may not always be possible to do the interpolation step described in Section 2.1.2 as illustrated by the following example in Equation 9. The reason it was possible to rewrite Equation 2 as shown in Equation 4 was that the weight expressions were constant at each point in space allowing them to be factored out of the middle summation term.

Consider this slightly modified piece of stencil code in Equation 9, which is identical to Equation 1 except now there may be unique weight values at each point in space.

$$B_{t,i} = \begin{cases} A_i & \text{if } t = 0 \\ B_{t-1,i} & \text{elif } 0 < t \le T \text{ and } i = 0 \text{ or } i = N \\ w_{i-1} B_{t-1,i-1} + w_i B_{t-1,i} + w_{i+1} B_{t-1,i+1} & \text{else} \end{cases} \tag{9}$$

It is necessary to further inspect the definition of the value of the weights, w_i before proceeding. If there exist regions with constant weights, as is common in *isotropic* stencils [23] with absorbing boundaries for example, then it is possible to carry out ABFT with low overhead. For example, isotropic stencils with absorbing boundaries typically have weights with varying values near the boundaries (i.e., close to $i = 0$ and $i = N$ in our example here) but constant "in the middle". The definition of w_i could be of the following form,

$$w_i = \begin{cases} f(i) & 0 \le i < M \\ 0.333 & M \le i \le N - M \\ f(i) & N - M < i \le N \end{cases} \tag{10}$$

where f is some arbitrary non-constant function. In this case, it is possible to do ABFT, just not on the entire domain. The solution is to recognize that the third branch of Equation 9 can be split into three branches, one for each branch in Equation 10, of which the middle branch then has the desired constant-weights form. This is straightforward for a clever human to see, but much more difficult for a classical compiler that reasons at the statement level. All of these challenges are magnified in higher dimensions and beyond the most simple symmetrical stencils implementations, this is not something that we would want to do by hand. This kind of analysis is aptly suited to polyhedral compilation which enables reasoning at the statement *instance* level.

3 Polyhedral program representation

The polyhedral model [25–30] is a mathematical formalism for reasoning about a precisely defined class of computations. It provides the technology to map high-level descriptions of compute- and data-intensive programs to a range of highly parallel targets. Polyhedral "programs" are most cleanly viewed as *equations* defined over *polyhedral domains*, evaluating an *expression* at each point therein. In the context of ABFT, it enables us to reason precisely at the program statement instance level to construct the compact sets characterizing the hyper-trapezoidal regions illustrated previously in Figure 2. The Alpha language [31, 32] is a high-level equational language that separates the specification of the program from its execution plan. The semantics of an Alpha program closely follows the program's equivalent equational representation.

The task of extracting the polyhedral representation (i.e., the set of affine recurrence equations) from a series of nested loops with affine control structure has been well studied [33].

Given a set of affine recurrence equations, one can subsequently write the equivalent Alpha program in a straightforward manner. We do not review how to do this here and for the remainder of our discussion, we assume that the input stencil program under study is itself an Alpha program.

In Figure 3, we provide an Alpha implementation of the 1D FDTD stencil from Section 2.2. This will be our working example for the remainder of the paper. Note the layout, with the following main sections of code: inputs, outputs, locals, and equations. The domains of the program variables

```
 0: affine fdtd [T,N,M]->{: 1<M<N and 2<T}
 1:   inputs
 2:     A : {[i] : 0<=i<=N}
 3:   outputs
 4:     A_out : {[i] : 0<=i<=N}
 5:   locals
 6:     B : {[t,i] : 0<=t<=T and 0<=i<=N}
 7:     w0,w1 : {[i] : 0<=i<=N}
 8:   let
 9:     w0[i] = case {
10:       {: 0<=i<M or N-M<i<=N } : f(i);
11:       {: M<=i<=N-M } : 0.284;
12:     };
13:     w1[i] = case {
14:       {: 0<=i<M or N-M<i<=N } : f(i);
15:       {: M<=i<=N-M } : 0.148;
16:     };
17:
18:     B[t,i] = case {
19:       {: 0<=t<2} : A[i];
20:       {: t>2 and (i=0 or i=N)} : B[t-1,i];
21:       {: t>2 and 0<i<N} : w0[i-1]*B[t-1,i-1]
                          + w0[i]*B[t-1,i]
                          + w0[i+1]*B[t-1,i+1]
                          + w1[i]*B[t-2,i];
22:     };
23:     A_out[i] = B[T,i];
24: .
```

Figure 3. 1D FDTD Alpha program with multiple time dependencies (from Section 2.2) and regions with variable weights (Section 2.3).

are represented as sets of integer points with a syntax that closely follows that of `isl` (the integer set library) [34]. Local variables are used only in the context of this program (i.e., they only have *scope* in this program).

3.1 Equations

Alpha equations are of the form below where Y is a program variable, E is an expression defined over the domain $\mathcal{D}_E$ and f is an affine function,

$$Y[f(z)] = E[z] \qquad \forall z \in \mathcal{D}_E$$

The expression E is evaluated at each point z in $\mathcal{D}_E$ and the answer is written to the location in the variable Y specified by $f(z)$. Each Alpha expression is associated with two polyhedral domains. First, the ***expression domain*** of E is the domain over which E is well defined and is computed *bottom-up*. Next, the ***context domain*** of E is the domain over which it needs to be evaluated (i.e., where each value that contributes to an answer in the output needs to be written) and is computed *top-down*. For example, the expression $B[T, i]$ in line 23 of Figure 3 has the expression domain, $[T, N] \rightarrow \{[t, i] : 0 \leq t \leq T \text{ and } 0 \leq i \leq N\}$ with $O(TN)$ points because this is the domain over which the variable B is defined. Since it only contributes to an answer in *A_out* when $t = T$, its context domain is $[T, N] \rightarrow \{[t, i] : t = T \text{ and } 0 \leq i \leq N\}$ with $O(N)$ points. The important thing to keep in mind is that Alpha expressions are defined over polyhedral domains and when we use point-wise expressions like this, we are implicitly handling subsets of these domains.

The Alpha grammar formally defines and supports many special types of expressions. We will only review the aspects minimally needed to make our discussion in this paper self-contained.

3.2 Dependence expressions

A ***dependence expression*** is an expression of the following form, where f is an affine function and z is a point in the expression domain of the expression E,

$$E[f(z)]$$

and should be understood as "the expression E evaluated at the point mapped from z by f" or equivalently as, "read the expression E at the point $f(z)$". All of the variable accesses in Figure 3 (e.g., w$[i-1]$, B$[t-1, i+1]$, etc.) are dependence expressions.

3.3 Restrict expressions

A ***restrict expression*** is an expression of the following form, where $\mathcal{D}$ is some domain,

$$\mathcal{D} : E$$

which should be read as "the expression E restricted to the subdomain $\mathcal{D}$". The expression domain of this restrict expression is the intersection of $\mathcal{D}$ with the expression domain of E.

3.4 Case expressions

Expressions can also be piecewise expressions defined over multiple semicolon-delimited disjoint pieces using ***case expressions***, which have the following form,

$$case\ \ \{\mathcal{D}_0 : E_0;\ \ \mathcal{D}_1 : E_1;\ \ \mathcal{D}_2 : E_2;\ \ ...\}$$

for any number of disjoint domains $\mathcal{D}_i$. Here, each of the pieces is itself a restrict expression.

3.5 Reduce expressions

Finally, the Alpha language also supports reduction expressions as first-class objects. Alpha ***reduce expressions*** generally have the following form, where Y is a program variable, E is the expression of the reduction body with the domain $\mathcal{D}_E$, and f_p is an affine function,

$$Y[f_p(z)] = \bigoplus E[z] \quad \forall z \in \mathcal{D}_E$$

where the expression E, evaluated at each point z in $\mathcal{D}_E$, is accumulated into the element of the variable Y at the location $f_p(z)$ by the $\oplus$ operator. The syntax for the reduce expression that represents this is,

$$Y[f_p(z)] = \text{reduce}(\oplus, f_p, \mathcal{D}_E : E[z])$$

3.6 Checksums as Alpha reductions

We use reductions to represent the checksum values over the program variables in the following sections. Equation 2 can be expressed in Alpha as,

```
C[t,l,m] = reduce(+, (t,l,m,i->t,l,m),
                              {: l<=i<=m } : B[t,i])
```

for some new variable C representing the *family* of all possible checksum *instances* at a particular time step t over a particular range of i for $l \leq i \leq m$. We need the family to fully cover the program as discussed in Section 4.2. The reduction body here is a restrict expression with a 4-dimensional expression domain over the indices t, l, m, and i. At each point in this domain, the program variable B is read at $B[t, i]$ and accumulated into the checksum instance at $C[t, l, m]$.

In this context, the objective is to transform this reduce expression into the equivalent efficient expression over the base of the corresponding trapezoidal domain in Figure 2. Alpha supports algebraic substitution and simplification of individual expressions, which makes this possible.

4 Automatic Checksum Derivation

To facilitate the presentation we make the following assumptions about the input stencil program,

- the outer dimension on stencil variables is interpreted as the index over time
- the equations characterizing the stencil body update points at time step t from points at strictly earlier time steps (i.e., $t - k$ for some constant $k > 0$)
- the dimensionality of all stencil variables is the same

This introduces no loss in generality because it is always possible to achieve this by reindexing the program variables. For example, consider the following Gauss-Seidel stencil equation,

$$X_{t,i} = aX_{t,i-1} + bX_{t-1,i} + cX_{t-1,i+1} \tag{11}$$

with the dependency across the same time step from $[t, i]$ to $[t, i-1]$. In such cases, we can reindex X such that all dependencies point strictly backward in time with the mapping

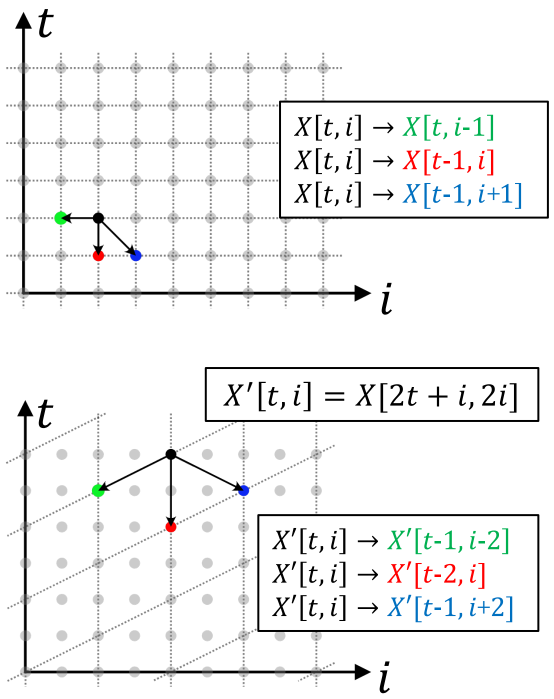

Figure 4. Normalization of dependencies in Equation 11 (top) such that all dependencies subsequently point strictly backward along time t (bottom).

$\{[t, i] \longrightarrow [2t+i, 2i]\}$ as illustrated in Figure 4. The interested reader can notice that this is the task of finding a set of valid scheduling hyperplanes.

With these assumptions in mind, principally our approach involves the following three steps to automate ABFT,

1. constructing checksums via algebraic substitution
2. replicating checksums over the program domain
3. scheduling and code generation

which are discussed in detail in the following subsections.

4.1 Step 1 - Construct invariant checksum with computer algebra

Given an input stencil program like the one provided in Figure 3, the goal of this step is to express the stencil in such a way that we can systematically build, and subsequently insert, two algebraically equivalent expressions computing the same checksum.

4.1.1 Merge variables. Some preprocessing may be required. We will call two stencil variables, X and Y, *coupled* if they belong to the same strongly connected component in the program dependency graph. For example, if the equation for X has an expression involving Y and the equation for Y in turn has some expression involving X, then we will say that X and Y are *coupled*. There could be more than two variables as well. For example, imagine that X depends on Z which depends on Y. In this case we will say that X is coupled with both Y and Z and we will refer to the set of coupled variables as $\{X, Y, Z\}$. When there exist coupled variables like this, we merge them into a single variable as follows. Let $X_i[s]$ be a dependency expression on the i'th such coupled variable. First, create a new higher dimensional variable X' with an additional dimension. Then embed the equations for each variable X_i in X' by associating the i'th index value in this additional dimension with X_i. This is purely a syntactic rewrite and is always legal.

Many user-written stencil codes simulating electromagnetism, for example, contain multiple stencil variables. Consider the following example, adapted from other FDTD codes in the MathWorks library [35], with separate variables for the electric (E) and magnetic (H) fields,

$$E_{t,i} = \begin{cases} E_{t-1,i} & t > 0 \text{ and } i = 0 \\ E_{t-1,i} - a_i(H_{t-1,i} - H_{t-1,i-1}) & t > 0 \text{ and } 0 < i \le N \end{cases}$$

$$H_{t,i} = \begin{cases} H_{t-1,i} - b_i(E_{t-1,i+1} - E_{t-1,i}) & t > 0 \text{ and } 0 \le i < N \\ H_{t-1,i} & t > 0 \text{ and } i = N \end{cases}$$

which can be merged into a single variable (M) as,

$$M_{t,i,z} = \begin{cases} M_{t-1,i,z} & \text{...and } z = 0 \\ M_{t-1,i,z} - a_i(M_{t-1,i,z+1} - M_{t-1,i-1,z+1}) & \text{...and } z = 0 \\ M_{t-1,i,z} - b_i(M_{t-1,i+1,z-1} - M_{t-1,i,z-1}) & \text{...and } z = 1 \\ M_{t-1,i,z} & \text{...and } z = 1 \end{cases}$$

with the same piecewise constraints associated with the new index z. Here z is the index associated with the new dimension, and since there are two coupled variables, z only takes on two values. The original variable E is associated with $z = 0$ and H with $z = 1$.

4.1.2 Normalize Sum of Products Expressions.

Expressions for each merged stencil variable, X, and its weights, w_i are transformed into expressions of the form,

$$w_1[f_1(z)] * X[f_1(z)] + w_2[f_2(z)] * X[f_2(z)] + ... \quad (12)$$

denoting a normal **S**ums **o**f **P**roducts (SoP) expression, where the left-hand side of each product is an expression for the weights and the right-hand side is the stencil variable. For example, the expression,

```
X[t-1,i] - a[i]*(X[t-1,i+1] - X[t-1,i])
```

is rewritten as the following normal SoP expression,

```
1*X[t-1,i] + (-1*a[i])*X[t-1,i+1] + a[i]*X[t-1,i]
```

This is purely a preprocessing step for the next step to identify the regions with constant weights.

4.1.3 Identify convolution domains.

Given two Alpha variables X and W, let us define the convolution of W with X by the operator $\bigotimes$ at the point $s \in \mathcal{D}_X$ as[2],

$$W \bigotimes X_s = \sum_{s' \in \mathcal{D}_W} W_{s'} X_{s+s'} \quad (13)$$

Given an SoP expression involving X as defined in Equation 12, we construct a definition for W. We want to identify the subdomain $\mathcal{P}$ that reads the same value from each weight expression appearing in the SoP. Then we define the value of $W[s]$ for each $s \in \mathcal{P}$ as the corresponding constant weight subexpression. Then we rewrite the portion of the SoP in $\mathcal{P}$ as the convolution defined in Equation 13.

As an example, consider the following equation,

$$X[i] = w * X[i-1] \quad (14)$$

for the 1D variable X with the domain $\mathcal{D}_X$. The binary expression $w * X[i]$ on the right-hand side is defined over $\mathcal{D}_X$, and the subexpression w is read at each point in this domain. The subexpression w here should be understood as the dependence expression that reads the scalar variable w at each point in the expression domain by the affine function $f : [i] \rightarrow []$. We say that *there is reuse of* w *in context* of its use because the null space of f has a non-empty intersection with the expression domain[3]. This notion of reuse in context is drawn from a related polyhedral optimization called simplifying reductions [36]. We leverage this to identify the reuse space common to all products in the SoP expression.

Looking again at the body of the restrict expression on line 21 of Figure 3, this reuse analysis can be run on each of the four expressions in a bottom-up fashion. The context domain $\mathcal{D}_{\text{context}}$ of this expression is $\{[t, i] : 0 \le t \le T \text{ and } 0 < i < N\}$, taken from its parent restrict expression. The first weights expression $w0[i-1]$ is only constant in the subdomain $\{[t, i] : 0 \le t \le T \text{ and } M \le i - 1 \le N - M\}$. All four weights expressions are simultaneously constant in the subdomain $\mathcal{P} = \{[t, i] : 0 \le t \le T \text{ and } M < i < N - M\}$.

From here, we split the case branch containing this SoP into two cases. In this example, we are left with the following two expressions,

```
{: t>2 and (0<i<=M or N-M<=i<N)} :
      w0[i-1]*B[t-1,i-1] + w0[i]*B[t-1,i]
    + w0[i+1]*B[t-1,i+1] + w1[i]*B[t-2,i];
{: t>2 and M<i<N-M} :   // the convolution
      0.284*B[t-1,i-1] + 0.284*B[t-1,i]
    + 0.284*B[t-1,i+1] + 0.148*B[t-2,i];
```

The second expression here with constant weights can be seen as the convolution $W \bigotimes B_s$ where $W[-1, -1]$, $W[-1, 0]$, and $W[-1, 1]$ are 0.284 and $W[-2, 0]$ is 0.148.

[2]The symbol "$*$" is commonly used denote convolutions, but to avoid conflating it with Alpha's multiplication, we use $\bigotimes$ for convolutions here.

[3]Note that we use the `isl`[34] notation to indicate that f maps onto a 0-dimensional space denoted by the empty tuple $[]$.

4.1.4 Stencil SSTC form.

Given a convolution expression involving the stencil variable X, we rewrite the convolution as, what we will call, the following **S**um of **S**ingle **T**imestep **C**onvolutions (SSTC) form,

$$X_{t,i,...} = W_1 \bigotimes X_{t-1,i,...} + W_2 \bigotimes X_{t-2,i,...} + ... \quad (15)$$

where W_k denotes the slice of W with length 1 along the time dimension such that $t - k = 0$. In other words, the index values of the time dimension for all points in $\mathcal{D}_{W_1}$ is -1, for all points in $\mathcal{D}_{W_2}$ is -2, so on and so forth. This time step separation is used subsequently in Section 4.1.5 to determine where to perform the substitutions. Looking again at the convolution identified at the end of Section 4.1.3, we can identify $\mathcal{D}_{W_1} = \{[-1,-1]; [-1,0]; [-1,1]\}$ and $\mathcal{D}_{W_2} = \{[-2,0]\}$.

4.1.5 Construction of the invariant checksums.

For each SSTC expression associated with the stencil variable X defined over the domain $\mathcal{P}$ identified by the preceding analysis, we construct one checksum expression pair C and C'. Let C be defined as the following reduction over points in Q, a rectangular subdomain of $\mathcal{P}$,

$$C[s] = \sum_{s \in Q} X[s] \quad (16)$$

Let C'_i denote the alternate checksum initialized as the SSTC expression based on the definition of X in Equation 15,

$$C'[s] = \sum_{s \in Q} \left(W_1 \bigotimes X[s] + W_2 \bigotimes X[s] + ... \right) \quad (17)$$

Now we repeatedly perform the following steps, τ times:

1. Split the reduction C' into two pieces, C'_1 and C'_2 such that C'_1 only contains the points with the largest time index and C'_2 contains everything else.

$$\begin{aligned} C' &= C'_1 + C'_2 \\ &= \sum_{s_1} (A \bigotimes X[s_1]) + \sum_{s_2} (B \bigotimes X[s_2] + ...) \end{aligned}$$

2. Substitute occurrences of $X[s_1]$ in C'_1 by its definition, the SSTC expression.
3. Recombine C'_1 and C'_2 into a single reduction in the form of Equation 17.

Convolutions are associative and commutative,

$$A \bigotimes \left(B \bigotimes X \right) = \left(A \bigotimes B \right) \bigotimes X \quad (18)$$

This property is used when C'_1 and C'_2 are recombined in step 3 above. Since the time dependencies all point strictly backward, after τ repeated substitutions, we have an expression for C' as a function of points with time step no greater than $t - \tau$. Each C and C' pair form a hyper-trapezoid as shown previously in Figure 2. Points in C reside on the "top" of the trapezoid and points in C' on the "bottom". Silent errors occurring at any point inside the trapezoid result in different computed values for C and C'.

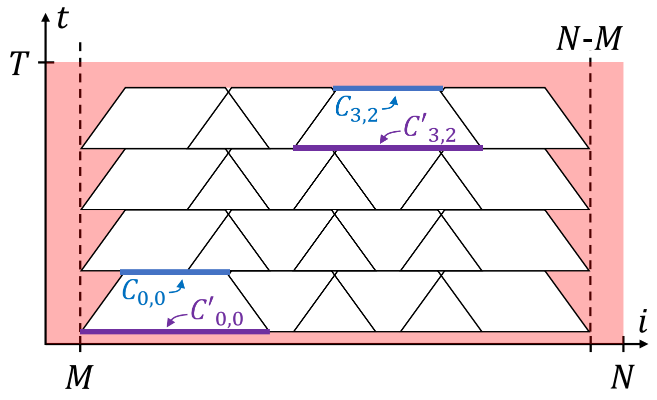

Figure 5. Striding checksum expression pairs over the corresponding convolution domain, shown here for the example in Figure 3. The subscripts $[ct, ci]$ here denote the ct'th row (from bottom to top) and ci'th column. The shaded red region illustrates the domain where duplication is needed.

4.2 Step 2 - Program Error Protection Coverage

Now we have a single checksum expression pair for each convolution appearing in the normalized program. However, each pair only enables the detection of errors within their corresponding hyper-trapezoidal subdomain. In order to use this over the rest of the convolution, we fix the trapezoid to a constant (non-parametric) size from the family of all possible checksums and then stride it over the convolution domain (very similar to tiling for all intents and purposes) as shown in Figure 5.

Still, it is not possible to cover every point. Points in the red region in Figure 5 need to be handled with some other error detection scheme because they are not contained by any trapezoid. We can use duplication here, which is not a concern from an overhead perspective since this only occurs on the boundaries with an asymptotically smaller complexity. Let $\mathcal{D}_B$ denote the domain of the stencil variable B defined in line 6 of Figure 3. Let $\mathcal{D}_{ABFT}$ denote the domain of the union of all trapezoids. Let $\mathcal{D}_{dup}$ denote the duplication domain and be defined as, $\mathcal{D}_{dup} = \mathcal{D}_B \setminus \mathcal{D}_{ABFT}$ where "\" denotes set difference. Then we add another program variable, called B_dup, using the same equation as B but restricted to $\mathcal{D}_{dup}$.

4.3 Step 3 - Scheduling and code generation

At this point, the original stencil computation and the checksum expression pairs are completely decoupled. Here we describe how to schedule the checksum computations relative to the corresponding stencil variable. Everything needs to be computed in a lock-step fashion because practical stencil implementations only use as much memory as required for two time steps' worth of computation. For example, typical implementations of the Jacobi 1D stencil in Section 2.1 look like the following code snippet,

```
for (t=1; t<=N; t++) {
  B[(t)%2][0] = B[(t-1)%2][0]
  for (i=1; i<N; i++)
    B[(t)%2][i] = w0*B[(t-1)%2][i-1] + ...
  B[(t)%2][N] = B[(t-1)%2][N]
}
```

with "modulo 2" accessing used on the time dimension. Points in the checksums need to be accumulated into C and C' before they are overwritten at the next time step iteration.

For each identified convolution in the normalized program, there are three sets:

1. $\{V[t, i_1, ..., i_d] : ...\}$, the domain $\mathcal{D}_{\mathcal{V}}$ of the corresponding variable $\mathcal{V}$
2. $\{A[ct, ci_1, ..., ci_d, t, i_1, ..., i_d] : ...\}$, the domain $\mathcal{D}_{ABFT}$ of the ABFT region (i.e., the union of trapezoids)
3. $\{D[t, i_1, ..., i_d] : ...\}$, the domain $\mathcal{D}_{dup}$ of the duplication region of $\mathcal{V}$

We can construct the schedule where at each time step, we first update the stencil variables, followed by any corresponding checksum variables if they exist, followed by duplication regions. Note that many time steps have no updates to any checksum variables since updates only occur at the bottom and top of the trapezoidal tiles. Alpha can be used to generate C code.

5 Evaluation

At the time of writing, we do not yet have a complete implementation and we hope to have a more complete evaluation in a future version of this work. Here, we present some preliminary data for Jacobi 1D stencil from Section 2.1. We measure the *overhead* in terms of execution time relative to the input stencil without any added checksum code. We measure *effectiveness* in terms of the error detection rate as we inject errors by flipping a random bit in a random location of the data grid. Both are measured as a function of the checksum size (i.e., the area of the trapezoidal regions in Figure 5). In all experiments, single-precision (32-bit) data types are used. We generated 21 different versions of the Jacobi 1D stencil using trapezoids with the following sizes, where (cT)x(cN) corresponds to the height cT and width cN: 2x8, 2x16, 2x32, 4x16, 4x32, 4x64, 8x32, 8x64, 8x128, 16x64, 16x128, 16x256, 32x128, 32x256, 32x512, 64x256, 64x512, 64x1024, 128x512, 128x1024, and 128x2048.

5.1 Overhead

We expect smaller sizes to result in slower execution times because the entire data grid must be touched more frequently. The generated codes do not include duplication code (the red region in Figure 5) at this time, so the execution times reported in our overhead experiments do not reflect this. This should not be a concern since these regions are asymptotically smaller than the main ABFT-covered regions and

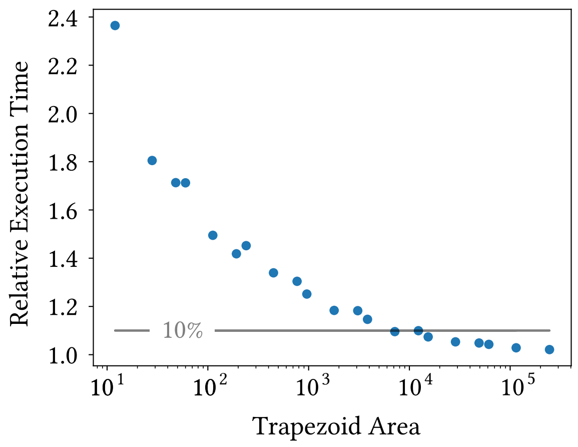

Figure 6. Overhead vs. checksum size

we wish to study the penalty incurred by scheduling the checksum computations together with the original program.

To measure the overhead, we compare the execution times of the same stencil program *with* and *without* the trapezoidal checksum regions. The ratio of execution times is reported in Figure 6. In all overhead experiments, we use a data grid size of $N = 1,000,000$ updated over $T = 1000$ time steps. As expected, we see that for sufficiently large trapezoids (consisting of more than 10^4 elements) the overhead drops below 10%.

5.2 Detection Accuracy

For each ABFT-augmented stencil version, we flip a single bit at a random location in the data grid at a random time step. We use a data grid size of $N = 12,000$ updated over $T = 100$ time steps. For each bit flip position, we run 100 trials. If there was at least one checksum expression pair with a difference ($\Delta C_{i,j} = C_{i,j} - C'_{i,j}$ from Figure 5) above the threshold[4] 10^{-5}, then the run is counted as a success. The fraction of total successes is reported for each checksum size and bit flip position in Figure 7.

We can see that even for very small checksum sizes, not *all* bit flips (below the 10th least significant bit) can be detected. It is likely that flips to these bit positions should not even be treated as *true errors*, but we have not studied this here. As we expect, the detection rate drops as the trapezoid size becomes large enough, even for the higher order bits (blue and green in Figure 7). We see a sharp drop-off in the ability to detect bit flips in the middle of the fraction section. Taken together, from Figures 6 and 7, we can see that there is a sweet spot where both the overhead is low and the detection rate is high.

[4]We have not said much about how to choose the appropriate threshold. For now, we just use the value based on the analysis given in prior work [22].

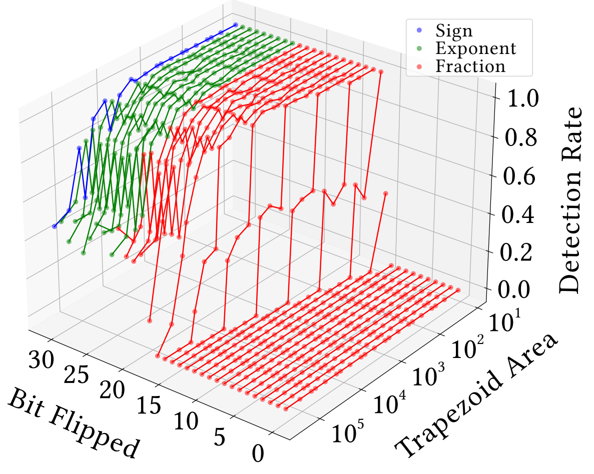

Figure 7. Error detection rate after 100 trials of randomly injected bit flips.

6 Related Work

With any fault tolerance scheme, there is typically a trade-off between error coverage and overhead. Duplication and related schemes like Triple Modular Redundancy [37], for example, have the highest error coverage but are often too costly. Cavelan and Ciorba showed how to use ABFT on stencil computations via interpolating checksum expressions over multiple time steps [22]. Their approach requires very careful and manual human analysis, however, and can not handle stencils with multiple time dependencies as with the example in Section 2.2. Their work is a rediscovery of an even older work on ABFT in the context of stencils by Roy-Chowdhury et. al. [21]. Our work can be viewed as a generalization of these two approaches with a wider range of applicability.

There is other prior work that employs fault tolerance analysis at compile time. Tavarageri et. al. proposed a compiler-assisted detection of memory errors [14]. Their approach works by ensuring that any data written to memory locations remain unchanged during any of its subsequent uses. This is achieved by augmenting the input program with checksums at compile time to keep track of the values *written into* and subsequently *read from* memory. Their approach has low overhead but has a lower error coverage than ABFT approaches. Errors that occur outside of memory, in the floating point arithmetic units, for example, go undetected. Our analysis here is based on numerical properties of the computed data which means we can detect errors *anywhere* in the pipeline as long as they are not swallowed by floating-point round-off noise.

7 Open Questions and Future Work

Our approach relies on being able to identify regions in the stencil domain computed using convolutions. However, these weight expressions in the stencil's convolution kernel may not be constant at each point in space or time. Stencils of this form correspond to the class of *anisotropic* partial differential equations [23]. If the weights truly are unique at each point in space, then efficient interpolation of checksums across time steps is not possible. This is because the weights can not be factored out of the expression in Equation 4. To the best of our knowledge, there has been no prior work on how to carry out ABFT efficiently on such anisotropic stencil applications and this remains an open and interesting problem.

8 Conclusion

We have studied the use of ABFT for stencil computations and proposed a new technique to automatically augment the input program with checksums to detect the occurrence of silent transient errors. We show that low overhead code can be easily generated and our preliminary results illustrate that there is an interesting trade-off worth exploring further.

References

[1] A. Hwang, I. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, p. 111–122.

[2] S. Höeffgen, S. Metzger, and M. Steffens, "Investigating the effects of cosmic rays on space electronics," *Frontiers in Physics*, vol. 8, 2020.

[3] L. Juracy, M. Moreira, A. Amory, and F. Moraes, "A survey of aging monitors and reconfiguration techniques," *CoRR*, vol. abs/2007.07829, 2020.

[4] A. Jagirdar, R. Oliveira, and T. Chakraborty, "Efficient flip-flop designs for SET/SEU mitigation with tolerance to crosstalk induced signal delays," in *Proc. IEEE Workshop Silicon Errors Logic Syst. Effects.* Citeseer, 2007.

[5] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, 2010.

[6] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold voltage (NTV) design: Opportunities and challenges," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1153–1158.

[7] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," *CoRR*, vol. abs/2102.11245, 2021.

[8] G. Aupy, A. Benoit, A. Cavelan, M. Fasi, Y. Robert, H. Sun, and B. Uçar, *Coping with silent errors in HPC applications.* Cham: Springer International Publishing, 2017, pp. 269–292.

[9] C. D. Martino, Z. Kalbarczyk, R. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: the case of blue waters," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 610–621.

[10] C. D. Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer, "Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 hpc application runs," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp.

25–36.

[11] M. Gupta, D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. Tullsen, and R. Gupta, "Compiler techniques to reduce the synchronization overhead of GPU redundant multithreading," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017, pp. 1–6.

[12] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "Swift: software implemented fault tolerance," pp. 243–254, 2005.

[13] N. Aggarwal, P. Ranganathan, N. Jouppi, and J. Smith, "Configurable isolation: Building high availability systems with commodity multicore processors," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, p. 470–481, 2007.

[14] S. Tavarageri, S. Krishnamoorthy, and P. Sadayappan, "Compiler-assisted detection of transient memory errors," ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 204–215.

[15] H. Kuang-Hua and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[16] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, p. 225–234.

[17] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 4, pp. 410–416, 2009.

[18] W. Sying-Jyan and N. Jha, "Algorithm-based fault tolerance for FFT networks," *IEEE Transactions on Computers*, vol. 43, no. 7, pp. 849–854, 1994.

[19] K. Zhao and et. al., "FT-CNN: Algorithm-based fault tolerance for convolutional neural networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1677–1689, 2021.

[20] T. Marty, T. Yuki, and S. Derrien, "Safe overclocking for CNN accelerators through algorithm-level error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4777–4790, 2020.

[21] A. Roy-Chowdhury, N. Bellas, and P. Banerjee, "Algorithm-based error-detection schemes for iterative solution of partial differential equations," *IEEE Transactions on Computers*, vol. 45, no. 4, pp. 394–407, 1996.

[22] A. Cavelan and F. Ciorba, "Algorithm-based fault tolerance for parallel stencil computations," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019.

[23] M. Patra and M. Karttunen, "Stencils with isotropic discretization error for differential operators," *Numerical Methods for Partial Differential Equations*, vol. 22, no. 4, pp. 936–953, 2006.

[24] O. Ovcharenko and V. Kazei, "Simple fdtd wave propagation in MATLAB," 2018. [Online]. Available: https://ovcharenkoo.com/WaveProp_in_MATLAB/

[25] S. Rajopadhye, S. Purushothaman, and R. Fujimoto, "On synthesizing systolic arrays from recurrence equations with linear dependencies," in *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*. New Delhi, India: Springer Verlag, LNCS 241, 1986, pp. 488–503.

[26] S. V. Rajopadhye, "Synthesizing systolic arrays with control signals from recurrence equations," *Distributed Computing*, vol. 3, pp. 88–105, 1989.

[27] P. Quinton and V. V. Dongen, "The mapping of linear recurrence equations on regular arrays," *Journal of VLSI Signal Processing*, vol. 1, no. 2, pp. 95–113, 1989.

[28] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[29] P. Feautrier, "Some efficient solutions to the affine scheduling problem. Part II. multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, 1992.

[30] P. Feautrier, "Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time," *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–347, 1992.

[31] C. Mauras, "Alpha: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones," Ph.D. dissertation, University of Rennes 1, 1989.

[32] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "Alphaz: A system for design space exploration in the polyhedral model," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2012, pp. 17–31.

[33] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, pp. 23–53, 1991.

[34] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Mathematical Software – ICMS 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–303.

[35] S. Rao, "1D finite difference time domain simulation (fdtd) with perfectly matched layer (pml)," 2022. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/53433-1d-finite-difference-time-domain-simulation-fdtd-with-perfectly-matched-layer-pml

[36] Gautam and S. Rajopadhye, "Simplifying reductions," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 30–41.

[37] R. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

Received 24 November 2022; revised 19 December 2022; accepted 12 January 2023