FPDetect: Efficient Reasoning About Stencil Programs Using Selective Direct Evaluation

ARNAB DAS, University of Utah
SRIRAM KRISHNAMOORTHY, Pacific Northwest National Laboratory
IAN BRIGGS, University of Utah
GANESH GOPALAKRISHNAN, University of Utah
RAMAKRISHNA TIPIREDDY, Pacific Northwest National Laboratory

We present FPDetect, a low-overhead approach for detecting logical errors and soft errors affecting stencil computations without generating false positives. We develop an offline analysis that tightly estimates the number of floating-point bits preserved across stencil applications. This estimate rigorously bounds the values expected in the data space of the computation. Violations of this bound can be attributed with certainty to errors. FPDetect helps synthesize error detectors customized for user-specified levels of accuracy and coverage. FPDetect also enables overhead reduction techniques based on deploying these detectors coarsely in space and time. Experimental evaluations demonstrate the practicality of our approach.

CCS Concepts: • **Hardware** → **Error detection and error correction**; • **Computer systems organization** → *Reliability*; • **Software and its engineering** → *Software verification*; • **Mathematics of computing** → **Numerical analysis**;

Additional Key Words and Phrases: Soft error detection, floating point round-off error, stencil computations, affine analysis, interval analysis, silent data corruption, software bug detection

ACM Reference format:
Arnab Das, Sriram Krishnamoorthy, Ian Briggs, Ganesh Gopalakrishnan, and Ramakrishna Tipireddy. 2020. FPDetect: Efficient Reasoning About Stencil Programs Using Selective Direct Evaluation. *ACM Trans. Archit. Code Optim.* 17, 3, Article 19 (August 2020), 27 pages.
https://doi.org/10.1145/3402451

19

1 INTRODUCTION

Approaching the end of Moore's law, chip manufacturers are seeking smaller lithographies, higher densities often achieved by emerging 3D stacking methods, newer memory/interconnect technologies, and increasing use of GPUs and other accelerators. These trends are increasing the likelihood

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award No. 66905. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract No. DE-AC05-76RL01830. This work is also supported by NSF CCF Grants No. 1704715, No. 1817073, and No. 1918497.
Authors' addresses: A. Das, I. Briggs, and G. Gopalakrishnan, University of Utah, 50 Central Campus Drive, Salt Lake City, UT, 84112; emails: arnabd@cs.utah.edu, ianbriggsutah@gmail.com, ganesh@cs.utah.edu; S. Krishnamoorthy and R. Tipireddy, Pacific Northwest National Laboratory, P.O. Box 999, MSIN J4-30, Richland, WA, 99352; emails: {sriram, Ramakrishna.Tipireddy}@pnnl.gov.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2020 Copyright held by the owner/author(s).
1544-3566/2020/08-ART19
https://doi.org/10.1145/3402451

of soft errors [4, 37, 41, 43] already noticed to be high in GPU-based systems [50]. The increased pressure toward specialization [26] may reduce the economies of scale achieved through general purpose parts and put cost-reduction pressures on verification methods, which already are stressed [29]. This can increase residual logical bugs in chips. The increasing complexity of compiler optimizations will also increase the likelihood of introducing logical bugs [3]. Software-level error detectors can serve as uniform and application-aware ways of trapping both soft errors [38] and logical bugs [3], and these detectors are needed more than ever before.

Unfortunately, system resilience research has seldom been transitioned into practice. Current resilience solutions targeting soft errors do not come with rigorous guarantees, cause a slowdown, and also generate false positives. Their possible additional benefits—such as flagging logical errors—have not been demonstrated either.

Our main contribution in this article is to demonstrate that for a narrow class of applications—specifically stencil-based—we can indeed develop solutions that benefit *both* logical error protection and soft error protection, owing to our solutions having two key attributes: (1) no false positives and (2) acceptable overheads. With this combination, we strongly believe that resilience solutions will be welcomed more readily, at least for their immediate impact on logical bugs.

Stencils belong to an important class of iterative solvers operating on dense D-dimensional arrays modeling partial differential equations (PDEs) with applications belonging to fluid dynamics, cosmology, combustion, and so on.[1] They are part of the essential building blocks for solving larger more composite problems involving multiple PDEs.

The ability to write good assertions that capture a stencil computation's evolving profile of data is tricky and error-prone. Existing work on system resilience that address data integrity include time-series data analysis methods [17] and data outlier detection methods constructed using machine learning (e.g., References [45, 47]). Unfortunately, these approaches have high overheads, and can only help loosely characterize the expected data value ranges. They additionally bring in training bias into the models constructed. Given that they both overestimate and underestimate the data ranges, detectors based on them generate both false positives and false negatives. With soft error detection, false positives are virtually unacceptable, given that the natural soft error rates themselves are quite low (false positive needlessly engage checkpoint/recovery systems).

FPDetect is the first approach that takes the novel approach of basing data-space protection on *accurate floating-point round-off error estimation.* Like any calculation carried out by numerical code, stencil-based calculations are also subject to floating-point rounding error. However, given that stencils are more structured, we show that one can develop a rigorous floating-point round-off error estimation approach for them, guaranteeing a certain number of mantissa bits after every stencil application. Consequently, in the FPDetect approach, if "round-off" appears suddenly exaggerated, we can attribute it reliably—without any false positives—to a logical bug or a soft error.

Contemporary floating-point error estimation approaches using interval analysis [6, 32] tend to give excessively conservative estimates (due to loss of correlation) of round-off error—especially for large programs that are iterated over time. These conservative estimates with large error bounds implies guarantees for only the higher order mantissa bits, i.e., only those errors that cause very high magnitude changes can be trapped using them.[2] FPTaylor [11] builds symbolic Taylor forms for the error expressions resulting in tight guarantees but do not scale to be usable beyond a few hundred operators. FPDetect, however, provide tighter guarantees on the error for large stencil programs, thus helping protect more mantissa bits and also allows a *tunable* approach to trading off overhead against detection precision, as we show in this article.

[1] While our approach applies to sparse stencils, we focus on stencils operating on dense matrices.
[2] That is, should these methods be used to synthesize error detectors, which has not been done.

Classical approaches like dual modular redundancy (DMR) can rigorously detect silent data corruptions by utilizing a duplicated execution thread and checking for result matches. This unfortunately can double the overall computation time. Clover [33] exercises a selective instruction duplication strategy to bring down the SDC detection cost to around 26% using tail-DMR. However, this approach is meant only for soft errors; it cannot be used to trap compiler bugs, because they impact *both* the execution threads.

In this article, we show that by focusing on a narrow (but important) class of applications—in our case stencils—we can arrive at a *unified solution* for detecting compiler bugs and soft errors while also offering rigorous guarantees. We offer our tool FPDetect, a *software-based error detector*, whose core approach is based on rigorous floating-point round-off error analysis for an iterated application of the stencil across T computational steps. To the best of our knowledge, this is the first work that encompasses two important correctness checks around the single central idea of offline round-off analysis.

Roadmap: Section 2 provides an overview of FPDetect, including error analysis, and optimizations. Section 3 is an overview of how FPDetect helps detect software bugs and soft errors. To reduce overheads, we establish a crucial result that if we leapfrog stencil applications and the detectors T steps ahead do not detect a soft error, then we can establish a *certified baseline* that allows earlier time steps to be forgotten. This allows FPDetect detectors to be deployed sparsely in space and time (e.g., once per 64 time steps) (Sections 4 and 5). We perform offline detector synthesis to create lookup tables, and instantiate specific detectors just before execution (Section 6). Section 7 evaluates software bug detection and resilience via fault injection. Related work (Section 8) and conclusions (Section 9) follow.

2 OVERVIEW

Floating-point error analysis: We provide only a brief overview (see References [24, 36] for details). A floating point number system, $\mathbb{F}$, in radix, β, is a subset of the set of real numbers, and can be expressed as a 5 tuple (β, s, m, e, p). We use binary ($\beta = 2$) double precision with number of *precision bits*, $p = 53$. $s \in \{-1, 1\}$ is the sign bit, e is the exponent in the range $-1{,}022 \leq e \leq 1{,}023$, and m is the mantissa or the significand, and represents the magnitude $s \cdot m \cdot 2^e$. If $x \in \mathbb{R}$, then $x_f \in \mathbb{F}$ denotes an element in $\mathbb{F}$ closest to x obtained by applying the rounding operator $(\circ)$ to x. We use the bound consistent for all IEE754 [1] rounding modes.

In floating point arithmetic, the *absolute error* is the magnitude difference between the values yielded by computations done in the space of real numbers ("true answer") and those done in floating point. The relative error is the ratio of the absolute error and the true answer. Every $x \in \mathbb{R}$ lying in the range of $\mathbb{F}$ can be approximated by an element $x_f \in \mathbb{F}$ with a relative error no larger than the *unit round-off* $\boldsymbol{u} = 0.5 \times \beta^{1-p}$. Here, β^{1-p} corresponds to the *unit of least precision(ulp)* for exponent value of 1. We use μ to denote ulp(1), such that $\mu = 2\boldsymbol{u}$. In our case $\boldsymbol{u} = 2^{-53}$, and hence $\mu = 2^{-52}$. Hence, for all rounding modes$(\circ)$, $\circ(x) = x(1+\delta)$, $|\delta| \leq \boldsymbol{u} = \mu/2$. Given two exactly represented floating point numbers x_f and y_f, arithmetic operators $\diamond \in \{+, -, \times\}$ have the following guarantees across any rounding modes:

$$x_f \diamond_f y_f = (x_f \diamond y_f)(1+\delta) = (x_f \diamond y_f) + (x_f \diamond y_f)\delta, \quad |\delta| \leq 2\boldsymbol{u} = \mu. \tag{1}$$

In our work, we employ affine arithmetic [16] for error estimation. In an affine representation, each input variable $\hat{x}$ is represented by $\hat{x} = x_0 + \sum_{i=1}^{n} x_i \epsilon_i$, with x_0 denoting the central value of the corresponding interval, and coefficients x_i being finite floating point numbers. The ϵ_i are *formal noise variables*, which are unknown until concretized but assumed to lie in the interval $[-1, +1]$.

Fig. 1. Simplified 1D stencil over six time steps.

Fig. 2. Illustration of path dominance.

Fig. 3. Essential (E_w) and protected widths (P_w).

Table 1. Evolution of Coefficient Values as We Unroll the Stencil

Set Number	Coefficients at Offsets						
offset index →	−3	−2	−1	0	1	2	3
Set_1	0	0	0.25	0.5	0.25	0	0
Set_2	0	0.0625	0.25	0.375	0.25	0.0625	0
Set_3	0.015625	0.09375	0.234375	0.3125	0.234375	0.09375	0.015625

The x_i model the noise coefficients such that the round-off error in Equation (1) can be expressed as $x_i = (x_f \diamond y_f) \times 2\mu$, with a noise variable $\epsilon_i \in [-1, 1]$.

We define two operators, σ and γ, that allows us to retrieve the associated noise variable and coefficient information for each affine variable.

- σ: Defines the mapping from an affine variable to its set of noise variables. Thus, $\sigma(\hat{x}) = \{\epsilon_i\}_{i=1}^{n}$.
- γ: Defines the mapping from a 2-tuple of *(affine variable, noise variable)* to its corresponding noise coefficient. Thus, $\gamma(\hat{x}, \epsilon_i)$ return x_i if $\epsilon_i \in \sigma(\hat{x})$ else return 0.

Using these operators, at every operator site, a fresh noise variable is introduced and their collective impact in the forward path is tracked using affine analysis. For codes implemented with the round to nearest model (as in most cases), one can configure $\mathbf{u} = \mu$, to obtain stricter bounds.[3]

Illustrative example: We now illustrate the concepts and practical details behind FPDetect using the simplified 1D stencil of three inputs leapfrogged (iterated) over six time steps (Figure 1). In this example, the stencil coefficients are 0.25, 0.5, 0.25 (say, modeling discretized heat flow) working on problem array A. The value $A[x, t+1]$ can be calculated to be $0.25 * A[x-1, t] + 0.5 * A[x, t] + 0.25 * A[x+1, t]$. We call this approach of obtaining the values at the next time step the *iterated* evaluation scheme. One can also imagine obtaining the value at $A[x, t+6]$ through *direct* evaluation that leapfrogs 6 steps, provided one calculates the *effective path coefficients*. Table 1 illustrates how these coefficients may be calculated by taking the sum of the product of the path weights (Set_2, for instance, is $0.375 = 0.25 * 0.25 + 0.25 * 0.25 + 0.5 * 0.5$).

Now, given a specific unroll (leapfrog distance) of T, we can then generate a similar table for that unroll, obtaining, for a k point stencil, $(k-1)T + 1$ effective coefficients. Then, given $(k-1)T + 1$ runtime values of these points, the direct evaluation scheme can obtain the value T steps ahead, and is the basis of realizing FPDetect detectors. We in fact implement this dot product of input values with the effective coefficients being in higher precision, and also employ Kahan's [30] summation

[3] A full derivation of error bound using affine arithmetic are presented in Reference [14] for the interested reader in **Appendix A**.

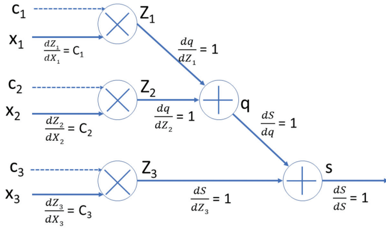

Fig. 4. Computational graph with highlighted derivatives for one step of a 1D three-point stencil.

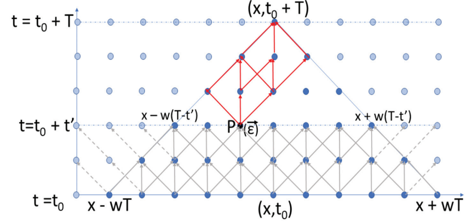

Fig. 5. Error propagation from point $(x - 1, t_0 + t')$ to $(x, t_0 + T)$ (1D stencil).

algorithm, as well as vectorization, thus minimizing the *relative error* of this answer.[4] Thus, we obtain an estimate for the relative error with minimal precision loss; call it R_d for *relative error under direct evaluation.*

While we know R_d, we cannot give any bounds on the number of mantissa bits preserved unless we also know the iterated evaluation's relative error, say R_s (s for "stencil" evaluation). A key contribution we make is to tightly estimate R_s analytically.

Iterated evaluation: There are many reconvergent computational paths to be taken into account during iterated evaluation for generating the output at $[x, t + 6]$ from the inputs at t. In such situations, unless we keep the errors on various re-convergent paths correlated, we will obtain uselessly exaggerated error bounds. Fortunately, affine arithmetic is well known for being able to handle error analysis in such situations. As an example, $(x - x)$ yields 0 under affine arithmetic (whereas interval arithmetic will yield an interval with the error in x doubled [36]).

FPDETECT comprises of an offline phase that includes static error analysis of the stencil combined with conservative profiling of the stencil (detailed in Section 6) for a set of protection goals as shown in Figure 6. Our error analysis method uses affine arithmetic building upon the works of References [6, 12, 27] but are applied to much larger expressions. As a simple illustration, consider a single step iteration of a one-dimensional (1D) three-point stencil with stencil coefficients $\{c_1, c_2, c_3\}$ centered around x_2 to evaluate x_2 for the next time step, denoted here as S in Figure 4. Thus, the computational graph evaluates $S = ((c_1 \times x_1) + (c_2 \times x_2)) + (c_3 \times x_3)$. If ϵ' is a noise variable belonging to $\sigma(x_1)$, then the error contribution's propagation at S will be

$$\gamma(x_1, \epsilon') \cdot K_{x_1} = \gamma(x_1, \epsilon') \cdot \left| \frac{dS}{dx_1} \right| = \gamma(x_1, \epsilon') \cdot \left| \frac{dS}{dS} \cdot \frac{dS}{dq} \cdot \frac{dq}{dz_1} \cdot \frac{dz_1}{dx_1} \right| = \gamma(x_1, \epsilon') \cdot c_1. \tag{2}$$

Thus, for all such $\epsilon' \in \sigma(x_1)$, the $\gamma(x_1, \epsilon')$ gets propagated by c_1 (and similarly for x_2, x_3, z_1, z_2, z_3, and q). Effectively, every incoming node or an internal compute node has a locally generated error term, and a propagation factor that propagates this error to the output node.

We can iteratively evaluate the stencil and arrive at this output value by (conceptually) picking all intermediate points, such as P that is highlighted in Figure 5, and first finding the error flowing into P. Then find out how this error is *modified* by all the path coefficients from P to the output. Do this for each point in the iteration space that iteratively contribute to the output at the detector location. Accumulating these error terms and normalizing them gives us the relative error R_s under iterated evaluation. While iterated evaluation mimics the stencil evaluation itself, *we are conservatively estimating this error during the offline phase* of FPDETECT. Our error analysis suitably combines the error for all such conceptual points P using affine arithmetic.

[4] We also employ vectorization to reduce overheads in the FPDETECT detector.

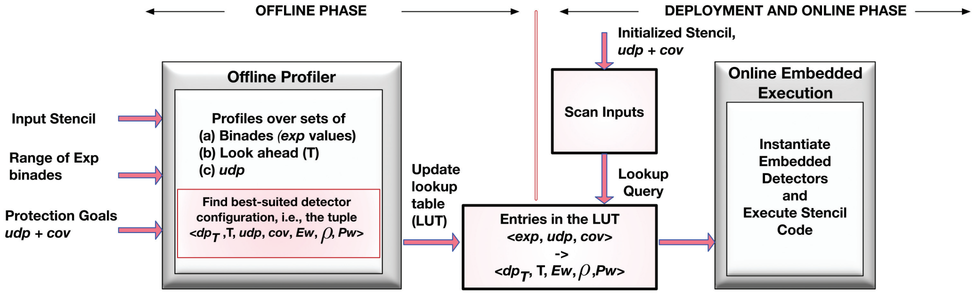

Fig. 6. Overview of FPDetect.

Detector precision (dp_T): We now can define a precision threshold to be checked for correctness at the detector locations.[5] Called *detector precision* dp_T (where the subscript T indicates it is a function of the number of lookahead Tsteps), it is a conservative bound on bits preserved, obtained by removing the maximum of the uncertainties between the direct and iterated evaluation.

Given that we minimize R_d, the key factor impacting the number of bits preserved will be the main stencil's iterated evaluations. Soft errors or logical bugs that throw the actual error beyond the dp_T limit can then be trapped with the guarantee of *no false positives*. Furthermore, all events whose impact falls within the guaranteed precision will be trapped. The only case of omission (which is outside of our model) is when two faults simultaneously land within the scope of two adjacent detectors, and both these fault-flows cancel out before reaching their respective detectors.

Offline/Online split: Our approach consists of an offline phase and an online one (Figure 6). The user presents the uninitialized input stencil and the range of intervals (in binades—meaning exponent spans) expected for each input. In the offline phase, we explore a rich search space. It is composed of the exponent binade ranges and protection goals defined by required precision accuracy and protection coverage. The goal is to obtain a best fit detector configuration for a pair of (*precision*, *coverage*) for each exponent binade and populate a look-up table for each application. Each application is identified by its unique coefficient set. Note that any change of coefficients for a given stencil results in re-synthesis as a new application, creating a new entry in the lookup table. In the online phase, we scan the input to obtain an interval that maps to a canonical exponent binade range. This allows FPDetect to query the LUT for any unseen data at runtime with user defined precision and coverage goals to obtain a best fit detector configuration. The worst case binade difference is extracted only at the beginning for the initialized input set in our experiments. Our rigorous guarantees are with respect to these binade differences (a fact that can be optionally revalidated at runtime).

Protected width: Our runtime goal is to not deploy the detectors at every time step, but only at every ρ time steps (Figure 3, where x-axis corresponds to the grid points in the stencil evaluation) by certifying the correctness of points within the ρ detection latency. Moreover, our goal is also to spatially distribute the detectors sparsely: the $\rho \times P_w$ boxes shown in this figure capture this goal, and these boxes are stacked horizontally. To arrive at these "stacking" optimizations, the user also provides the protection goals (*user defined precision* or *udp* defined in Section 4).

Optimizations, and overall algorithm: The actual stencil computation and detector deployment in a realistic problem are as presented in Table 2. In particular, FPDetect can handle multiple

[5] A full derivation of dp_T using affine arithmetic and error analysis details are presented in Reference [14] for the interested reader in **Appendix A**.

```
1  //D−dimensional array type
2  using MultiDimArray = ...;
3  //stencil routine on N multi−dimensional arrays
4  auto stencil(MultiDimArray A[N]) {
5   for(int t=0; ; t +=ρ) {
6    for(int tt=t; tt<t + ρ; tt++) { //in one time tile
7     if(converged) {
8      DETECT_ERRORS(); //<−−(Detector−2)
9      return ;
10    }
11    for(int x=0; x<N; x++) { //every array
12     for(i⃗ | l⃗ ≤ i⃗ ≤ h⃗) //every valid index
13      for(int y=0; y<N; y++) //every RHS array
14       for(s⃗ ∈ S_{x,y} ) //the stencil
15        TMP[i⃗] += c_{x,y}(s⃗) ∗ A[y][i⃗ + s⃗]; //update
16     for(i⃗ | l⃗ ≤ i⃗ ≤ h⃗) //every valid index
17      A[x][i⃗] = TMP[i⃗];
18    } //for 'x'
19   } // for 'tt'
20   DETECT_ERRORS(); //<−−(Detector−1)
21  } // for 't'
22 }
```

Fig. 7. Stencil computation pseudo-code. Compilers can transform lines 11–19 into any equivalent form. Detector-1 is used throughout the computation at time tile boundaries. Detector-2 is used at computation end.

Table 2. Stencil Computation Parameters

Parameter	Remark
$\vec{l} = \{l_1, \ldots, l_D\}$	lower bound of array region
$\vec{u} = \{u_1, \ldots, u_D\}$	upper bound of array region
N	number of arrays
ρ	time tile size
$S_{x,y} = \{\vec{s_1}, \ldots, \vec{s_n}\}$	Stencil offsets to update each point of array A_x from points in A_y
$c_{x,y}(\vec{s})$	stencil coefficients to update array A_x using array A_y

$\vec{l}$, $\vec{u}$, and $\vec{s_i}$ contain integers. $N, T \in \mathbb{Z}^+$. Implicitly, $\forall \vec{s} \notin S_{x,y} : c_{x,y}(\vec{s}) = 0$. $\forall \vec{s}, x, y : c_{x,y}(\vec{s}) \in [-1, 1]$.

computational arrays that iteratively update each other over time. With respect to the parameters specified in Table 2, FPDetect automatically instantiates and configures the requisite detectors. The pseudo-code in Figure 7 also presents the automatically inserted DETECT_ERRORS() calls corresponding to where the error is detected.

Figure 7 presents the pseudo-code for a stencil program operating on dense D-dimensional arrays. It involves updating the "neighborhood" of each point of every array in a given region, denoted by $\vec{l}$ and $\vec{u}$, with scaled values from other arrays. The neighborhood used to update array A_x from array A_y is defined by the stencil $S(x, y)$. At each time step t, every valid array index in each array x (i.e., bounded by $\vec{l}_x$ and $\vec{u}_x$) can be updated from the array locations in the stencil neighborhood of all other arrays y using the coefficient function $c_{(x,y)}$. A distinct stencil coefficient function can be defined for each pair of arrays in an update operation. A temporary array TMP is employed to handle write-after-read dependencies in the array being updated. The DETECT_ERRORS() instantiates the required detectors for the corresponding time tile. In the case of program converging and exiting before finishing the entire loop of the time tile, there is an extra set of trailing detectors instantiated by DETECT_ERRORS at the loop exit for protection of the end segment of the computation.

As Figure 2 shows, we can determine a collection of inputs such as $[x-4, t]$ that can be removed from consideration, *because they have no effect within* dp_T *bits of precision*. This calculation is described in Section 4. Our hope to leapfrog by T steps will not be realized unless we can "seal off" past computational points from future consideration. In Section 5, provide the analysis underlying this strategy. In our example stencil, it will allow us to shift the baseline by ρ steps as illustrated

in Figure 3. The new baseline is obtained at ρ steps in the future while T is the leapfrogging step. In fact, the following is an invariant for any detector instantiation: $\forall(\rho, T) : \rho \leq T$.

Key highlights of our approach include: (1) rigorous floating-point precision analysis made possible by exploiting the structured nature of our computations, (2) being able to define concepts such as essential (E_w) and protected (P_w) widths that help us reduce the amount of computation necessary to achieve a certain level of protection, and (3) cost analysis for optimal detector deployment (explained in Section 5) for a given amount of protection.

3 APPLICATIONS

FPDetect helps trap software-level bugs that may be introduced during compiler transformations and also soft-errors within thresholded precision limit.

3.1 Software Bug Detection

There has been a renewed interest in the generation of optimized programs that exploit CPU and GPU architectures, instruction sets, and customized hardware. Loop optimization frameworks (Pluto [7] that performs polyhedral transformations and Pochoir [48] that generates efficient implementations of stencil computations from high-level specification) are two examples. The generated code must be checked for bugs early in the toolchain. Building fully verified toolchains is non-trivial. FPDetect can be used as part of the testing toolchain for these tools as they are developed. Thus, FPDetect can enable faulty compilers being identified and corrected before being used in production environments. Specifically, compiler transformed programs can harbor bugs, as shown in the work on Polycheck [3]. Prior efforts have studied the design of approaches to verify or check the correctness of code generated by such optimizers [3, 40, 52]. These approaches detect bugs that impact the changes in control and data flow, with limited support for semantic transformations. Unlike the control or data-flow-based approaches, FPDetect is not limited by the nature of the transformations employed and carefully captures the semantics of the stencil operations. A drawback is that the analysis is imprecise in a different fashion than control or data flow analysis that are not affected by the floating point round-off errors.

We show that, with respect to the benchmarks chosen, commonly considered loop transformation bugs can be trapped with around 2% overhead, allowing tested by unverified codes to be shipped to third-parties who may keep the detectors turned on till gaining sufficient trust.

3.2 Soft-error Detection

FPDetect is intrinsically designed and optimized to help protect applications by detecting SDCs with small enough detection latency. We consider a single event error model wherein a single error at some time t *transiently* corrupts one or more values of one or more participating arrays in the computation resulting in a soft-error. These errors are non-permanent (transient) in nature and cannot be replicated making them extremely difficult to detect and isolate.

The soft error detectors generated using the FPDetect approach are, by construction, devoid of any modeling bias unlike those are generated through machine learning [45, 47] or time-series data analysis methods [17], which tend to reflect modeling bias. FPDetect's detectors faithfully represent the evaluation of the stencil by sampling the actual data space sparsely. It computes the output generated by a stencil application by faithfully accounting for the precision contribution of each point in the detector's dependence cone (as noted in Section 2). FPDetect enables the user with a parametric knob called the **user-defined precision (udp)**, with which they can define the required accuracy at the program output. Given a *udp* value of, say, 20, and a detector designed for T-step direct evaluation with an error bound of dp_T, FPDetect finds the *minimal set of points which contribute at least 20 bits of precision to retain* dp_T *bits of precision at the detector.*

We actually develop two detection strategies in the context of FPDetect. The first employs one direct evaluation and the other employs two direct evaluations. The latter is necessary given that the last leapfrogging step may not finish at a multiple of T steps; in those cases, the second direct evaluation provides the effect of a trailing detector at the end of the stencil computation.

Single-direct detector: At time t_0, direct evaluation is used to compute the estimated value of a point $A_x[t_0 + T][\vec{i}]$, T time steps ahead. When the iterative stencil computation reaches time $t_0 + T$ and evaluates this point, it is compared with the previously estimated value. If the two values differ more than can be ascribed to floating-point round-off error, then it indicates an error impacting either the direct evaluation or in some part of the iterative evaluation leading to that point, triggering a soft-error detection event. This strategy involves a single additional dot-product. The error is detected after T time steps, requiring that the computation executes at least till time step $t_0 + T$.

Double-direct detector: Values at two time steps t_0 and t_0+t' can be used to estimate a value at a future time t_0+T (T>t'). This involves two direct evaluations, one each at time steps t_0 and t_0+t', whose estimates can be compared in the latter of the two time steps. This allows flexible detection and does not require that the iterative computation reach time $t_0 + T$. A mismatch between the two estimated values (beyond the bound on floating-point round-off error) indicates an error in the computation between time steps t_0 and $t_0 + t'$. This detection strategy requires two dot products, potentially doubling the overhead. We minimize the overall overhead by using the single-direct detector as much as possible, and use the double-direct detector only at the expected end of the computation (typically determined by monitoring convergence).

A crucial efficiency consideration built into FPDetect is that it detects soft errors by placing detectors at time-tile boundaries (line 20 of Figure 7). This allows a programmer or an optimizer to transform the statements within each time tile into any semantically equivalent form. For example, a polyhedral optimizer (e.g., Pluto [7] or PolyOpt [51]) can optimize this stencil *without interference from the detector.*

4 OPTIMIZED DETECTOR SYNTHESIS

At a high level, FPDetect operates by comparing the iterative and direct evaluations within dp_T bits of precision. However, during the time interval t=t_0 to t=$t_0 + T$, not all points in the detector's dependence cone contribute to dp_T bits of precision; this is because of the round-off effects of using (finite-precision) floating-point arithmetic. This allows us to selectively carve out points from the input space. Two key concepts[6] introduced in this section—namely, that of **Essential Width** (E_w) and **Protected Width** (P_w)—helps define the amount of computation that can be carved out while still offering our guarantees. In Section 5, we describe an optimized detector synthesis scheme based on these concepts.

FPDetect works by synthesizing detectors for the expected ranges of binades that the computation begins with, and is also assumed to be present at every certified baseline. A T-step detector's support comprises of $(2 * T * \vec{w} + 1)$ points centered around $\vec{x}$, that is the points in $X_D = [\vec{x} - \vec{w}T, \vec{x} + \vec{w}T]$, where $\vec{w}$ represents the footprint of one iteration of the stencil expressed as a vector to include higher dimensional spaces. Each point x_i in the X_D belongs to the tightest encompassing interval I. Upon multiplying each x_i by their respective T-step coefficients, c_i, generates separate intervals for each individual product term denoted as

$$[\underline{y_i}, \overline{y_i}] = [\min(c_i\underline{x_i}, c_i\overline{x_i}), \max(c_i\underline{x_i}, c_i\overline{x_i})].$$

This results in a new set of N points, $Y = [y_1, y_2, \ldots, y_N]$, where $N = (2T\vec{w} + 1)$.

[6]These are presented in more detail as **"Precision Driven Optimizations"** in **Appendix B** of Reference [14].

Define summation $S_Y = \sum_{j=1}^{N}[\underline{y_j}, \overline{y_j}]$, which falls in the interval $[\underline{S_Y}, \overline{S_Y}]$. If $[\underline{y_i}, \overline{y_i}]$ does not influence S_Y in the scope of dp_T, then point y_i is a candidate for removal. We do this pointwise analysis for each y_i. For this, we define $S_{Y\setminus i} = \sum_{j=1,\neq i}^{N}[\underline{y_j}, \overline{y_j}]$ and its bounds as $[\underline{S_{Y\setminus i}}, \overline{S_{Y\setminus i}}]$. Now define $d_{\min}(y_i)$ as the distance between the exponent of the lower bound of $S_{Y\setminus i}$ and the exponent of the upper bound for y_i:

$$d_{\min}(y_i) = \max(0, exp(\underline{S_{Y\setminus i}}) - exp(\overline{y_i})). \tag{3}$$

This equation and the next help check whether y_i's *maximal* contribution manages to affect the *minimal contribution* from all the remaining points (if not, then due to the exponent differences, the magnitude contributed by y_i gets "shifted out" during the process of normalizing the exponents while doing floating-point addition):

$$maxContrib_p(y_i, S_Y) = p - d_{\min}(y_i). \tag{4}$$

Equation (4) gives the maximal precision contribution of point y_i to the final sum S_Y when S_Y is computed up to p precision bits. A y_i becomes part of the exclusion set, if $maxContrb_p(y_i, S_Y) < 0$. Note that $y_i = (c_i \times x_i)$ already includes one multiplicative term adding an extra half ulp error to the analysis.

Now, for a detector placed at $A_u[\vec{x}, t_0 + T]$, we define **Essential Width** (E_w) as the width of the multidimensional rectangular region around $\vec{x}$ such that the direct evaluation over E_w is *sufficient* to guarantee dp_T precision at the detector.[7] Specifically, the region is defined by extents E_{wl} and E_{wr} such that $E_w = E_{wl} + E_{wr} + 1$. The caveat from Equation (4) is that all such excluded candidates could *collectively* affect the output within the precision limits; therefore, we must engage in an iterative process of considering subsets of points to carve out such that Equation (5) holds. In the general case, E_w is a vector and is written $\vec{E_w}$. Figure 3 illustrates E_w for evaluation of a 1D stencil over T-steps.

LEMMA 4.1. *In the absence of an error, direct evaluation over the essential width (E_w) ensures the correctness of the computation to at least dp_T bits of precision at the detector location.*

For all points in the region excluded from $E_w \forall \tilde{X} : \tilde{X} \subset ([\vec{x} \pm \vec{w}T] - \vec{E_w})$, the following holds:

$$\forall x_i \in \tilde{X} : maxContrib_{dp_T}\left(\sum(y_i = c_i \times x_i), S_Y\right) \leq 0. \tag{5}$$

One form of false positives involves detection of soft errors when no error affects any iteration point on which the detector depends (aka the detector's dependence cone). The preceding lemma guarantees that, despite the reduced evaluation cost, in the absence of errors affecting the dependence code, the direct evaluation is equivalent to the iterative evaluation within dp_T bits and no soft error notification (false alarms) is triggered.

Covering multiple points with a single detector: Protected Width (P_w): As depicted in Figure 5, each point inside the dependence cone of the detector has varying contribution depending on its effective path contribution. To guarantee that an error affecting a bit at an iteration point is detected by a detector protecting it, the minimum contribution of that bit must impact the dp bits at the detector. This will result in the values computed by iterative and direct evaluation being different, triggering a soft error notification.

To quantify the minimal possible influence a y_i has on the final sum S_Y, we define another distance metric $d_{\max}(y_i)$ as the distance between the exponent of the upper bound of $S_{Y\setminus i}$ and the

[7] The word "sufficient" is important, because we might, in general, have non-contiguous chunks of inputs that can be discarded. In FPDetect, we extend E_w to make it a contiguous region.

lower bound of y_i:

$$d_{\max}(y_i) = \max(0, exp(\overline{S_{Y\setminus i}}) - exp(\underline{y_i})). \tag{6}$$

Then, the minimal influence y_i has on the final sum when computed correct to dp_T bits of precision is evaluated as

$$minContrib_{dp_T}(y_i, S_Y) = dp_T - d_{\max}(y_i). \tag{7}$$

A detector's reach can then be configured to guarantee an user defined precision (*udp*) by carving out a set Y_{udp} from X such that

$$Y_{udp} = [y_i : minContrib_{dp_T}(y_i, S_Y) \geq udp, \quad y_i \in Y]. \tag{8}$$

The most significant *udp* bits of precision of each point inside Y_{udp} is then protected if S_Y is evaluated correctly to dp_T bits of precision. Equation (8) leads to our primary soft-error detection model that provides the guarantee that an error affecting within the most significant *udp* bits of points inside Y_{udp} is detected. In our stencil model, for a detector placed at $A_u[\vec{x}, t_0 + T]$, we define **Protected Width** ($P_w(T, \rho, udp)$) as the width of a multidimensional rectangular region centered around $\vec{x}$ such that with respect to a detector placed at time $t_0 + T$ and spatial position $\vec{x}$, for each $\vec{p} \in \vec{x} \pm P_w(T, \rho, udp)$ at time $t_0{+}\rho$, the above guarantee holds.

LEMMA 4.2. *An error affecting any of the MSB* udp *bits of a point inside the protected width is detected.*

For a point y_i in y, if y_i belongs to Y_{udp} and its minimal precision contribution is p_i at the final output, then $p_i \geq udp$. If $y_i \in [\underline{y_i}, \overline{y_i}]$, then an error, err_{y_i} affecting within the MSB *udp* bits will be bounded by $err_{y_i} \geq 2^{exp(\underline{y_i})-p_i+1}$. Since we are matching dp_T bits at the detector, hence the threshold of detectable error is bounded by $2^{exp(\overline{S_Y})-dp+1}$.

For our guarantee to hold, generated error ≥ error threshold. This means $2^{exp(\underline{y_i})-p_i+1} \geq 2^{exp(\overline{S_Y})-dp_T+1}$. Taking logarithms and simplifying the above terms leads to the following relation that must hold true for all points inside the P_w region: $exp(\overline{S_Y}) - exp(\underline{y_i}) \leq dp_T - p_i$ and $udp \leq p_i \leq dp_T - (exp(\overline{S_Y}) - exp(\underline{y_i}))$.

In determining the protected region, where a detector is used to protect iteration points in multiple time steps, we choose ρ such that the protected width chosen at ρ is also valid for all time points $t_0 \leq t \leq t_0 + \rho$:

$$\forall\, 0 \leq t \leq \rho : P_w(T, \rho, udp) \leq P_w(T, t, udp). \tag{9}$$

This enclosed region forms the protected region for the given detector location. Figure 3 illustrates the protected region in terms of P_w and ρ steps for a given *udp*.

Probabilistic detection: While the above strategy guarantees detection, the stencil structure and user-specified *udp* requirements can severely constrain the protection region, incurring high detection overheads. To further trade-off overheads for detection capability, we consider a detector that only provides the detection guarantee on only a fraction of the points in the protected region associated with a detector. Given a specified coverage requirement *cov*, we choose the protected width P_w such that the fraction of points with guaranteed protection is at least *cov*. An error impacting the MSB *udp* bits of an iteration point is guaranteed to detected by its enclosing detector, if it is among the points for which the detection is guaranteed. Alternatively, such an error is detected with probability of at least *cov*.

5 OPTIMIZED DETECTOR PLACEMENT

Dampening errors: We consider a single event fault model wherein a single fault occurring at time t can transiently corrupt one or more participating arrays. Based on properties discussed in the preceding section, an error at time t escaping detection by the nearest set of detectors must have impacted less significant bits. Suppose the coefficient set $C = \{c_i\}_{i=1}^{n}$ denotes the set of forward contributions of a point in the stencil. Then Lemmas 5.1 and 5.2 given below hold under the following assumptions:

- $\forall i, |c_i| \leq 1$: This implies all forwards contributions move the error magnitude toward the least significant bits
- $\sum_i c_i \leq 1$: The collective error spread from an affected point to all neighboring points in the forward path is less than equal to itself
- There is no error cancellation within the scope of a detector. Multiple detectors may trigger a detection event as long as the triggering errors are localized to the scope of those detectors without any intervening error cancellations from its neighbors.[8]

The first two conditions are driven by stability criteria and smoothening impact of the stencil.

Lemma 5.1. *An error may never amplify in the forward path.*

Lemma 5.2. *Error dampening guarantees an error missing detection does not affect future computations within the required precision limits.*

Assuming dampening, detectors can be arranged to cover the iteration space as follows:

- *Certified baselines:* Error dampening allows us treat computations up to t that have passed error detection checks as effectively error-free. These checked points at t will be used as new *certified baselines* to perform direct evaluation of the next detector.
- *Horizontal detector placement:* If a set of detectors is placed horizontally, i.e., at the same time step t, P_w apart in all dimensions, then the points at time t, once they pass the detection checks, will constitute a new certified baseline. Thus, for a d-dimensional stencil of size $N=\{N_i\}_{i=1}^{d}$ along each dimension, if the detectors are placed $P_w=\{pw_i\}_{i=1}^{d}$ apart along each dimension, then the number of detectors required is $\prod_{i=1}^{d}(N_i/pw_i)$.
- *Vertical detector placement: Stacking.* Because the detector characteristics are influenced by the input range, at every baseline the input range needs to be computed. Alternatively, a bound on the input range can be predetermined based on the initial/boundary conditions and used for the entire computation. In both scenarios, the detectors can be vertically stacked to create new certified baselines as execution moves forward.

Illustration: Figure 8 illustrates the placement of detectors for the first two certified baselines for a 1D stencil. D_{01}, D_{02}, and so on, show the placement of the first set of detectors at time step T. These are computed from the array values at time step 0. $D_{11}, \ldots$, the second set of detectors at time $T + \rho$, computed from the certified baseline at time step ρ. The detectors are vertically spaced ρ time steps apart, and horizontally spaced P_w apart. The protected regions are marked as squares and labeled at the right top. For example, P_{01} is the region protected by detector D_{01}. The shaded region denotes the boundary region unprotected by the detectors. Of this region, the shaded gray region has no influence on the detectors protecting points in that time step (outside their dependence cones). Errors in this region are not detected. The remaining region has an influence on at least one of the

[8] Such cancellations, if they occur, do not affect the final output.

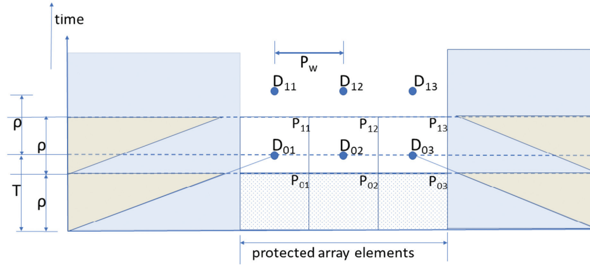

Fig. 8. Detector arrangement for a 1D stencil. Horizontal: array elements; vertical: time steps.

detectors. Errors in this region might be detected but are not part of guaranteed coverage.

$$\%cov = \prod_{i=1}^{d} \left(\frac{N_i - 2wT}{N_i} \right) * 100 \tag{10}$$

Boundaries: Although our method is applicable to the boundary points, the contributions from the boundary points need to computed differently from the interior, incurring additional overhead. For cases where time dependent boundary conditions and Neumann boundary conditions [2] exists, FPDetect lacks prior knowledge of how the boundary points are forced in the intermediate time tiles. For fixed Dirichlet boundary conditions, we can provide full coverage. To simplify the implementation, we ignore protection for the boundary regions in the generalized cases. Specifically, any computation point influenced by the boundary between certified baselines is not checked. Note that this reduces the maximum possible coverage to be below 100%, however, still providing an accurate bound on the guaranteed coverage per Equation (10) akin to probabilistic coverage that exemplifies the flexibility afforded by trade-off detection quality versus cost.

An error affecting such an unprotected region can spread to the rest of the computation space and result in erroneous output. This is akin to probabilistic coverage, with the probability of detection reduced by fraction of the total computation space left unprotected at the boundaries. Let N_i denote the problem size along i'th dimension in a d-dimensional problem. Equation (10) gives the percentage of covered region for detectors instantiated with $Tstep = T$ (w is the stencil width[9]).

Optimal detector configuration: For a given choice of detector, the essential width E_w is a function of the Tstep, the input exponent range, and the detector precision (*dp*). P_w additionally depends on *udp* and ρ. The cost function is evaluated as the relative cost with respect to the actual stencil evaluation. Section 6.1 details the cost function and the offline analysis used to construct optimal detector configurations.

6 OVERALL ALGORITHM

FPDetect performs an offline manuever to obtain a look-up table with optimal detectors configurations for the detectors. For each anticipated choice of *udp*, probability of detection, and anticipated input range, the offline analysis (Section 6.2) determines the detector configuration to be used during online execution. A detector configuration consists of: T, the distance at which the single-direct detector is evaluated; ρ, the number of iterations between certified baselines; E_w, the essential width to evaluate the detector; Detector coefficients for the support in the essential width; and P_w the spatial separation between detectors in the same time step. These parameters completely specify the detector configuration at runtime.

[9]The implementation uses the exact expression, accounting for stencil asymmetry and non-Cartesian shapes.

6.1 Offline Determination of Detector Configurations

Constraints on detector configuration: We use offline analysis to efficiently identify optimal detector configurations for use during online execution. To constrain the search space of possible configurations, we use an upper-bound on the $Tstep$ (say, $T_{\max} = 256$ in our evaluation) that one might use for the detector evaluation. Larger $Tstep$ values are useful and often minimize the detector operational cost. However, this requires larger coefficient sets, needed for the direct evaluation, to be stored in memory. For a d-dimensional stencil with N-arrays, the space overhead encountered to store the effective coefficient space is $O(N \cdot T_{\max}^{d+1})$. The range of *udp* values to be considered is bounded by the number of bits (1–53) and the maximum precision (dp) that can be preserved by both direct and iterative stencil evaluations informed by the round-off error analysis. We determine the detector configurations for a finite set of coverage choices between 0% and 100%, corresponding to the fraction of points in a detector's protected region that are protected to the desired *udp* bits. The actual coverage including the boundary is computed online.

Relativized input exponent ranges: Consider two exact floating point numbers represented in triplet form as $a_f = (s_a, m_a, e_a)$ and $b_f = (s_b, m_b, e_b)$, where the mantissa m_a and m_b are represented in $p = 53$ bits. For a floating point addition (subtraction), the mantissa of the number with the lower of the two exponents has to be shifted by $|e_a - e_b|$ to match their exponent values. Thus, the binary addition (subtraction) of the mantissa depends the relative distance between the exponents for shifting (not on actual exponents).

In stencils, update rules are modeled as weighted sums involving only addition (subtraction based on sign) and multiplication. Multiplication by exact scalars involves binary multiplication of the mantissa, followed by addition of the exponent terms, maintaining linearity in the exponents. For example, let the coefficients associated with a_f and b_f be $\alpha_1 = (s_{\alpha_1}, m_{\alpha_1}, e_{\alpha_1})$ and $\alpha_2 = (s_{\alpha_2}, m_{\alpha_2}, e_{\alpha_2})$, respectively. The product $\alpha_1 a_f$ will have an exponent of $(e_{\alpha_1} + e_a)$ and $\alpha_2 b_f$ will have $(e_{\alpha_2} + e_b)$. Thus, the relative distance of exponents between these two terms will be $|(e_{\alpha_1} - e_{\alpha_2}) + (e_a - e_b)|$, which depends essentially on the coefficient's exponents (that remain unchanged) and the relative operand exponent differences. This fact can be utilized to map different interval ranges to some canonical exponent range that models the maximum relative distance of points inside the interval. To do this, we scan the input to find the smallest data point (in magnitude) and the largest data point (in magnitude) and characterize that interval with the exponent difference between these two data points. For an input interval, if (m_a, e_a) represents the smallest value (in magnitude) and (m_b, e_b) is the largest value (in magnitude) seen in the interval, then factoring out e_a produces the following mapping: $[(m_a, e_a), (m_b, e_b)] \equiv (1, e_a)[(m_a, 0), (m_b, e_b - e_a)]$.

We can further factor out the corresponding mantissa and increment the mapped exponent interval width by 1 to have a larger bounding interval for the given input range. Since our analysis is conservative, bounds that hold for larger exponent ranges also hold for smaller exponent ranges:

$$[(m_a, e_a), (m_b, e_b)] \equiv (m_a, e_a)\left[(1, 0), \left(\frac{m_a}{m_b}, e_b - e_a\right)\right] \equiv [(1, 0), (1, e_b - e_a + 1)]. \tag{11}$$

The operation of a stencil on an input range can be *relativized* to a small number of canonical exponent ranges. This allows us to use the offline analysis performed on specific exponent ranges on different set of actual exponents that belong to the same range. In our experimental setup, we profiled exponent ranges form 0 through 20 ($exp_{\max} = 20$). The maximum *udp* is set as 40. Given the range of exponents and *udp* choices, the offline algorithm construct a lookup table that returns a 5-tuple (T, ρ, dp, P_w, E_w) corresponding to an optimal detector configuration for the given interval, required *udp*, and minimum detection probability.

Offline algorithm[10]: We consider all T and ρ such that $T \leq T_{\max}, \rho < T$. For a d-dimensional stencil, P_w is d-dimensional vector, corresponding to protected region of iteration points centered at the detector. E_w is represented by a rectangular region with a computed number of points along left and right of the detector along each direction. The cost function is derived as the ratio of the total detector overhead to the total cost of the stencil application evaluation. Post simplification of the cost function for a 5-tuple detector configuration $(exp, udp, T, \rho, Coeffs)$ is given as

$$Cost = \frac{1}{\rho} \prod_{i=1}^{d} \left(\frac{ew_i}{pw_i} \right) \tag{12}$$

derived fraction of overhead cost to total stencil compute cost. Even though the tuple elements exp, T, udp, and $Coeffs$ do not explicitly appear in the cost function, they implicitly influence the cost through E_w and P_w.

The algorithm takes as input parameters for which an offline profiles need to be determined: the maximum number of time steps (Tmax), the set of input ranges (as exponents in exp_set), set of *udp* values (as udp_set), and probabilistic coverage values (cov_set). The algorithm determines the configurations one input range choice at a time. Using floating-point round-off analysis, the maximum number of bits that will be preserved for each possible time step t is computed (as maxdp). For each t, the cost of evaluating each detector is computed as the product of the E_w dimensions to guarantee $maxdp[t]$ bits of precision of the direct evaluation t time steps away. Then, for each candidate protected region, the fraction of points with guaranteed coverage of b bits (where b is a candidate *udp* in the input parameter udp_set) is computed. If this fraction is greater than a desired coverage and if the associated cost is lower than that of any configuration seen thus far, then this configuration is chosen. After evaluating all feasible solutions, the algorithm returns the last chosen configurations.

The cost of the offline procedure is dominated by dimensionality of the space to be explored. While this exhaustive evaluation of every feasible configuration can be expensive, for the benchmarks considered (in the next section), each offline procedure completes within several minutes. Search-space exploration techniques might further lower this cost.

6.2 Online Detector-embedded Execution

We briefly discuss the algorithm[10] to obtain the detector configurations and embed the detectors within the original stencil code. During online deployment of FPDetect's detectors, the input values are scanned to compute the input exponent range to be handled in the floating-point space. To account for the unprotected boundary regions, FPDetect maps user's input coverage value to an *equivalent internal coverage value* (as in_cov) that corresponds to the detection probability on the grid excluding the boundaries. The adjustment is done with the guarantee that the *fraction of points covered due to (in_cov) is at least as many required by cov over the entire computation space.* If FPDetect fails to find an equivalent mapping, then it conveys an error message for unsupported coverage. The input range, user-specified *udp*, and modified detection probability (in_cov) are used to lookup the detector configuration.

After an initial evaluation of the detector and the stencil for ρ iterations, the execution of the stencil is split into two segments: iterations till the next detector evaluation (`eval_detector`) and iterations till the next detector check (`detector_check`). Thus, the stencil is executed with *interleaved detector evaluation and checking*. Note that the algorithm assumes ρ is greater than half the detector evaluation T-step. This scenario requires at most two "live," i.e., unchecked detectors

[10]The algorithmic listing is available in the supplementary material (Appendix C) of Reference [14].

Parabolic (heat):
$\frac{\partial u}{\partial t} = \nabla^2 u(x, y) + f$
$\alpha = 3, \zeta = 1.2$

[Dirichlet initial/boundary condition]
h1. $f = \zeta - 2 - 2\alpha, u = 1 + x^2 + \alpha y^2 + \zeta t$
h2. $f = 0$, $u = 1 + x^2 + \alpha y^2 + \zeta t$
h3. $f = 2\zeta - 2 - 2\alpha$, $u = 1 + x^2 + \alpha y^2 + \zeta t^2$

[Neumann initial/boundary condition] $f = 0$
h4. $u(x, y, t) = e^{-\pi^2 t/2} \sin \pi(\frac{x+y}{2})$
h5. $u(x, y, t) = 4 + e^{-\pi^2 t/2} \cos \pi(\frac{x+y}{2})$
h6. $u(x, y, t) = 2 + e^{-\pi^2 t/2} \left[\sin \pi(\frac{x+y}{2}) + \cos \pi(\frac{x+y}{2})\right]$

Poisson equation:
$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = \text{RHS}$

[Dirichlet initial/boundary condition]
p1. RHS = -6, $u = 1 + x^2 + y^2$
p2. RHS = $-6(2 + x + y)$, $u = 1 + x^3 + y^3$
p3. RHS = $-2 - 12y$, $u = 1 + x^2 + 2y^3$

[Deflection in a membrane]
$p_1 = 4e^{-5((x-0.6)^2+(y-0.6)^2)}$
$p_2 = 2e^{-5((x-0.3)^2+(y-0.3)^2)}$
$p_3 = 4e^{-5((x-0.3)^2+(y-0.6)^2)}$
p4. RHS = p_1, $u = 0$
p5.RHS = $p_1 + p_2$, $u = 0$
p6. RHS = $p_1 + p_2 + p_3$, $u = 0$

[Neumann initial/boundary condition]
p7. RHS = $10e^{-\frac{(x-0.5)^2+(y-0.5)^2}{0.02}}$, $\frac{\partial u}{\partial n}(x, y) = -\sin(5x)$
p8. RHS = 0, $\frac{\partial u}{\partial n}(x, y) = -\sin(5x)$
p9. RHS = $20e^{-\frac{(x-0.25)^2+(y-0.25)^2}{0.01}}$, $\frac{\partial u}{\partial n}(x, y) = -\sin(5x)$

Hyperbolic (Second-order wave):
$\frac{\partial^2 u}{\partial t^2}(x, y, t) = c^2 \nabla^2 u(x, y, t)$

[initial/boundary conditions: $c = 1$, $\frac{\partial u}{\partial t} = 0$]
w1. $u(x, y, t) = \cos(\sqrt{2}\pi t) \sin(\pi x) \sin(\pi y) + x^2 - y^2$
w2. $u(x, y, t) = \sin(\sqrt{2}\pi t) \cos(\pi x) \cos(\pi y) + x^2 - y^2$
w3. $u(x, y, t) = \sin(\sqrt{2}\pi t) \cos(\pi x) \cos(\pi y) + x^2 - y^2$

[initial and boundary conditions: $c = 0.7$, $\frac{\partial u}{\partial t} = 0$]
w4. $u(x, y, t) = 16 + 2\sin(\frac{\pi}{4}x)\sin(\frac{\pi}{4}y)\cos(\frac{\pi}{2}c^2 t)$
w5. $u(x, y, t) = 16 + \sin(\frac{\pi}{2}x)\sin(\frac{\pi}{2}x)\cos(\frac{\pi}{4}c^2 t)$
w6. $u(x, y, t) = 16 + 2\sin(\frac{\pi}{2}x)\cos(\frac{\pi}{4}x)\sin(\frac{\pi}{2}c^2 t)$

Hyperbolic (convection diffusion):
$\frac{\partial u}{\partial t} + \alpha \nabla u(x, y) = \zeta \nabla^2 u(x, y)$
initial/boundary condition from
$u(x, y, t) = \frac{1}{4t+1} e^{\frac{-(x-\alpha t-0.5)^2-(y-at-0.5)^2}{\zeta(4t+1)}}$

c1. $\alpha = 0.8, \zeta = 0.01$
c2. $\alpha = 0.4, \zeta = 0.4$
c3. $\alpha = 0.1, \zeta = 0.8$

Fig. 9. Benchmark PDEs (p1–p9, h1–h6, w1–w3, c1–c3) solved using the stencil finite-difference method. Initial and boundary conditions are derived from the equations provided. All stencils span the [0, 1] spatial domain and are run 4,000 time steps.

per spatial position in the iteration space, at any time. If not, then at each spatial position, multiple detectors need to be evaluated and retained until they are checked. When all iterations have been executed, the iterations past the last certified baseline need to be checked. These leftover iterations are checked using the trailing detection strategy.

7 EVALUATION

Benchmarks: We evaluate FPDetect on the stencil kernels shown in Figure 9. We choose a set of benchmark problems from elliptic, parabolic, and hyperbolic PDEs evaluated as stencils using the explicit finite difference method. Benchmark examples considered are Poisson equation with different combinations of Dirichlet and Neumann boundary conditions, heat equation with different initial and boundary conditions, and second-order wave and convection-diffusion equations. Our error analysis and optimization techniques are based on the input data interval and not the exact values. We test our hypothesis rigorously by exercising each benchmark equation with multiple initial and boundary conditions. In addition to the exponent and sign bits, fault-injection-based evaluation was conducted for a user-defined precision of 15 mantissa bits. This corresponds to a total protection of 27 most significant bits.

Experimental setup: All benchmarks were compiled using ICC 18.0.5 with `-qopenmp-O3` options and run on dual 14-core Intel Xeon CPU E5-2680v4 2.60 GHz CPUs system (total 28 cores) with 64 GB of RAM. Pluto optimized code was generated with "PLUTO version 0.11.4-350-g8debc44."

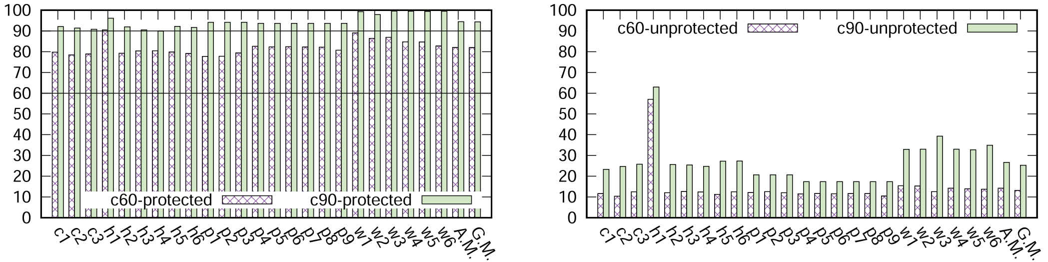

Fig. 10. Detection rates of single bit-flip error injections in bits protected (left) and unprotected (right) by the detector (A.M.: arithmetic mean; G.M.: geometric mean).

Detector coverage excluding boundaries: To reduce the complexity of the detection strategy, we did not protect statement instances impacted by the boundary between certified baselines. This limits the maximum coverage that can be guaranteed by our implementation. For all benchmarks and configurations evaluated, we observed this fraction to be greater than 90% of the computation space, demonstrating the simplification does not significantly limit detection ability.

Space overheads: At stencil runtime, FPDetect requires space to store:

- the set of all possible coefficients of interest,
- the direct evaluation configurations for specific input ranges and precision needs, and
- the result of the selective direct evaluation by each detector.

We bound the offline analysis with $T_{\max} = 256$ iterations and require the full coefficient set for T=0 to $T_{\max}$. This allows a runtime choice of time tile size between 1 and 256. This incurs a space overhead of 128 MB. While incurring a large memory footprint, only a fraction of this data—one rectangular portion the single-direct detector—is used during online execution and the accessed portions are reused for all detectors at a time step. In addition, the trailing detector accesses one rectangular portion determined by the number of iterations not executed by the last tile. This results in good cache behavior, limiting their impact on overall performance.

The offline configuration lookup table is built by profiling over 20 exponents, 40 *udp* and coverage values from 0 to 100% (in increments of 5%). Thus, the offline lookup table is indexed by a 3-tuple of (exp, udp, cov), with each lookup returning a 6-tuple configuration of $(T, \rho, dp, \vec{P_w}, \vec{E_{wl}}, \vec{E_{wr}})$, where $\vec{E_{wl}}$, $\vec{E_{wr}}$, and $\vec{P_w}$ are $d-$dimensional vectors for a $d-$dimensional stencil program. Here, $\vec{E_{wl}}$ and $\vec{E_{wr}}$ corresponds to the left and right extents of E_w.[11] FPDetect scans the data at runtime to determine the necessary exponent range. This is combined with the user-specified precision and coverage to index the lookup table and obtain the corresponding detector configuration. For a 2-dimensional problem, a key-value pair in the table occupies a space of 48 bytes. Over all the profiled configurations, the configuration lookup table requires a total space of ≈750 KB per benchmark.

In addition to the coefficients and the lookup table, we require one floating-point number per array at each detector location to store the value computed through selective direct evaluation. The maximum number of instantiated detectors comes to around 2,546K for the wave benchmarks. For most other benchmarks is around 100K for a problem size of 10K × 10K and $T_{\max} = 256$.

[11] $\vec{E_{wl}}$ and $\vec{E_{wr}}$ are equal in the case of symmetric stencils.

7.1 Software Bug Detection

We evaluate the effectiveness of FPDetect in detecting logical errors injected into the code generated for the stencil programs in Table 9 optimized by Pluto [8]. Specifically, we inject three classes of bugs injected in the evaluation of PolyCheck [3], a tool designed to check errors introduced by loop transformers: incorrect loop bounds, invalid array accesses, and invalid loop reorderings.

For each software bug introduced, we evaluated its impact using bitwise comparison of the result with that of the non-buggy version. Any mismatch is treated as a bug. Some bugs might not result in an erroneous result for the problem instances evaluated. FPDetect cannot aid in their detection.

The bugs introduced in the source potentially affect multiple runtime operations, making them easier to detect than soft errors. Exploiting this, we evaluate FPDetect 's effectiveness in detecting software bugs when deployed with a small *udp* value of 4. For each benchmark, the default detector precision (*dp*) determined for this *udp* is used in the evaluation.

Pluto-generated code: To generate optimized versions using Pluto, we implemented all the stencils in the form of affine loop nests enclosed in `scop` pragmas. This code input to Pluto has consists of median 53 source lines of code (SLOC)[12] per benchmark. Pluto takes these `scop`-annotated benchmarks as input to generate optimized versions with median 1157 SLOC. The median of the number of loops in the transformed code was 365 across all the benchmarks with a median nesting depth of 6. Checking such complex codes is greatly helped by FPDetect.

Loop bound and array access bugs: We automated the injection of loop bound bugs. For loop bounds, the injected bug either offsets the lower bound by one in the positive direction or offsets the upper bound by one in the negative direction. We automated the injection of array access bugs. Specifically, array accesses were made incorrect by doubling or halving the indexing term.

False positives: Some bugs do not result in any difference in our bitwise comparison of the result with the non-buggy version. Therefore, our approach incurs no false positives in our evaluation. This is because some source-level bugs do not manifest at runtime. For example, a loop iterator might be constrained by multiple loop bounds (e.g., `min` or `max` of multiple expressions), with the loop bound never reaching the error-injected expressions.[13] In a situation with multiply nested loops where the indexing for an inner loop depends conditionally on outerloop indices, a bug impacting the outer loop bound doesn't often affect the inner loop, and hence the stencil's output.

Table 3 summarizes FPDetect's bug detection effectiveness for the above two categories of bugs. The table only lists detection percentages when the bitwise comparison with the non-buggy version flags an error-injected version as being in error.

The *#SL* enumerate all the possible source locations for the types of bug injected. *#RL* enumerates the number of source locations that were reached at runtime. *%det* denotes the fraction of errors flagged by bitwise comparison with correct execution that was detected by FPDetect. We observe that FPDetect has a high but not perfect detection rate. We investigated the scenarios in which FPDetect missed detection and all of them were a result of the logical error having a low impact on the stencil's output. Specifically, the impacted point's effect on the stencil was beyond the required detector precision (*dp*) evaluated by our conservative error analysis.

To illustrate, consider two cases where FPDetect detect such an impact and another where it escapes FPDetect's detection. In the first case, the detector evaluates to the expected value to `0x40101b539cbac780` (we present in hex values for readability), while the bug affected stencil computes `0x40101b53a11cf49f`. They match in the first 23 bits of the mantissa. This example,

[12] Source lines of code are measured using sloccount tool.
[13] An example can be found in Appendix D of the supplementary material in Reference [14].

Table 3. Software Bug Detection Results

	Loop bound			Array access				Loop bound			Array access		
	#SL	#RL	%det	#SL	#RL	%det		#SL	#RL	%det	#SL	#RL	%det
H1	374	374	100	300	267	99	H2	370	370	100	300	258	100
H3	374	374	100	300	267	98	H4	486	106	99	300	26	100
H5	486	106	100	300	26	100	H6	486	106	100	300	26	100
P1	10	10	100	18	17	100	P2	12	12	100	18	17	100
P3	12	12	100	18	17	100	P4	370	310	100	300	223	87
P5	370	310	96	300	223	82	P6	370	310	96	300	223	82
P7	414	104	98	300	48	97	P8	416	104	100	300	44	98
P9	416	104	98	300	44	98	W1	360	280	100	300	215	55
W2	360	280	44	11	6	54	W3	360	350	31	300	266	45
W4	260	280	45	300	215	60	W5	360	280	53	300	215	62
W6	360	350	100	300	266	100	C1	360	360	100	300	262	100
C2	360	360	100	300	262	100	C3	360	360	100	300	262	100

evaluated for $W1$ has $dp = 30$, hence is trapped. In the second scenario, detector evaluate the expected value to `0x4004178de3e4ab00` while the buggy stencil evaluates `0x4004178de3e4aac4`, differing in only the last 9 bits of mantissa. Given the low detector precision of 30 for wave benchmarks, this error goes undetected. In general, our conservative error analysis and low *udp* choice predicts only upto *dp* bits at the detector, attributing precision beyond *dp* bits to round-off errors. While this leads to missed detection opportunities, we still observe high detection rates.

Loop order bugs: Unlike changes to array accesses and loop bounds, changing loop orders required non-local changes to the Pluto-generated code. Therefore, we handcrafted the error injected versions. The handcrafted scenarios included swapping nested loop pairs and re-ordering of non-nested loop blocks. The optimized codes included a maximum of 360 nested `for` loops with a maximum loop nesting depth of 11. Testing all possible combinations of loop reorderings will be prohibitively expensive. We limited our testing to 15 loop order bug injections per benchmark.

Across all benchmarks, a median of 3 injected bugs per benchmark did not result in a bitwise difference in the output. This could either be due to the reordered version being correct due to commutativity of loops, or the error being too small to persist under finite precision arithmetic for the inputs chosen. These bugs were also not detected by FPDetect. Other than benchmarks W1 and W3, loop order bugs injected in all other benchmarks were detected. In W1 and W3, there was one injected bug in each that resulted in the bitwise comparison flagging an error was not detected by FPDetect. In both cases, the difference was within 8 least significant bits (corresponding to $< 10^{-12}$ relative error), and could not be distinguished from floating point round-off errors.

In summary, we observe that FPDetect can detect a large class of software bugs with a low *udp* of as low as 4 bits. In all cases evaluated, we observed negligible overheads (<2%), making it a useful component in a test suite or as a low cost online checking tool for flagging systematic errors. As observed earlier, increasing the *udp* and *dp* can improve the percentage of bugs detected, at the cost of greater runtime overheads. Note the a change of *udp* only changes the number of points being sampled for detection. The P_w corresponding to this *udp* only decides the relative distance between the sampled points.

Soft error detection: We performed fault injection experiments to validate the coverage guarantees provided by FPDetect. Single bit flips were injected in the kernel source code by corrupting

array locations involved in the stencil computation. Such a fault model abstracts the cause of the fault to an anomalous behavior in data evolution. FPDetect is designed to detect an anomalous behavior in data evolution with a precision guarantee irrespective of where it originates as long as it impacts the computation. The time steps and bit locations subject to injection were selected randomly with equal probability across the iteration space, array index space, and bit locations. The bit flips were injected with equal probability on all array locations, including the unprotected boundaries. Each benchmark was exercised through a fault injection campaign comprising of 10,000 executions to determine the detection rate. In the reported results, FPDetect was evaluated for detector configurations guaranteeing 15 bits of mantissa precision (total of most significant 27 bits including 11 exponent bits and 1 sign bit) with a coverage of 60–90%.

We observe that, across all benchmarks and including the boundary region, errors injected within the most significant *udp* bits (the guaranteed protection) were detected 92–99% of the cases for the 90% coverage, and 78–90% of the cases for the 60% coverage configuration. We observed similar trends when multiple errors are injected, at multiple and randomly selected locations in the iteration space. Figure 10 shows the detection rates for the two coverage scenarios when only considering error affecting the protected region (the most significant 15 bits, labeled “protected”) and only those affecting the unprotected region (labeled “unprotected”). Furthermore, we observe that flips in bits beyond the unprotected region (not guaranteed to be detected) were detected on average 26% (13%) of the cases for 90% (60%) coverage scenario. Hence, even with a conservative configuration for guaranteeing user-defined bits of protection, in practice, our approach *empirically* protects a larger fraction of the computation space and error scenarios.

Recent studies [39] show that single-bit errors yield a higher percentage of SDCs in most cases, compared to multi-bit errors. In cases of data corruption, two or more bit flips in the opposite direction, can reduce the overall error magnitude thus filtering it out of the detectable range. We performed a small scale multi-bit flip experiment to check FPDetect’s detection ability in such scenarios over a selected number of representative benchmarks for a coverage guarantee of 90%. Given that the space of multi-bit flip errors is large, we restrict ourselves to double bit-flip errors with each injection executed at a random time step, splitting the data array into 16 byte sections and selecting one section randomly within which 2 random bits are flipped.

For multi-bit flip campaigns including at least one bit flipped inside the protected range, errors were detected in 90–99% of the cases. Furthermore, multi-bit flips encountered outside the protection range were detected on average for 35% of the cases. Thus, even though our guarantees are restricted to single bit errors, empirically the detectors can handle multi-bit errors.

Comparative study: Our detector synthesis strategy provides a variety of tuning knobs for flexibility in terms of minimum precision and protection coverage with precise bounds. To ascertain the effectiveness of our tool, we compare them with two state-of-the-art soft error detectors, AID [17] and SSD [46], which build data value centric model for soft error detection.

SSD builds an epsilon-insensitive support vector machine regression model to detect SDCs. As spatial features, it includes values of a given point’s neighboring data points as the training data. It runs into scalability issues by requiring the generation of data traces of every point in the simulation over all time steps before being fed to the learning model. We had to scale down the problem to a small size of $1k$ points per dimension. It learns the classifier first, triggering false positives for early stages of the simulation trace. In particular, we encountered false positives in initial one to five iterations. Hence, we devised the fault injection mechanism after atleast 10 iterations have passed. SSD did not trigger a detection in any of our error-injected runs past the first 10 iterations. We believe, in these benchmarks, the variations observed in the first few iterations make SSD consider the error-injected behavior appear normal.

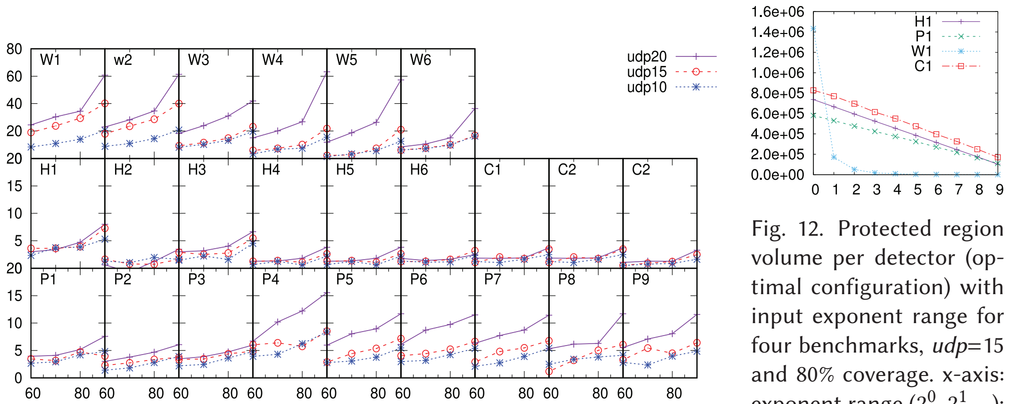

Fig. 11. Sequential benchmark execution overheads for three different user-defined precision (*udp*) values (20, 15, and 10) and varying. X-axis: coverage; y-axis: execution time overhead in percentages.

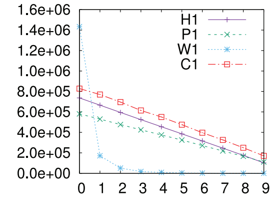

Fig. 12. Protected region volume per detector (optimal configuration) with input exponent range for four benchmarks, *udp*=15 and 80% coverage. x-axis: exponent range ($2^0, 2^1, ..$); y-axis: number of iteration points in the protected region per detector.

AID is an adaptive SDC detector wherein a best-fit prediction model gets selected adaptively based on local online data. AID faired better with our benchmarks than SSD. However, the detection rates and overheads were extremely conservative. The comparative analysis by Kestor et al. [31] reports significant slowdown at a maximum of 1.4× for AID when snapshots were taken for every time step. We observed similar overheads when taking snapshots at every step. To reduce the overhead, we performed two experiments with snapshot every three steps and every 10 steps. In the former case, AID reported similar overheads averaging around 80% with a maximum detection rate of 54%. In the latter case, the overhead reduced to the 20–30% range, while the average detection rate was around 25% with a maximum and minimum of 37% and 21%, respectively.

Sequential overheads: Figure 11 shows the sequential execution overheads of our detection approach for three *udp*-coverage configurations: 10, 15, and 20 bits. We observe that the overhead depends on the user-required coverage guarantee, with overheads, in general, below 10% for *udp*=15 and 80% coverage, for heat, Poisson, and convection-diffusion benchmarks. Overall, we observe that protecting additional bits or providing greater coverage increases the overhead. All wave benchmarks (and some Poisson benchmarks for 90% coverage) incur far greater overheads, reaching over 60% for *udp* of 20 bits and 90% coverage. This is due to the nature of the stencils that, as discuss below, leads to a sharp reduction in the protected region (Figure 12). Despite this increase, these results demonstrate the approach's flexibility in supporting low detection guarantees when a reduced overhead is desired.

Scalability: Figure 15 shows the scalability of the baseline and detector-embedded versions on up to 24 threads using OpenMP. Across all benchmarks, both variants achieve similar speedups, demonstrating that the detectors do not interfere with efficient parallel execution. While not shown here, we observed similar trends (in terms of low overheads) with various thread counts for other user-defined precision and coverage values.

For the wave ("W") benchmarks, which model hyperbolic PDE equations, we encounter reduced scalability for stricter configurations of high *udp* and coverage. We believe this is due to higher *udp* values requiring more detectors. This coupled with the larger E_w results in potentially increased cache contention and hence the associated reduced scalability.

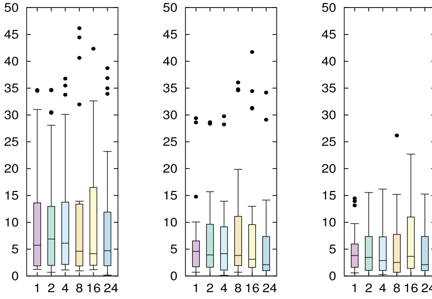

Fig. 13. Detection overheads across all benchmarks for 80% coverage for *udp* values 20, 15, and 10. x-axis: thread count; y-axis: execution overhead in percentage.

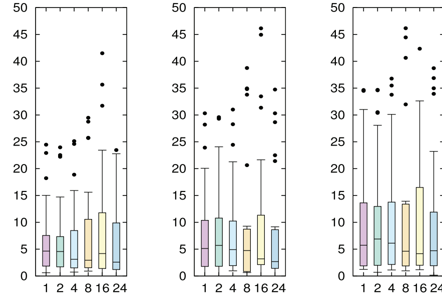

Fig. 14. Detection overheads across all benchmarks for *udp* 20 and coverage of 60%, 70%, and 80%. x-axis: thread count; y-axis: execution time overheads in percentage.

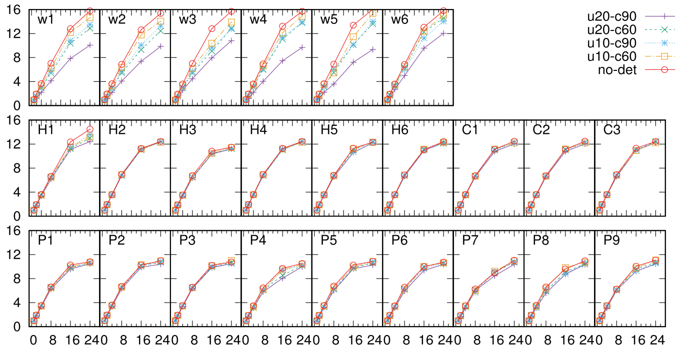

Fig. 15. Scaling on up to 24 threads for the baseline without detectors and with detectors over three combinations of *udp* and coverage [(u=*udp*,c=cov)=(20,90), (10,90), (20,60), (10,60)]. x-axis: thread count; y-axis: speedup over single-threaded execution with no detectors.

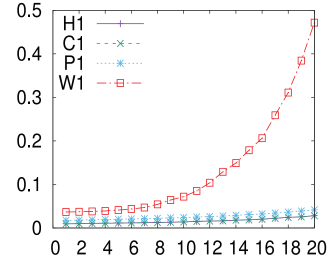

Fig. 16. Optimal cost from Equation (12) (y-axis) vs. *udp* (x-axis) for four benchmarks.

Summarizing the overheads, Figure 13 shows the distribution of overheads for 80% coverage and three *udp* values. Figure 14 shows the distribution of overheads for a fixed *udp* (20) and three different coverage percentages. In both cases, we observe greater coverage percentage or *udp* requirement can increase overheads. However, the median overhead remains close to 5% and does not increase with scale.

Table 4 summarizes the optimal configuration, number of detectors instantiated, detection rates and overhead factors for user defined protection goals requiring *udp* = 20 and 90% coverage for a problem size of 10K ×10K. We select the summary for a subset of the example benchmarks that have shown the largest variations in their configuration and detection rates in their class of kernels.

7.2 Analysis of the Overhead Cost Function

Since FPDetect's detection strategy is sensitive to the specific stencil and the characteristics of the initial/boundary conditions, we have chosen multiple benchmarks and configurations, including different initial and mixed boundary conditions. For a broader analysis of the space of choices evaluated in the offline phase, we examine four candidate benchmarks—h1, c1, w1, and p1–in detail.

Table 4. Summary Table for Optimal Configurations, Number of Detectors, Detection Rate, Sequential Overhead Overhead and Scalability Factors (k Threads) for udp of 20 and Coverage 90% for a Problem Size of 10K×10K

Benchmark	optimal config		Num Dets	% det		% Seq ovh	Scalability factor		
	(T,dp_T)	(ew,pw)		single-bit	Multi-bit		k=2	k=8	k=24
H1	(254,39)	(63,14)	107584	96.14	94.66	8.0	1.76	6.18	12.41
H4	(256,38)	(63,14)	107584	90.47	91.08	3.81	1.84	6.54	12.03
P1	(160,40)	(69,24)	37620	94.12	93.88	7.59	1.78	6.27	10.41
P4	(160,37)	(69,17)	76729	93.65	94.92	15.55	1.67	5.78	9.96
P7	(160,37)	(69,17)	76729	93.65	95.05	11.43	1.65	6.02	10.35
W1	(10,36)	(20,3)	1575261	99.25	99.48	60.78	1.20	4.13	10.07
W4	(10,36)	(20,6)	2546440	99.25	99.74	41.91	1.15	4.03	9.67
C1	(250,38)	(66,15)	94249	92.10	90.76	3.72	1.82	6.54	12.03

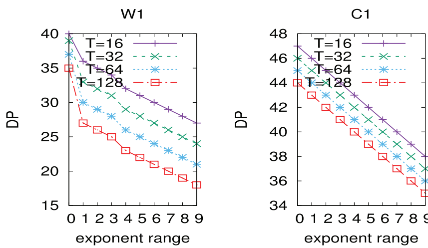

Fig. 17. Maximum bits preserved by iterative stencil (y-axis) as function of input exponent range (x-axis), determined by our analysis, for two benchmarks (w1 and c1) for different maximum T-step choices.

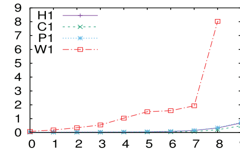

Fig. 18. Optimal configuration's cost for different input exponent ranges for a *udp* of 20 and 60% coverage. x-axis: input exponent range ($2^0, 2^1, ..$); y-axis: cost for optimal configuration (from Equation (12)).

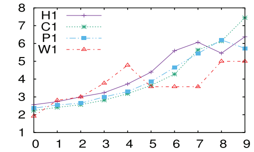

Fig. 19. Optimal configuration's E_w to P_w ratio for different input exponent ranges for a *udp* of 20 and 90% coverage. x-axis: input exponent range ($2^0, 2^1, ..$); y-axis: E_w/P_w for optimal configuration (from Equation (12)).

Variation of essential and protected widths: Figure 12 shows the variation in optimal P_w volume with input exponent range. W1, a second-order wave equation representing hyperbolic PDEs, exhibits the highest overhead and cost, since its P_w volume rapidly diminishes with increasing exponent sizes. Benchmarks h1 and c1 exhibit a slower change in the optimal P_w.

Figure 19 plots the ratio of E_w to P_w for their optimal configuration with varying input exponent range for a fixed *udp* and *cov*. Larger binade differences in the input values result in increasing separation between the E_w and P_w values, often increasing the E_w to P_w ratio. Together with ρ (Equation (12)) this can potentially lead to an increase in detector overhead.

Impact of input range and *udp*: Figure 18 shows the cost function values for four benchmarks (h1, c1, p1, and w1) for hypothetical range of input values from 2^0 to 2^9 and user-defined precision of 20 bits. We observe that cost increases with input range, especially for w1. Thus, accurate yet efficient evaluation of the input range can help reduce the detector overheads.

Figure 16 shows the evaluated cost function value as *udp* is varied. Similar to the input exponent range, we observe that some benchmarks (especially w1) are more sensitive than others to increase in *udp*. Thus, careful choice of *udp* can maximize coverage while minimizing overhead.

8 ADDITIONAL RELATED WORK

Floating-point error analysis was the central driving concept in our work. Boldo [5, 6], Darulova [13], Magron [34], and Solovyev [44] are three recent pieces of work that conduct rigorous error analysis. Zhang [56] uses *reduced precision check* to detect errors in the floating point units as a hardware solution. Daumas et al. reason about floating-point operations using interval arithmetic [15]. These tools focus on programs operating on fixed and small number of inputs. They are unable to handle the kinds of complexity presented by stencil expressions unfolded in time. They also do not steer their analysis toward the synthesis of online error detectors like in this work.

We exploit the structure of stencil operations to simplify analysis of *parametrically* sized programs. Kramer established worst-case bounds for interval arithmetic [32]. We build on the guarantees for individual operations (IEEE 754 [30]) to analyze worst-case bounds for stencil programs.

Chisel [35] and Rely [9] consider potentially erroneous execution of portions of a program by analyzing the probability of the output being erroneous. They track the probability of an erroneous output rather than its magnitude. Soft error analysis has been performed for specific algorithm classes (e.g., linear algebra [53, 54] and iterative solvers [20, 49]). Huang and Abraham introduced algorithm-based fault tolerance (ABFT) [28] to detect errors in matrix multiplication related operations. Elliott et al. [19] present selective reliability to provide numerical bounds on anticipated behavior and use this analysis in the design of resilient algorithms [18, 21]. Our work focuses on soft error detection for stencil programs. Application-independent approaches for iterative programs rely on observing the evolution of a value over time to detect anomalies (e.g., AID [17] uses curve fitting, SSD [46] uses support vector machines (SVM) regression, and Reference [42] uses a machine learning-based approach to build regression models for synthesizing low cost detectors). Gomez and Capello exploits multivariate interpolation to detect and correct corruption in stencil application [25]. Xiaoguang [55] presents a *grid sampling*-based DMR scheme that determines the sampling points based on the error propagation pattern in the grid.

Gamell considers local recovery from fail-stop errors affecting stencil programs [23]. Fang et al. [22] analytically model application overhead of recovery from detected soft errors via localized recovery. Our approach can complement such recovery algorithms via efficient error detection.

9 CONCLUSIONS AND FUTURE WORK

In this article, we present FPDetect, an approach to detect both logical bugs and soft errors in the data space of stencil computations. Schemes comparable to FPDetect have been observed to generate false positives, incur higher overheads, or not provide similar rigorous guarantees. We report FPDetect's overheads for different thread counts as well as its performance in conjunction with polyhedral optimizations for various user-defined precision values. We believe FPDetect can be used as part of a holistic error detection system (e.g., involving cross-layer concerns [10]) in which the most impactful of errors affecting stencil programs can be protected.

As future work, we will investigate the use of FPDetect for runtime precision profiling, given that developers often use high precision as a safety net for floating point errors, which may be wasteful in many cases. FPDetect's evaluation units are uniquely designed to allow it to serve as a very close proxy to real values at runtime. Thus, it can help profile runtime precision requirements. In addition to re-instating confidence in the evolving results, this approach may enable the user to dynamically tune the working precision based on the stability of the evolving results. To this end, we have prototyped a machine learning model built on profiled simulation data that attempts to predict the minimum precision around a rectangular region centered around a point for which FPDetect's evaluation unit was instantiated. Current results show encourag-

ing trends with prediction accuracies within two precision bits. Some of our ongoing work aims to leverage this information in tuning simulation parameters for improved performance. Furthermore, multi-bit flips will encompass a larger detectable cross section (unless they cancel out), hence specialized detectors for this purpose might allow for better detection and increased coverage.

REFERENCES

[1] IEEE. 2008. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008* (Aug. 2008), 1–70.

[2] George A. Articolo. 2009. *Partial Differential Equations & Boundary Value Problems with Maple, Second Edition* (2nd ed.). Academic Press, Orlando, FL.

[3] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2016. PolyCheck: Dynamic verification of iteration space transformations on affine programs. In *Proceedings of the POPL*. 539–554.

[4] R. Baumann. 2005. Soft errors in advanced computer systems. *IEEE Design Test Comput.* 22, 3 (May 2005), 258–266. DOI: https://doi.org/10.1109/MDT.2005.69

[5] Sylvie Boldo and Jean-Christophe Filliâtre. 2007. Formal verification of floating-point programs. In *Proceedings of the ARITH*. 187–194.

[6] Sylvie Boldo and Thi Minh Nguyen. 2011. Proofs of numerical programs when the compiler optimizes. *Innov. Syst. Softw. Eng.* 7, 2 (June 2011), 151–160. DOI: https://doi.org/10.1007/s11334-011-0151-6

[7] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proceedings of the ETAPS CC*.

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the PLDI*. ACM, New York, NY, 101–113.

[9] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the OOPSLA*. 33–52.

[10] Eric Cheng, Shahrzad Mirkhani, Lukasz G. Szafaryn, Chen-Yong Cher, Hyungmin Cho, Kevin Skadron, Mircea R. Stan, Klas Lilja, Jacob A. Abraham, Pradip Bose, and Subhasish Mitra. 2018. Tolerating soft errors in processor cores using CLEAR. *IEEE Trans. CAD Integr. Circ. Syst.* 37, 9 (2018), 1839–1852.

[11] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. 2017. Rigorous floating-point mixed-precision tuning. In *Proceedings of the POPL*. 300–315.

[12] Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. In *Proceedings of the POPL*. 235–248.

[13] Eva Darulova and Viktor Kuncak. 2017. Towards a compiler for reals. *ACM Trans. Program. Lang. Syst.* 39, 2 (Mar. 2017).

[14] Arnab Das, Sriram Krishnamoorthy, Ian Briggs, Ganesh Gopalakrishnan, and Ramakrishna Tipireddy. 2020. FPDetect: Efficient Reasoning About Stencil Programs Using Selective Direct Evaluation. arxiv:cs.DC/2004.04359.

[15] Marc Daumas, Guillaume Melquiond, and César A. Muñoz. 2005. Guaranteed proofs using interval arithmetic. In *Proceedings of the ARITH*. 188–195.

[16] Luiz Henrique de Figueiredo and Jorge Stolfi. 2004. Affine arithmetic: Concepts and applications. *Numer. Algor.* 37, 1 (Dec. 2004), 147–158.

[17] Sheng Di and Franck Cappello. 2016. Adaptive impact-driven detection of silent data corruption for HPC applications. *Trans. Parallel Distrib. Syst.* 27, 10 (2016), 2809–2823.

[18] James Elliott, Mark Hoemmen, and Frank Mueller. 2014. Evaluating the impact of SDC on the GMRES iterative solver. In *Proceedings of the IPDPS*. 1193–1202.

[19] James Elliott, Mark Hoemmen, and Frank Mueller. 2014. Resilience in numerical methods: A position on fault models and methodologies. *CoRR* abs/1401.3013 (2014).

[20] James Elliott, Mark Hoemmen, and Frank Mueller. 2015. A numerical soft fault model for iterative linear solvers. In *Proceedings of the HPDC*. 271–274.

[21] James Elliott, Mark Hoemmen, and Frank Mueller. 2016. Exploiting data representation for fault tolerance. *J. Comput. Sci.* 14 (2016), 51–60.

[22] Aiman Fang, Aurélien Cavelan, Yves Robert, and Andrew A. Chien. 2017. Resilience for stencil computations with latent errors. In *Proceedings of the ICPP*. 581–590.

[23] Marc Gamell, Keita Teranishi, Michael A. Heroux, Jackson Mayo, Hemanth Kolla, Jacqueline Chen, and Manish Parashar. 2015. Local recovery and failure masking for stencil-based applications at extreme scales. In *Proceedings of the SC*. 70:1–70:12.

[24] David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* 23, 1 (Mar. 1991), 5–48.

[25] L. A. B. Gomez and F. Cappello. 2015. Detecting and correcting data corruption in stencil applications through multivariate interpolation. In *Proceedings of the CLUSTER*. 595–602.

[26] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. DOI: https://doi.org/10.1145/3282307

[27] Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). Society for Industrial and Applied Mathematics. Retrieved from https://epubs.siam.org/doi/pdf/10.1137/1.9780898718027.

[28] Kuang-Hua Huang and Jacob A. Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.* 33, 6 (1984), 518–528.

[29] Padma Jayaraman and Ranjani Parthasarathi. 2017. A survey on post-silicon functional validation for multicore architectures. *ACM Comput. Surv.* 50, 4 (Aug. 2017). DOI: https://doi.org/10.1145/3107615

[30] William Kahan. 1996. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes Status IEEE* 754, 94720-1776 (1996), 11.

[31] Gokcen Kestor, Burcu Ozcelik Mutlu, Joseph Manzano, Omer Subasi, Osman Unsal, and Sriram Krishnamoorthy. 2018. Comparative analysis of soft-error detection strategies: A case study with iterative methods. In *Proceedings of the CF*. 173–182.

[32] Walter Krämer. 1997. A priori worst-case error bounds for floating-point computations. In *Proceedings of the ARITH*. 64.

[33] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2015. Clover: Compiler directed lightweight soft error resilience. *SIGPLAN Not.* 50, 5 (June 2015). DOI: https://doi.org/10.1145/2808704.2754959

[34] Victor Magron, George Constantinides, and Alastair Donaldson. 2017. Certified roundoff error bounds using semidefinite programming. *ACM Trans. Math. Softw.* 43, 4 (Jan. 2017).

[35] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the OOPSLA*. 309–328.

[36] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. 2009. *Handbook of Floating-Point Arithmetic*. Birkhauser.

[37] H. Quinn and P. Graham. 2005. Terrestrial-based radiation upsets: A cautionary tale. In *Proceedings of the FCCM*. 193–202. DOI: https://doi.org/10.1109/FCCM.2005.61

[38] Jude A. Rivers, Meeta S. Gupta, Jeonghee Shin, Prabhakar N. Kudva, and Pradip Bose. 2011. Error tolerance in server class processors. *IEEE Trans. CAD Integr. Circ. Syst.* 30, 7 (2011), 945–959.

[39] B. Sangchoolie, K. Pattabiraman, and J. Karlsson. 2017. One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors. In *Proceedings of the DSN*. 97–108.

[40] Markus Schordan, Pei-Hung Lin, Daniel J. Quinlan, and Louis-Noël Pouchet. 2014. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *Proceedings of the ISoLA*. 493–508.

[41] N. Seifert. 2010. *Radiation-induced Soft Error: A Chip-level Modeling*. Delft, The Netherlands.

[42] Vishal Sharma, G. Gopalkrishnan, and Greg Bronevetsky. 2015. Detecting soft errors in stencil based computations. In *the 11th IEEE Workshop on Silicon Errors in Logic – System Effects (SELSE'15)*.

[43] Marc Snir, Robert W. Wisniewski, Jacob A. Abraham, Sarita V. Adve, Saurabh Bagchi, et al. 2014. Addressing failures in exascale computing. *Proceedings of the IJHPCA* 28, 2 (2014), 129–173.

[44] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2019. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. *ACM Trans. Program. Lang. Syst.* 41, 1 (2019), 2:1–2:39.

[45] Omer Subasi, Sheng Di, Prasanna Balaprakash, Osman S. Unsal, Jesús Labarta, Adrián Cristal, Sriram Krishnamoorthy, and Franck Cappello. 2017. MACORD: Online adaptive machine learning framework for silent error detection. In *Proceedings of the CLUSTER*. 717–724.

[46] Omer Subasi, Sheng Di, Leonardo Bautista-Gomez, Prasanna Balaprakash, Osman S. Ünsal, Jesús Labarta, Adrián Cristal, and Franck Cappello. 2016. Spatial support vector regression to detect silent errors in the exascale era. In *Proceedings of the CCGrid*. 413–424.

[47] Omer Subasi and Sriram Krishnamoorthy. 2017. A gaussian process approach for effective soft error detection. In *Proceedings of the CLUSTER*. 608–612.

[48] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The pochoir stencil compiler. In *Proceedings of the SPAA*. 117–128.

[49] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z. Zhang, Darren J. Kerbyson, and Zizhong Chen. 2016. New-Sum: A novel online ABFT scheme for general iterative methods. In *Proceedings of the HPDC*. 43–55.

[50] Devesh Tiwari, Saurabh Gupta, George Gallarno, Jim Rogers, and Don Maxwell. 2015. Reliability lessons learned from GPU experience with the titan supercomputer at oak ridge leadership computing facility. In *Proceedings of the SC*. ACM, New York, NY.

[51] Ohio State University. 2012. the PolyOpt Polyhedral Compiler. Retrieved from http://hpcrl.cse.ohio-state.edu/wiki/index.php/Polyhedral_Compilation.

[52] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2012. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. Program. Lang. Syst.* 34, 3 (2012), 11:1–11:35.

[53] Panruo Wu and Zizhong Chen. 2014. FT-ScaLAPACK: Correcting soft errors on-line for ScaLAPACK cholesky, QR, and LU factorization routines. In *Proceedings of the HPDC*. 49–60.

[54] Panruo Wu, Nathan DeBardeleben, Qiang Guan, Sean Blanchard, Jieyang Chen, Dingwen Tao, Xin Liang, Kaiming Ouyang, and Zizhong Chen. 2017. Silent data corruption resilient two-sided matrix factorizations. In *Proceedings of the PPoPP*. 415–427.

[55] Ren Xiaoguang, Xu Xinhai, Wang Qian, Chen Juan, Wang Miao, and Yang Xuejun. 2015. GS-DMR: Low-overhead soft error detection scheme for stencil-based computation. *Parallel Comput.* 41 (2015), 50–65.

[56] Yaqi Zhang, Ralph Nathan, and Daniel J. Sorin. 2015. Reduced Precision Checking to Detect Errors in Floating Point Arithmetic. arxiv:cs.NA/1510.01145.

Received November 2019; revised April 2020; accepted May 2020