Aloe: Verifying Reliability of Approximate Programs in the Presence of Recovery Mechanisms

Keyur Joshi
University of Illinois at Urbana-Champaign, USA
kpjoshi2@illinois.edu

Vimuth Fernando
University of Illinois at Urbana-Champaign, USA
wvf2@illinois.edu

Sasa Misailovic
University of Illinois at Urbana-Champaign, USA
misailo@illinois.edu

Abstract

Modern hardware is becoming increasingly susceptible to silent data corruptions. As general methods for detection and recovery from errors are time and energy consuming, selective detection and recovery are promising alternatives for applications that have the freedom to produce results with a variable level of accuracy. Several programming languages have provided specialized constructs for expressing detection and recovery operations, but the existing static analyses of safety and quantitative analyses of programs do not have the proper support for such language constructs.

This work presents Aloe, a quantitative static analysis of reliability of programs with recovery blocks – a construct that checks for errors, and if necessary, applies the corresponding recovery strategy. The analysis supports reasoning about both reliable and potentially unreliable detection and recovery mechanisms. It implements a novel precondition generator for recovery blocks, built on top of Rely, a state-of-the-art quantitative reliability analysis for imperative programs. Aloe can reason about programs with scalar and array expressions, if-then-else conditionals, and bounded loops without early exits. The analyzed computation is idempotent and the recovery code re-executes the original computation.

We implemented Aloe and applied it to a set of eight programs previously used in approximate computing research. Our results present significantly higher reliability and scale better compared to the existing Rely analysis. Moreover, the end-to-end accuracy of the verified computations exhibits only small accuracy losses.

CCS Concepts • **Theory of computation → Program specifications**; **Program verification**.

Keywords Reliability, Approximate Computing

CGO '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7047-9/20/02…$15.00
https://doi.org/10.1145/3368826.3377924

ACM Reference Format:
Keyur Joshi, Vimuth Fernando, and Sasa Misailovic. 2020. Aloe: Verifying Reliability of Approximate Programs in the Presence of Recovery Mechanisms. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368826.3377924

1 Introduction

With the end of Dennard scaling and the slowdown of Moore's law, hardware is becoming increasingly susceptible to operational errors, due to imperfect manufacturing, aging, and variations of environmental factors such as temperature, voltage, or radiation [4, 7, 15, 36, 43]. While many hardware errors are still routinely caught and corrected at the hardware level, some errors can propagate across the system stack and silently corrupt program data. Such silent data corruptions (SDCs) can lead to unacceptable program results. Classical solutions for detecting and correcting SDCs come at a high cost, e.g., indiscriminate instruction replication or n-module redundancy can double or triple a program's energy usage.

More recently, researchers observed that many modern applications have the freedom to produce results with variable level of accuracy. Example workloads include image and video processing, machine learning, big-data analytics, probabilistic inference, and sensing. These applications can tolerate *selective* SDC detection – identifying that some subcomputation experienced an error and produced an unacceptable result – and *low-cost* recovery – recomputing the correct or approximate result, or sometimes continuing the execution without correction at all. Guided by these observations, various automated and developer-assisted techniques enable developing acceptably reliable executions with reduced time/energy cost [11, 13, 16, 18, 22, 26–28, 31, 39–41].

Several reliability-aware languages, e.g. Relax [13] and Topaz [1], expose *recovery blocks* to the developer. These blocks have the following conceptual form:

```
try { code; }
check { checker(programState); }
recover { code; }
```

Potentially unreliable code executes inside the `try` block. The `checker` function analyzes the program state and attempts to detect potential errors in the computation. It can use various exact procedures (e.g., checking that a SAT formula is correct is much less time consuming than finding the solution) or approximate procedures (e.g., anomaly detection [1]). If the check fails, the execution runs the recovery code, inside the `recover` block. Recovery blocks are appealing as they empower the developers with fine-grained control over customized recovery strategies and can provide hints to program analyses and compilers on how to generate code. However, existing languages cannot rigorously analyze the reliability of recovery blocks.

Statically analyzing quantitative reliability and safety of computations has been a topic of increased interest in the programming languages community. Safety analyses that ensure that programs do not violate important logical invariants include EnerJ's type analysis [42], RHL relational safety [8, 23], and verification with first-class execution fault models in Leto [5]. Quantitative reliability denotes the probability with which a computation with unreliable operations produces the correct results. Rely [10], Chisel [33], Decaf [6], and Parallely [20] all compute the probability with which an unreliable execution produces the same (or acceptably similar) result as a fully reliable execution. For instance, a reliability of 0.99 states that the program will generate the same result as the reliable execution on 99% of runs.

Despite the ability to successfully determine reliability bounds on unreliable executions, existing reliability analyzes are significantly imprecise when analyzing computations with SDC detection and recovery mechanisms. At best, they treat these constructs as arbitrary conditional code, leading to a uniform reduction in the reliability of the computation. This is imprecise, because recovery blocks will typically not reduce the reliability of the computation, even if the code within the `try` block is unreliable. Supporting detection and recovery mechanisms within the quantitative static analyses will greatly improve precision, and consequently the ability of these analyses to analyze larger computations and ensure that the computations have the desired reliability properties.

Our Work. We present Aloe, the first static analysis of quantifiable reliability of programs that include constructs for selective SDC detection and recovery. Our language extensions expose recovery blocks and multiple detection and recovery choices to the developer. Aloe can compute the impact of both the `checker` function and the `recover` block. We support both *perfect* and *imperfect* checkers. A perfect checker will always detect whether or not an error occurred while executing the `try` block. An imperfect checker may occasionally fail to detect errors (false negative) or may detect an error even when no error occurred (false positive). We analyze computations with idempotent code inside the `try` block (i.e. the code can be re-executed without having previous executions affect the results of the new execution) and `recover` blocks that re-execute the computation either reliably or unreliably.

Aloe's analysis builds on top of Rely's precondition generator, which computes a quantitative reliability predicate for each statement in the program. Our analysis specifies the rules for precisely analyzing recovery blocks with multiple recovery strategies. We also provide new rules for simplification of reliability predicates that improve the applicability of our analysis. The analysis time is proportional to the number of statements in the program after unrolling bounded loops (as in Rely). Our approach directly extends to other similar analyses [6, 20, 33]. We believe that this approach provides a valuable guidance for future compiler infrastructures that automate selective reliability analysis and placement.

Results. We implement our analysis and apply it to eight programs from various benchmarks that were previously used in approximate computing research. We use the specifications of the unreliable hardware from EnerJ [42] and the specifications of the approximate checkers from Topaz [1].

Our experiments show that the analysis of the recovery blocks is significantly more precise than the existing analysis in Rely – Aloe verifies all kernels identified in each benchmark for the reliability bound 0.9999. Aloe also verifies the end-to-end reliability of the program's output for the bound 0.99. These results are orders of magnitude more precise than the baseline Rely analysis, which is not able to verify any of the kernels or end-to-end bounds. We also show that kernels with recovery blocks produce results with acceptable reliability even when the checkers are imperfect.

Contributions. Our paper makes several key contributions:

- **Concept:** We present the concept of quantifiable reliability for computations with recovery blocks.
- **Analysis with Perfect Checkers:** We present a Rely-style analysis for programs with recovery blocks and perfect checkers.
- **Analysis with Imperfect Checkers:** We present how to extend our analysis to operate with imperfect checkers that may report non-existent errors, or fail to identify existing errors.
- **Evaluation:** We show the effectiveness of our analysis and generate precise reliability bounds for a set of eight real-world approximate computations.

2 Example

We will use PageRank as our running example. PageRank is a link analysis algorithm that ranks web pages according to their importance. Figure 1 presents the implementation of the PageRank kernel for a single node in a graph.

The PageRank of an node is a weighted sum of the PageRanks of each incoming edge. The PageRank is updated in this manner over multiple iterations. This computation is known to be tolerant to errors, partly due to its iterative nature.

We study the execution of this kernel on an unreliable hardware, in which the arithmetic operations can fail with

```
newPagerank = 0.15;
inlink = Inlinks[i];
j = 0;
repeat MaxEdges {
  if j < inlink {
     neighbor = Edges[j];
     outN = Outlinks[neighbor];
     current = Pageranks[neighbor];
     temp = 0.85 * current;
     temp = temp / outN;
     newPagerank = newPagerank + temp;
     j = j + 1;
  };
};
```

Figure 1. PageRank Kernel.

some probability. For example, EnerJ [42] presents several approximation strategies that provide arithmetic instructions that produce erroneous results with probability 10^{-6}, 10^{-4}, or 10^{-2} but save energy. We model such instructions using the probabilistic-choice statements as $x = e_{correct}$ $[p]$ `rand()`; the arithmetic operation succeeds with probability p, but results in a random error value with probability $1 - p$.

2.1 Try-Check-Recover Blocks

A common approach to increase the reliability of computations is to run them using unreliable hardware, check if an error occurred during the execution of the computation, and redo the computation using precise hardware. Figure 2 presents this approach for the PageRank kernel. The program first runs the statements in the try block. The arithmetic operations in the try block are unreliable and only produce the correct value with probability 0.999. The execution continues until the end of the try block.

Checking. Then, the program evaluates the checker function (`checker`) on the program state to detect if there was an error in the computation. If the checker detects an error, it runs the recover block, which has fully reliable arithmetic operations. If the checker can identify all errors, the overall computation becomes fully reliable. Moreover, if the execution of the checker is inexpensive, and the errors are infrequent, the overall computation may be faster and/or more energy efficient than the original program, since the recover block will be rarely executed.

The checker needs to ensure full correctness, i.e. all output variables have correct values. Error detection mechanisms can either be implemented in hardware [32, 35] or in software. Software techniques include re-executing the computation and comparing the results (useful when the try block is much faster than the recover block), or verifying the result with a verification algorithm (e.g. results of NP-complete problems are verifiable in P time). In addition, machine learning can be used to identify outliers in execution [1]. For our example, the code sets an internal flag when an error occurs in the try block, and the checker detects whether this flag was set (similar to Relax [13]).

```
newPagerank = 0.15;
inlink = Inlinks[i];
j = 0;
repeat MaxEdges {
  if j < inlink {
    neighbor = Edges[j];
    outN = Outlinks[neighbor];
    current = Pageranks[neighbor];
    try {
      temp = 0.85 * current [0.999] rand();
      temp = temp / outN [0.999] rand();
      sum = newPagerank + temp [0.999] rand();
    } check { checker(sum, current, outN, newPagerank) }
    recover {
      temp = 0.85 * current [1] rand();
      temp = temp / outN [1] rand();
      sum = newPagerank + temp [1] rand();
    };
    newPagerank = sum;
    j = j + 1;
  };
};
```

Figure 2. PageRank Kernel Run on Unreliable Hardware.

Recovery. If the checker function detects an error during the execution, the recover block is executed to redo the computation in precise hardware that is guaranteed to produce a correct result (i.e. probability is 1.0).

2.2 Verification of Reliability

We define reliability as the probability that the results of the computation are correct (equal to the results of an execution with no errors). We want to verify that the reliability of the calculated variable `newPagerank` is fully reliable – equal to 1.0. It depends on the probability that the arithmetic operations produce correct values in the try block. However, if the checker can identify the errors, the computation will proceed to the recover block, which recomputes the values correctly, and therefore the final result will always be exact.

Analysis Requirements. Aloe's reliability analysis requires that the computation of the try block be idempotent (i.e. the try block can be re-executed without having previous executions affect the new execution) [14]. This allows us to re-execute the computation without expensive checkpoints. In addition, the computation of the try and recover blocks should execute equivalent computations that read from the same input variables and write to the same output variables. If the try and recover block perform completely different computations, then the results of the two blocks would not be comparable from a reliability viewpoint.

Encoding detection and recovery in Rely. The existing quantitative reliability analysis from Rely cannot be used to accurately calculate the reliability of a try-check-recover block. To represent this computation in Rely, one can convert the try-check-recover statement to a conditional statement, where S_{try} represents the instructions in the try block of our try-check-recover statement, and S_{rec} represents the instructions in the recover block:

```
⎡ S_try;                                                      ⎤
⎢ if ( checker(sum, current, outN, newPagerank) ) {           ⎥
⎢    skip;                                                    ⎥
⎢ } else {                                                    ⎥
⎢   S_rec;                                                    ⎥
⎣ }                                                           ⎦
```

The semantics of this computation closely mirror those of our try-check-recover block. However, Rely's analysis cannot infer that $\mathrm{S}_{\mathrm{rec}}$ only executes when $\mathrm{S}_{\mathrm{try}}$ fails and that it eliminates the error produced by $\mathrm{S}_{\mathrm{try}}$. It instead conservatively assumes that errors in $\mathrm{S}_{\mathrm{try}}$ can remain uncorrected.

2.3 Results

For `maxEdges` = 8, Aloe's analysis can automatically show that the reliability of `newPagerank` is 1.0 due to the recovery mechanism detecting and fixing all errors. The analysis runs in 3ms. In contrast, Rely's analysis is too conservative – it calculates a reliability of ~ 0.976. Aloe can analyze several other interesting scenarios which we describe next.

Recovery blocks can be unreliable. Suppose arithmetic instructions in the recover block can fail with probability 10^{-4}. Aloe computes that the reliability of the try and recover blocks is ~ 0.997 and ~ 0.9997 respectively. Since both try and recover must fail for try-check-recover block to fail, the overall reliability of the try-check-recover block ~ 0.9999991. When `maxEdges = 8`, the reliability of the entire kernel is ~ 0.999993. In this way, it is possible to combine unreliable components to produce more reliable components. In contrast, Rely's analysis calculates the reliability of the kernel as ~ 0.976.

Checkers can be unreliable. We also analyzed the impact of an imperfect checker function on the reliability of `newPagerank`. For a checker function that detects 95% of erroneous runs, and may detect spurious errors 5% of the time, Aloe calculated the `newPagerank` reliability of ~ 0.9999.

Recovery can be repeated multiple times. We can apply the re-execution strategy multiple times, to further increase the overall reliability. For convenience, we added the keyword `redo[n]`, which nests the try-check-recover statement inside the recover block n times.

3 Background

We review Rely's [10] quantitative reliability analysis.

Reliability Predicates. We generate reliability predicates to constrain the reliability of an approximate program. A reliability predicate Q has the following form:

$$R_f := r \mid \mathcal{R}(O) \mid r \cdot \mathcal{R}(O)$$
$$Q := r \leq R_f \mid Q \wedge Q$$

A reliability factor (R_f) is either a number r, a *joint reliability predicate* representing the probability that a set of variables O has the same values in an approximate execution as an exact, error-free execution, or a product of a number r and a *joint reliability predicate*. A reliability predicate (Q) is either a comparison between a number and a reliability factor or a conjunction of predicates.

For example, we can specify the constraint that the reliability of some variable x in a program is at least 0.99 (99%) using the reliability predicate $0.99 \leq \mathcal{R}(\{x\})$. In this predicate, $\mathcal{R}(\{x\})$ is the probability that an approximate execution of the program generates the same value for x as an exact, error-free execution.

Reliability Precondition Generation. The reliability precondition generator $\boxed{C \in S \times Q \mapsto Q}$ is a function that takes as inputs a statement and a postcondition that must be satisfied after executing the statement and produces the corresponding precondition as the output. We use the existing precondition generation rules as in Rely and extend them to our new recovery constructs as described in Sections 5 and 6. Some rules of interest are:

$$\begin{aligned} C(\, x = e,\, Q\,) &= Q\; [\, \mathcal{R}(\, \rho(e) \cup X\,) \;/\; \mathcal{R}(\, \{x\} \cup X\,)\,] \\ C(\, x = e_1\; [r]\; e_2,\, Q\,) &= Q\; [\, r \cdot \mathcal{R}(\, \rho(e_1) \cup X\,) \;/\; \mathcal{R}(\, \{x\} \cup X\,)\,] \\ C(\, \texttt{if}\; x\; \{S_1\}\; \texttt{else}\; \{S_2\},\, Q\,) &= C(\, S_1,\, Q\,) \wedge C(\, S_2,\, Q\,) \end{aligned}$$

For a simple assignment of an expression, any reliability specification containing x is updated such that x is replaced by the variables occurring in e ($\rho(e)$) as the reliability of x is only dependent on the reliability of variables used in the expression. For probabilistic assignment, the reliability of x is equal to r (the probability of the assignment evaluating *without errors*) times the reliability of variables occurring in e_1. Conditionals are analyzed as a nondeterministic choice between the *if* and *else* branches. The precondition for a conditional statement is the conjunction of the preconditions generated from the two branches. The reliability of the branch variable is incorporated into the analysis via conditional flattening ([10] Section 5.1). Bounder loops in Rely are unrolled into a sequence of nested conditionals.

Substitution. The substitution for reliability predicates in Rely, $e_0\,[\, e_2 \,/\, e_1\,]$, replaces all occurrences of the expression e_1 with the expression e_2 within the expression e_0. The substitution matches set patterns, e.g., the pattern $R(\{x\} \cup X)$ is a joint reliability factor that contains the variable x, alongside with the remaining variables in the set X. The result of $r \cdot \mathcal{R}(\{x, z\})\, [\, \mathcal{R}(\{y\} \cup X) \,/\, \mathcal{R}(\{x\} \cup X)\,]$ is $r \cdot \mathcal{R}(\{y, z\})$.

4 Language

Figure 3 presents the syntax of the language used in Aloe. The Aloe language supports scalar and array expressions, if-then-else conditionals, and bounded loops without early exits. To analyze the reliability of a program, we require three main components:

- **Program:** a program written in the Aloe language. The language adds support for recovery mechanisms to Rely.
- **Approximation Models:** specifies the probabilities that instructions produce incorrect results. We can define multiple models (e.g., separate for try and recover blocks).

n	$\in \mathbb{N}$	*quantities*
$\mathtt{m}$	$\in \mathbb{N} \cup \mathbb{F}$	*values*
r	$\in [0, 1.0]$	*probability*
x, b	$\in$ Var	*variables*
a	$\in$ ArrVar	*array variables*
f	$\in$ Func	*external functions*
op	$\in \{+, -, \ldots\}$	*arithmetic operators*
Exp	$\rightarrow m \mid x \mid f(Exp^*) \mid (Exp) \mid Exp\ op\ Exp$	*expressions*
t	$\rightarrow$ `int<n>` | `float<n>`	*basic types*
D	$\rightarrow$ t x | t a[n^+] | $D; D$	*variable declarations*
P	$\rightarrow$ D;S	*program*

recovery $\rightarrow$	
redo[n]	*redo up to n times*
| redo[ψ]	*redo on different reliability model*
| S	*other (custom) recovery*
$S \rightarrow$	
`skip`	*empty program*
| x = Exp	*assignment*
| x = Exp [r] Exp	*probabilistic choice*
| S ; S	*sequence*
| x = a[Exp^+]	*array load*
| a[Exp^+] = Exp	*array store*
| `if` Exp {S} `else` {S}	*branching*
| repeat n {S}	*repeat n times*
| x = (T)Exp	*cast*
| try {S} check {Exp} recover {recovery}	*try-check-recover*

Figure 3. Syntax

- **Specification of checker functions:** specifies the behavior of checker functions used to check for errors in execution. For each checker function it defines the false-positive rate (p_{FP}), and the false-negative rate (p_{FN}).

The syntax represents a subset of Rely [10]. We use the probabilistic choice statement from Parallely [20] to explicitly represent unreliable operations and their failure probability (in lieu of the '+.' notation and implicit hardware model).

4.1 Semantics

Approximation Model. An approximation model specifies the runtime behavior of the environment. It is a tuple $\psi \in (x \rightarrow \mathbb{R})$ that maps probabilistic choice expressions to the probability of an erroneous execution. Aloe uses this specification to get the reliability of statements for precondition generation.

Statements. The small-step relation $\boxed{\langle s, \sigma, h\rangle \xrightarrow{\lambda,p}_{\psi} \langle s', \sigma', h'\rangle}$ defines the program evaluating in a stack frame σ, and heap h with the transition label λ. The semantics of Aloe follow from Rely. The new addition to our language is the try-check-recover block. Figure 4 defines the semantics for the try-check-recover block and the probabilistic choice statements. More detailed definitions and the remaining semantics are given in Appendix A [24].

Probabilistic Choice. The probabilistic choice statement $x = e_{orig}\ [r]\ e_{approx}$ will evaluate the expression from the original program (e_{orig}) with probability r, or otherwise produce the approximate result by evaluating the expression (e_{approx}). It can be used to model many approximate computations. For example, we can model an unreliable arithmetic instructions as `z = (x op y) [r] randVal()`, which models an instruction that can produce an error with probability $1 - r$. The probability can be provided as a variable whose concrete value is defined in the approximation model.

try-check-recover. try {S_1} check {e} recover {S_2} will evaluate the statement S_1. It will then evaluate the expression e to check if an error occurred in the evaluation of S_1. If such an error is detected, then S_2 will be evaluated (as the computation in S_1 is idempotent, the recovery code in S_2 can continue from the current state). This behavior is similar to that of `try-catch` statements commonly used in exception handling, except that S_1 is fully evaluated before the check is executed.

Function Calls. Aloe also supports function calls as expressions similar to Rely. Reliability specifications for functions can be provided in the function signature relating the reliability of the inputs to the output. For example the specification `int<0.99*`$\mathcal{R}(x, y)$`> f(int x, int y)` states that the reliability of the return values of `f` is $0.99 * \mathcal{R}(x, y)$. Function semantics are the same as those of Rely.

Reliability The semantics of reliability predicates is the same as in Rely. We present precise definitions of reliability in Appendix B [24].

4.2 Preprocessing

To simplify the presentation of the analysis, we perform multiple preprocessing steps before the reliability analysis.

Desugaring Recovery Mechanisms. The language provides several syntactic constructs to help write programs:

- **`redo[n]`.** If the recovery mechanism of a try-check-recover block is redo[1], we replace the `redo` in the recover block with the code in the try block. If the recovery mechanism is `redo`[n] for $n > 1$, we use nested try-check-recover blocks to generate code similar to the following:

```
                            [ try{ s1 }
                              check{e}
[ try{ s1 }                   recover{
  check{e}              ↦       try{ s1 }
  recover{ redo[n] } ]          check{e}
                                recover{ redo[n-1] }
                              } ]
```

S-Assign-Prob-True

$$\frac{}{\langle x = e_1\ [r]\ e_2,\ \sigma,\ h\rangle \xrightarrow{C,r}_{\psi} \langle x = e_1,\ \sigma,\ h\rangle}$$

S-Assign-Prob-False

$$\frac{}{\langle x = e_1\ [r]\ e_2,\ \sigma,\ h\rangle \xrightarrow{F,1-r}_{\psi} \langle x = e_2,\ \sigma,\ h\rangle}$$

S-Try

$$\frac{\langle \text{S1},\ \sigma,\ h\rangle \xrightarrow{\lambda,r}_{\psi} \langle \text{S1}',\ \sigma',\ h'\rangle}{\langle \text{try \{S1\} check \{e\} recover \{S2\}},\ \sigma,\ h\rangle \xrightarrow{\lambda,r}_{\psi} \langle \text{try \{S1'\} check \{e\} recover \{S2\}},\ \sigma',\ h'\rangle}$$

S-Check-1

$$\frac{\langle e,\ \sigma,\ h\rangle \xrightarrow{r}_{\psi} \langle e',\ \sigma,\ h\rangle}{\langle \text{try \{skip\} check \{e\} recover \{S2\}},\ \sigma,\ h\rangle \xrightarrow{C,r}_{\psi} \langle \text{try \{skip\} check \{e'\} recover \{S2\}},\ \sigma,\ h\rangle}$$

S-Check-True

$$\frac{}{\langle \text{try \{skip\} check \{true\} recover \{S2\}},\ \sigma,\ h\rangle \xrightarrow{C,1}_{\psi} \langle \text{skip},\ \sigma,\ h\rangle}$$

S-Check-False

$$\frac{}{\langle \text{try \{skip\} check \{false\} recover \{S2\}},\ \sigma,\ h\rangle \xrightarrow{C,1}_{\psi} \langle S2,\ \sigma,\ h\rangle}$$

Figure 4. Process-Level Dynamic Semantics of Statements (Selection)

S-Assign-Prob-Exact

$$\frac{}{\langle x = e_1\ [r]\ e_2,\ \sigma,\ h\rangle \xrightarrow{C,1}_{1_{\psi}} \langle x = e_1,\ \sigma,\ h\rangle}$$

S-TRY-Exact

$$\frac{}{\langle \text{try \{skip\} check \{e\} recover \{S2\}},\ \sigma,\ h\rangle \xrightarrow{C,1}_{1_{\psi}} \langle \text{skip},\ \sigma,\ h\rangle}$$

Figure 5. Exact Execution Semantics of Statements (Selection)

- **`redo[`ψ_2`]`.** Another common recovery mechanism is to execute the program on a different hardware with higher reliability. Then we use the new failure probabilities provided by the developer through the alternate approximation model ψ_2. We duplicate the code and replace the probabilities in the probabilistic choice statements using ψ_2.
- **Bounded Loops.** As in Rely, Aloe supports bounded loops, that it unrolls before the analysis. As the iteration number is statically known, the unrolled program is finite.

5 Reliability Analysis: Perfect Checker

Our reliability calculation extends that of Rely by adding additional rules for generating reliability preconditions for our try-check-recover mechanism. These rules are only applicable to postconditions whose joint reliability factor contains at least one variable updated within the try-check-recover block. All other postconditions are unaffected by the try-check-recover block, and become part of the generated precondition unmodified, as in [10].

Consider the following try-check-recover block:

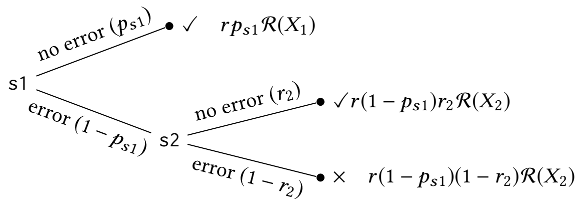
```
try{ s1; } check{ e } recover{ s2; }
```

For a try-check-recover block to satisfy a predicate Q after execution, it should be satisfied by all possible execution paths through the try-check-recover block.

5.1 Precondition Generation

Assume a perfect check. Figure 6 shows the tree of possible executions of the try-check-recover block. If the checker detects an error, then s2 is executed and the results of s1's execution are discarded. There are two possible execution paths through the try-check-recover block.

1. **s1 executes correctly and the checker passes:** In this case the check ensures that instructions in s1 do not degrade the reliability of any variables calculated in the try block. The reliability of these variables only depends on the reliability of values flowing into them in s1.
2. **At least one instruction in s1 executes incorrectly, the checker fails indicating an error, and s2 executes:** As we ensure that the computation in s1 is idempotent, the error in s1 does not affect the reliability of variables calculated in s2. Instead, it depends on the probability that statements in s2 update variables reliably and the reliability of values flowing into them in s2.



no error (p_{s1})
✓ $rp_{s1}\mathcal{R}(X_1)$
s1
error ($1-p_{s1}$)
s2
no error (r_2)
✓ $r(1-p_{s1})r_2\mathcal{R}(X_2)$
error ($1-r_2$)
× $r(1-p_{s1})(1-r_2)\mathcal{R}(X_2)$

Figure 6. Probabilities for Perfect Checkers

To handle these two scenarios we use and combine the preconditions generated independently for s1 and s2. Suppose the try-check-recover block must satisfy the postcondition $c \le r \cdot \mathcal{R}(X)$. Similar to the precondition generation steps in Rely, we need to replace $\mathcal{R}(X)$ in the postcondition with the *total* probability of reaching a state where the variables in X have the correct value after the try-check-recover block.

Case 1. Suppose that running Rely precondition generation on s1 results in the predicate $c \le r \cdot r_{s1} \cdot \mathcal{R}(X_{s1})$. Here, r_{s1} is the minimum probability that all instructions in s1 that affect variables in X execute correctly, and $\mathcal{R}(X_{s1})$ is the reliability of variables that flow into X in s1. However, the check only passes if *all* instructions in the try block execute correctly (as they may impact the checker) – not just the ones affecting variables in X. Instead of r_{s1}, the probability that X is calculated correctly *via case 1* depends on the probability that *all* instructions in s1 execute correctly, which we denote as p_{s1} (We discuss how to calculate p_{s1} in Section 5.2). Therefore, the total contribution of case 1 towards the probability that variables in X are calculated correctly is $p_{s1} \cdot \mathcal{R}(X_{s1})$.

Case 2. s1 executes incorrectly and fails the check with probability $1 - p_{s1}$. Suppose that running the Rely precondition generation algorithm on s2 results in the predicate $c \le r \cdot r_{s2} \cdot \mathcal{R}(X_{s2})$. Here, r_{s2} is the minimum probability that all instructions in s2 that affect variables in X execute correctly, and $\mathcal{R}(X_{s2})$ is the reliability of variables that flow into X in s2. However, case 2 only occurs if s1 fails. Therefore, the total contribution of case 2 towards the probability that variables in X are calculated correctly is $(1 - p_{s1}) \cdot r_{s2} \cdot \mathcal{R}(X_{s2})$.

Calculating $\mathcal{R}(X)$. We can now calculate the total probability that variables in X are calculated correctly as the sum of the probabilities of the two cases:

$$\mathcal{R}(X) = p_{s1} \cdot \mathcal{R}(X_{s1}) + (1 - p_{s1}) \cdot r_{s2} \cdot \mathcal{R}(X_{s2}),$$

We extended the reliability predicates with the addition operator, following the usual meaning of '+' since reliability factors denote probabilities over program states [10]. We can simplify this expression using the following ordering proposition from [10], which states that for two sets of variables A and B, if $B \subseteq A$ then $\mathcal{R}(A) \le \mathcal{R}(B)$. Therefore,

$$\mathcal{R}(X) \ge p_{s1} \cdot \mathcal{R}(X_{s1} \cup X_{s2}) + (1 - p_{s1}) \cdot r_{s2} \cdot \mathcal{R}(X_{s1} \cup X_{s2})$$
$$\mathcal{R}(X) \ge (p_{s1} + (1 - p_{s1})\, r_{s2}) \cdot \mathcal{R}(X_{s1} \cup X_{s2}).$$

Aloe evaluates the numerical expression in the brackets during analysis to simplify the constraint to the form $c \ge r' \cdot \mathcal{R}(X')$, since Rely's decision procedure can only solve constraints in this form. Following the subsumption rules in Proposition 2 of [10], we can replace $\mathcal{R}(X)$ with this probability in the postcondition to get the precondition:

$$\boxed{c \le r\,(p_{s1} + (1 - p_{s1})\, r_{s2}) \cdot \mathcal{R}(X_{s1} \cup X_{s2}).}$$

We present the proof of soundness in Appendix B [24]. Note that some variables in X may not be read or written to by either block. Such variables become part of $X_{s1} \cup X_{s2}$ unchanged, following Rely's precondition generation rules.

5.2 Minimum Success Probability of s1

The checker function ensures *complete* correctness of the execution of the try block. Recall, r_{s1} is the probability that s1 does not make any error that affects the variables tracked by the postcondition (X). We must separately calculate the probability p_{s1} that the try block executes without *any* error and therefore satisfies the check. For example, suppose the try block that contains these statements:

```
y = x [p0] 0; z = x [p1] 0;
```

When the postcondition only tracked the correctness of y, then r_{s1} only depends on p0. However, the checker also ensures that z is correct. Therefore, p_{s1}, the probability that the check passes, also depends on p1.

To compute the probability the try block executed correctly, we consider all possible control flow paths within the try block. A probabilistic choice statement x=a[p]b executes correctly with probability p. If a control flow path has multiple probabilistic choice statements, with correct execution probabilities $p_1, p_2, \ldots, p_n$, then the probability that that path as a whole executes correctly is $p_1 \cdot p_2, \cdot \ldots \cdot p_n$. Finally, p_{s1} is the minimum of the products over all paths.

5.3 Simplifying the Preconditions of s1 and s2

In Section 5.1, we assumed that the Rely precondition generation algorithm generated a simple precondition of the form $c \le r \cdot r_{s1} \cdot \mathcal{R}(X_{s1})$ for s1. In general, the Rely precondition generation algorithm may generate multiple precondition conjuncts from a single postcondition conjunct $c \le r \cdot \mathcal{R}(X)$. When generating preconditions for s1 (or s2) for our analysis, we can combine the preconditions into a single precondition that *subsumes* the original preconditions using Rely's subsumption rules (Proposition 2 of [10]).

Suppose we obtain the following precondition for s1:

$$c \le r_{11} \cdot \mathcal{R}(X_{11}) \land c \le r_{12} \cdot \mathcal{R}(X_{12}) \land \ldots c \le r_{1n} \cdot \mathcal{R}(X_{1n}).$$

Similarly suppose we obtain the following precondition for s2:

$$c \le r_{21} \cdot \mathcal{R}(X_{21}) \land c \le r_{22} \cdot \mathcal{R}(X_{22}) \land \ldots c \le r_{2n} \cdot \mathcal{R}(X_{2n})$$

Using the subsumption rule, we can replace s1's precondition with $c \le r \cdot r_{s1} \cdot \mathcal{R}(X_{s1})$ and s2's precondition with $c \le r \cdot r_{s2} \cdot \mathcal{R}(X_{s2})$, such that
$r_{s1} = \min(r_{11}, r_{12}, \ldots, r_{1n}), r_{s2} = \min(r_{21}, r_{22}, \ldots, r_{2n})$,
$X_{s1} = X_{11} \cup X_{12} \cup \ldots, X_{1n}$, and $X_{s2} = X_{21} \cup X_{22} \cup \ldots, X_{2n}$.

5.4 Full Example: Redo as Recovery

Simply re-doing the computation that experienced an error is a common recovery pattern. Below, the left side shows the analyzed code. Here, $p_1, p_2 \in [0, 1]$ and usually $p_1 \le p_2$. The right side shows Aloe's generated preconditions for the postcondition $0.99 \le \mathcal{R}(\{x\})$. The recovery block increases the reliability of x from $p_1 \cdot \mathcal{R}(\{y\})$ to $(p_1 + (1 - p_1)p_2) \cdot \mathcal{R}(\{y\})$.

```
try {
  x = y [p1] rand();
} check (f(x, y))
recover {
  x = y [p2] rand();
}
```

$\mapsto$

```
{0.99 <= (ps1 + (1 − ps1)p2)R({y})}
try { /* ps1 = p1 */
  {0.99 <= p1 * R({y})}
  x = y [p1] rand();
  {0.99 <= R({x})}
} check (f(x, y))
recover {
  {0.99 <= p2 * R({y})}
  x = y [p2] rand();
  {0.99 <= R({x})}
}
{0.99 <= R({x})}
```

Similarly Aloe's analysis shows that repeating the calculation in the try block on the same hardware at most n times upon detecting errors can increase the reliability of x to $(1 - (1 - p_1)^n) \cdot \mathcal{R}(\{y\})$. Such precise reliability calculations cannot be done with existing methods such as Rely.

Table 1. Checker Function Correctness Probabilities

	checker detects error	
	YES	NO
s1 experiences error	p_{TP}	p_{FN}
s1 does not experience error	p_{FP}	p_{TN}

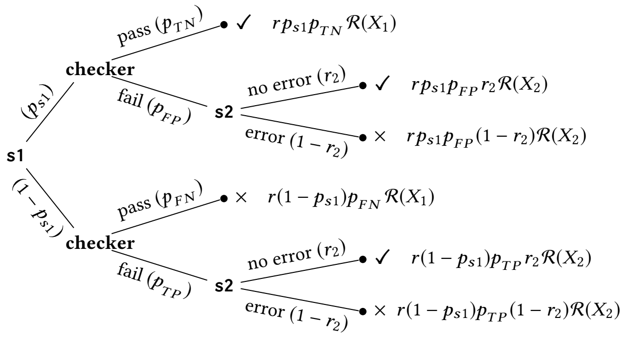

s1
(p_{s1})
checker
pass (p_{TN})
✓ $rp_{s1}p_{TN}\mathcal{R}(X_1)$
fail (p_{FP})
s2
no error (r_2)
✓ $rp_{s1}p_{FP}r_2\mathcal{R}(X_2)$
error $(1-r_2)$
× $rp_{s1}p_{FP}(1-r_2)\mathcal{R}(X_2)$
$(1-p_{s1})$
checker
pass (p_{FN})
× $r(1-p_{s1})p_{FN}\mathcal{R}(X_1)$
fail (p_{TP})
s2
no error (r_2)
✓ $r(1-p_{s1})p_{TP}r_2\mathcal{R}(X_2)$
error $(1-r_2)$
× $r(1-p_{s1})p_{TP}(1-r_2)\mathcal{R}(X_2)$

Figure 7. Probabilities for Imperfect Checkers

6 Reliability Analysis: Imperfect Checker

Aloe's reliability analysis can also be extended to checker functions that fail to capture all errors, or detect spurious errors. We specify imperfect checker functions through the common *probability of false positives* (p_{FP}) and the *probability of false negatives* (p_{FN}). We define them in Table 1, together with the probabilities of true positives (p_{TP}) and true negatives (p_{TN}). Recall that $p_{TP}+p_{FN}=1$ and $p_{FP}+p_{TN}=1$. These probabilities impact the reliability of the try-check-recover blocks. We next show how Aloe generates preconditions for try-check-recover statements that use imperfect checkers. Later, we discuss how we can express different checkers (together with additional assumptions) in this framework.

6.1 Precondition Generator

Consider the same program and constraint $c \leq r \cdot \mathcal{R}(X)$ as in Section 5. Figure 7 shows the tree of possible executions of the try-check-recover block for an imperfect checker. In this case, there are three possible execution paths through the try-check-recover block that result in a correct calculation of variables in X.

1. **s1 executes correctly and the checker passes:** This case is similar to case 1 for the perfect checker, however this time the check can still fail with probability p_{FP} due to the imperfect checker.
2. **s1 executes correctly but the checker fails, and s2 executes:** This case consists of situations where a error free execution is classified as having an error due to imprecisions in the checker functions. In this situation s2 is being executed even though s1 executed correctly.
3. **At least one instruction in s1 executes incorrectly, the checker fails and indicates an error, and s2 executes:** This case is also similar to case 2 for the perfect checker, however this time the check can still pass with probability p_{FN} due to the imperfect checker.

If any variable in X is updated in the try-check-recover statement, the execution path, where at least one instruction in s1 executes incorrectly, but the checker passes, always results in an incorrect calculation. As before, we start with the postcondition $c \leq r \cdot \mathcal{R}(X)$ and combine the preconditions generated independently for s1 and s2. Then we update all postconditions whose whose joint reliability predicate contains variables updated in the try-check-recover statement.

Case 1. Suppose that running Rely precondition generation on s1 results in the predicate $c \leq r \cdot r_{s1} \cdot \mathcal{R}(X_{s1})$. The statement s1 executes correctly and the check passes with probability $p_{s1} \cdot p_{TN}$. If these two events occur, then the probability that X is calculated correctly only depends on the reliability of variables flowing into variables in X that are updated in s1 ($\mathcal{R}(X_{s1})$). The probability that these two events occur *and* variables in X are calculated correctly in the first scenario is therefore $p_{s1} \cdot p_{TN} \cdot \mathcal{R}(X_{s1})$.

Case 2. Suppose that running Rely precondition generation on s2 results in the predicate $c \leq r \cdot r_{s2} \cdot \mathcal{R}(X_{s2})$. The statement s1 executes correctly but the check fails with probability $p_{s1} \cdot p_{FP}$. If these two events occur, then s2 is run, so the probability that X is calculated correctly is $r_{s2} \cdot \mathcal{R}(X_{s2})$. The probability that these two events occur *and* variables in X are calculated correctly in the second scenario is therefore $p_{s1} \cdot p_{FP} \cdot r_{s2} \cdot \mathcal{R}(X_{s2})$.

Case 3. s1 executes incorrectly and the check fails with probability $(1-p_{s1})p_{TP}$. If these two events occur, then s2 is run, so the probability that X is calculated correctly is $r_{s2} \cdot \mathcal{R}(X_{s2})$. The probability that these two events occur *and* variables in X are calculated correctly in the third scenario is therefore $(1-p_{s1}) \cdot p_{TP} \cdot r_{s2} \cdot \mathcal{R}(X_{s2})$.

Calculating $\mathcal{R}(X)$. We can now calculate the total probability that variables in X are calculated correctly as the sum of the probabilities of the three cases.

$$\begin{aligned}\mathcal{R}(X) &= p_{s1} \cdot p_{TN} \cdot \mathcal{R}(X_{s1}) + p_{s1} \cdot p_{FP} \cdot r_{s2} \cdot \mathcal{R}(X_{s2}) \\ &\quad + (1-p_{s1}) \cdot p_{TP} \cdot r_{s2} \cdot \mathcal{R}(X_{s2})\end{aligned}$$

Using the ordering preposition, we simplify this as follows:

$$\begin{aligned}\mathcal{R}(X) &\geq p_{s1} \cdot p_{TN} \cdot \mathcal{R}(X_{s1}, X_{s2}) + p_{s1} \cdot p_{FP} \cdot r_{s2} \cdot \mathcal{R}(X_{s1} \cup X_{s2}) \\ &\quad + (1-p_{s1}) \cdot p_{TP} \cdot r_{s2} \cdot \mathcal{R}(X_{s1}, X_{s2}) \\ \mathcal{R}(X) &\geq (p_{s1}p_{TN} + p_{s1}p_{FP}r_{s2} + (1-p_{s1})p_{TP}r_{s2}) \cdot \mathcal{R}(X_{s1} \cup X_{s2})\end{aligned}$$

Finally, by replacing $\mathcal{R}(X)$ with this probability in the postcondition, we get the precondition:

$$\boxed{c \leq r \cdot \Big(p_{s1}p_{TN} + p_{s1}p_{FP}r_{s2} + (1-p_{s1})\, p_{TP}r_{s2}\Big) \cdot \mathcal{R}(X_{s1} \cup X_{s2})}$$

6.2 Full Example: Imperfect Checkers

The following example shows the effect of an imperfect checker `f` on reliability precondition generation. We use the same naming convention as in Table 1.

```
try {
  x = y [p_1] rand();
} check (f(x, y))
recover {
  x = y [p_2] rand();
}
```

$\mapsto$

```
{0.99 <= (p_s1 * p_TN + p_s1 * p_FP * p_2
  +(1 - p_s1) * p_TP * p_2)R({y})}
try { /* p_s1 = p_1 */
  x = y [p_1] rand();
} check (f(x, y))
recover {
  x = y [p_2] rand();
}
{0.99 <= R({x})}
```

Our analysis shows that while the overall reliability of the computation is sensitive to the presence of imprecision of checker functions, using checkers still significantly improves the reliability over the unchecked computation.

6.3 Classes of Imperfect Checkers

We consider three main classes of imperfect checkers. The first class are those executed on unreliable hardware, just like the try and recover blocks. Thus they may also produce incorrect values during the check. If the checker function has the reliability specification `<r*R(Args)>`, then $p_{FN} = p_{FP} = r$. The second class of imperfect checkers, are randomized computations called *property checkers* [21] in theoretical computer science, which may identify all incorrect results ($p_{FN} = 0$), but may fail to identify a correct result ($p_{FP} > 0$). The third class of imperfect checkers use machine learning to infer a function `f'` that approximates the perfect checker `f` from data (e.g., outlier detection [1] or DNN checkers [30]). The probabilities $\widehat{p_{FP}}$ and $\widehat{p_{FN}}$ are estimated from training and validation data. But using these estimates in our equation requires care. For Rely and Aloe, Reliability is defined irrespective of the input (i.e., it is valid for *all* inputs), and randomness comes from errors in the computation. In contrast, $\widehat{p_{FP}}$ and $\widehat{p_{FN}}$ are estimated from the distribution of inputs during the training/validation of `f'`. The reliability predicates then only hold under the assumption that the input distribution remains the same between training/validation and production runs of the exact and unreliable programs.

7 Methodology

Benchmarks. To evaluate Aloe, we implemented a set of benchmarks from several application domains. These benchmarks can tolerate error in the output and have been studied in the approximate computing literature:

- *PageRank:* Computes PageRank for nodes in a graph [37]. We verify the reliability of one iteration of the PageRank kernel for one node.
- *Scale:* Computes a bigger version of an image. We verify the reliability of calculating one pixel of the output image.
- *Blackscholes:* Computes the prices of a portfolio of stock options. We verify the reliability of computing an option price.
- *SSSP:* Computes Single Source Shortest Path in a graph. We verify the reliability of one iteration of the SSSP kernel for one node.
- *BFS:* Breadth first search in a graph. We verify the reliability of one iteration of the search kernel for one node.
- *SOR:* A kernel for computing successive over-relaxation (SOR). We verify the reliability of one iteration of the SOR kernel for one element of the 2D array.
- *Motion:* A pixel-block search algorithm from the x264 video encoder. We verify that the reliability of computing the sum of squared differences (SSD) for one candidate block.
- *Sobel:* Sobel edge-detection filter calculation. We verify the reliability of calculating one pixel of the output image.

Table 2. Benchmarks Details

		LoC	LoC	LoC
Benchmark	**Source**	**(kernel)**	**(total)**	**(unrolled)**
PageRank	CRONO [2]	20	45	720K
Scale	Chisel [33]	57	79	561K
Blackscholes	Chisel [33]	81	97	270K
SSSP	CRONO [2]	22	31	800K
BFS	CRONO [2]	14	30	720K
SOR	Chisel [33]	24	37	1500K
Motion	Rely [10]	14	32	150K
Sobel	AxBench [46]	34	45	160K

Table 2 presents benchmark statistics, including the benchmark suite we derived the code from (Column 2), the size of the computation kernel (Column 3), the full benchmark (Column 4), and after unrolling loops (Column 5). These lines exclude setup and I/O code. The inputs we used for the experiments are listed in Appendix C [24].

Unreliable Operations. We used the probabilistic choice statements to simulate two different unreliable architectures where arithmetic operations can fail and produce an incorrect output with probability 10^{-3} and 10^{-4} respectively. That is, the reliability of arithmetic operations in the two architectures is 0.999 and 0.9999 respectively. We executed each benchmark with a runtime library that would randomly replace the result of arithmetic operations with 0 (as an error value) based on the error probability of the architecture.

Specifications and Checkers. Each benchmark has at least one try-check-recover block. The try block executes on the architecture where arithmetic operators have reliability 0.999. The recover block executes the same code on the more reliable architecture where arithmetic operators have reliability 0.9999. For perfect checkers, we assumed a hardware technique with support for detecting errors [13, 32, 35]. For imperfect checkers, we use multiple false-positive and false-negative values in the range of those in Topaz [1].

Table 3. Verified Kernel and End-to-End Reliability Postconditions

Benchmarks	Kernel-Level			End-to-End		
	Postcondition	Aloe	Rely	Postcondition	Aloe	Rely
PageRank	$0.9999 \le \mathcal{R}(\text{newPagerank})$	✓(3ms)	×(3ms)	$0.99 \le \mathcal{R}(\text{PageRank})$	✓(23.33s)	×(19.73s)
Scale	$0.9999 \le \mathcal{R}(\text{newPixel})$	✓(2ms)	×(2ms)	$0.99 \le \mathcal{R}(\text{ImageOut})$	✓(10.48s)	×(8.79s)
Blackscholes	$0.9999 \le \mathcal{R}(\text{optionPrice})$	✓(3ms)	×(2ms)	$0.99 \le \mathcal{R}(\text{Prices})$	✓(6.51s)	×(5.60s)
SSSP	$0.9999 \le \mathcal{R}(\text{dist})$	✓(3ms)	×(3ms)	$0.99 \le \mathcal{R}(\text{Distances})$	✓(18.60s)	×(18.25s)
BFS	$0.9999 \le \mathcal{R}(\text{visited})$	✓(3ms)	×(4ms)	$0.99 \le \mathcal{R}(\text{Visited})$	✓(15.22s)	×(15.14s)
SOR	$0.9999 \le \mathcal{R}(\text{result})$	✓(1ms)	×(1ms)	$0.99 \le \mathcal{R}(\text{ArrayOut})$	✓(21.02s)	×(17.90s)
Motion	$0.9999 \le \mathcal{R}(\text{MinSSD})$	✓(45ms)	×(45ms)	$0.99 \le \mathcal{R}(\text{MinSSD})$	✓(4.42s)	×(4.19s)
Sobel	$0.9999 \le \mathcal{R}(\text{ImageOut})$	✓(1ms)	×(1ms)	$0.99 \le \mathcal{R}(\text{ImageOut})$	✓(2.10s)	×(1.80s)

Environment. We ran all experiments on a computer with a Intel Xeon 3.6GHz CPU with 32 GB RAM that was running Ubuntu 18.04. We implemented the analysis using ANTLR for parsing and Python as the backend. For empirical evaluation, we compiled our benchmarks to the Go language and added instrumentation to track the number of errors and the end-to-end error magnitude.

8 Evaluation

8.1 Static Reliability Analysis: Perfect Checkers

Table 3 shows the reliability postcondition we verified for each benchmark in the presence of a perfect checker. Column 1 is the benchmark, Column 2 is the verified reliability postcondition for the benchmark kernel, and Column 3 indicates if the analysis is able to verify the bound (✓) or not (×) along with the runtime for the analysis. Column 4 shows if the same bound can be verified through a naive Rely analysis that treats the try-check-recover statement as a if-then-else statement as discussed in Section 2.2, and the runtime for that analysis. Columns 5, 6, and 7 show the results of the same analysis for the end-to-end reliability of the benchmark.

The results show that our analysis can verify the reliability of recovery mechanisms better than the existing analyses. Aloe can verify all kernel bounds, while the naive Rely approach fails to compute a satisfactory bound. This is because when treating the try-check-recover block as a if-then-else statement, Rely considers the lower reliability of the two branches as the reliability of the entire statement. In contrast, when using our approach with a perfect checker, the reliability of the try-check-recover block is *greater* than the reliability of the try and recover blocks in isolation, since our analysis takes into account the fact that both try and recover block need to fail together for an error to be introduced.

For most kernels, our approach performs the analysis in less than 3 milliseconds. The Motion kernel executes a computation for a large number of iterations leading to increased analysis time after unrolling. For end-to-end reliability, Aloe's analysis was able to prove surprisingly strong bounds on reliability for all benchmarks. Rely once again failed to satisfy these postconditions, for the same reason as above. In all cases, our analysis takes less than 25 seconds.

Table 4. Maximum Verifiable Reliability Postconditions for kernels with an Imperfect Checker

Benchmarks	Perfect	Imperfect			
	TP:1.0	TP:0.95	TP:1.0	TP:0.9	TP:0.8
	TN:1.0	TN:0.95	TN:0.95	TN:0.9	TN:0.8
PageRank	0.9999	0.9982	0.9968	0.9964	0.9375
Scale	0.9999	0.9993	0.9999	0.9987	0.9976
Blackscholes	0.9999	0.9992	0.9999	0.9985	0.9971
SSSP	0.9999	0.9995	0.9999	0.9991	0.9982
BFS	0.9999	0.9959	0.9999	0.9994	0.9839
SOR	0.9999	0.9997	0.9999	0.9994	0.9989
Motion	0.9999	0.9912	0.9918	0.8385	0.7031
Sobel	0.9999	0.9995	0.9999	0.9991	0.9982

8.2 Static Reliability Analysis: Imperfect Checkers

Table 4 shows the results of attempting to verify the same reliability postconditions for kernels as in Table 3 for each benchmark. Column 1 shows the benchmark and Column 2 shows the highest reliability that can be verified for a perfect checker. The next columns show the highest reliability that can be verified for an imperfect checker with a particular true positive (TP) rate and true negative (TN) rate. For all kernels except Motion, the verifiable postcondition reliability remains above 0.99 when $p_{\text{TP}}, p_{\text{TN}} \ge 0.9$, but degrades due to the possibility of false classifications in the checker. The reliability of the Motion kernel degrades faster as it executes a large number of iterations that compound the unreliability incurred through the imperfect checker.

8.3 Empirical Results

We empirically confirmed the results of our static analysis of end-to-end programs for a perfect checker. We ran each verified program 2000 times and calculated the average output error due to unreliable operations and the fraction of runs where the output was different from a completely reliable execution (fail rate). 2000 runs allows us to verify that the empirically calculated fail-rate is less than 0.01 (the

Table 5. Empirical Reliability Results

Benchmarks	**Error Metric**	**Perfect Checker Error**	**Fail%**	**No Recovery Error**
PageRank	ℓ_2	1.54×10^{-5}	0.7%	4.04×10^{-4}
Scale	PSNR	56.023 dB	0.55%	38.280 dB
Blackscholes	ℓ_2	6.26×10^{-8}	0.15%	4.80×10^{-5}
SSSP	ℓ_2	2.82×10^{-3}	0.1%	4.27×10^{-3}
BFS	ℓ_2	0	0.15%	4.25×10^{-5}
SOR	ℓ_2	9.25×10^{-5}	0.75%	1.00×10^{-3}
Motion	SSD	0	0.15%	9.85×10^{-4}
Sobel	ℓ_2	2.63×10^{-5}	0.95%	2.69×10^{-5}

verified bound) using a one-sided binomial test ($p_0 = 0.99$, $p_1 = 0.9837$, $\alpha = 0.05$, $\beta = 0.2$).

Table 5 gives the result of the empirical evaluation for each program. Column 1 shows the benchmark. Column 2 shows the error metric we used. We used error metrics that were used in prior work in the area – for Motion, it is the relative error in the calculated minimum SSD, for Scale, it is the PSNR of the unreliable output, and for the other benchmarks it is the ℓ_2 norm of the output vector. Columns 3 and 4 show the error and the fail rate (percentage of runs with incorrect results) for a perfect checker. Column 5 shows the error when the program does not use a recovery mechanism. The error values are the average over the runs in which a failure occurred (as most runs execute correctly).

The results show that the empirically calculated reliability (1 – fail rate) is always within the bounds verified by Aloe for a perfect checker. Further, when these benchmarks *do* fail, the error in the final output is small. This shows the amenability of these benchmarks to approximations. For BFS and Motion, the program automatically corrected errors that occurred, without any intervention.

The results also show how the error increased, in some cases significantly, when the program did not have a recovery mechanism. Further, in all benchmarks, the failure rate also exceeded the 1% limit.

9 Related Work

Approximate Program Analyses. Many static analyses have been proposed in the recent years for analyzing approximate or unreliable computations [5, 6, 8–10, 20, 33, 38, 42]. All existing analyzes suffer from imprecision when analyzing computations with recovery mechanisms. Our analysis extends Rely by adding additional precondition generation rules for recovery mechanisms which generate less conservative preconditions compared to Rely.

Error Detection. Hardware error detectors [13, 17, 19, 32] typically consist of special circuits within the processors and memory units which can detect if an error has occurred. The accuracy of these detectors is limited by circuit size and energy requirements [32]. Another approach is to run the same computation on multiple processors at the same time [35, 44] and report an error if the computations disagree. However this requires a significant amount of redundant computation. Mahmoud et al. [30] propose augmenting complex calculations with small neural networks that analyze the input and output to approximately determine if an error occurred. Achour and Rinard [1] use outlier detection by constructing feature vectors from inputs and outputs. Our approach is agnostic of the nature of the error detection mechanism. We expose the checker interface (via the probabilities of false positives and false negatives) and show how to incorporate both perfect and imperfect checkers in the analysis.

Recovery Mechanisms. Many systems deal with hardware failures using the checkpoint/restore method [29], which periodically saves the program state to reliable memory and restores the program state should an error occur. Languages such as Relax [13] and Topaz [1] expose recovery mechanisms to the developer. Relax allows retrying the unreliable computation (on the same unreliable hardware) as well as discarding incorrect calculations. Topaz instead opts for reexecuting the computation on a perfectly reliable hardware or droppoing tasks (following [40]). Several approaches in approximate computing provide implicit recovery from inaccuracy by dynamically adapting the approximation to the input properties [3, 25, 34, 45]. Containment Domains [12] provide programming constructs to define various software error detection and recovery mechanisms. None of these approaches provide a static analysis of reliability. Aloe is the first static analysis of reliability for programs with recovery mechanisms.

10 Conclusion

This work presented Aloe, a quantitative static analysis of reliability of programs with recovery blocks. The Aloe analysis supports reasoning about both perfect and imperfect detection of errors, as well as reliable and unreliable recovery code. It implemented a novel precondition generator for recover blocks, built on top of Rely's analysis. We implemented Aloe and applied it to a set of eight programs previously used in approximate computing research. Our results show that Aloe can verify significantly higher reliability conditions compared to the existing Rely analysis. Moreover, the end-to-end accuracy of the verified computations exhibits only small accuracy losses.

Acklowledgements

We thank our shepherd Fabrice Rastello and the anonymous reviewers for useful suggestions. The research presented in this paper was supported in part by NSF Grants CCF-1629431, CCF-1703637, and CCF-1846354, and DARPA DSSOC program, part of the Electronics Resurgence Initiative (ERI), under Contract No. HR0011-18-C-0122.

References

[1] S. Achour and M. Rinard. 2015. Energy Efficient Approximate Computation with Topaz. In *OOPSLA*.

[2] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. 2015. CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores. In *IISWC*.

[3] W. Baek and T. M. Chilimbi. 2010. Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation. In *PLDI*.

[4] S. Borkar. 2005. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro* 25, 6 (2005).

[5] B. Boston, Z. Gong, and M. Carbin. 2018. Leto: verifying application-specific hardware fault tolerance with programmable execution models. In *OOPSLA*.

[6] B. Boston, A. Sampson, D. Grossman, and L. Ceze. 2015. Probability type inference for flexible approximate programming. In *OOPSLA*.

[7] F. Cappello, G. Al, W. Gropp, S. Kale, B. Kramer, and M. Snir. 2014. Toward Exascale Resilience: 2014 Update. *Supercomput. Front. Innov.: Int. J.* 1, 1 (April 2014).

[8] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. 2012. Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs. In *PLDI*.

[9] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. 2013. Verified integrity properties for safe approximate program transformations. In *PEPM*.

[10] M. Carbin, S. Misailovic, and M. C. Rinard. 2013. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *OOPSLA*.

[11] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra. 2016. CLEAR: Cross-Layer Exploration for Architecting Resilience - Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores. In *DAC*.

[12] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. 2012. Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems. In *SC*.

[13] M. de Kruijf, S. Nomura, and K. Sankaralingam. 2010. Relax: an architectural framework for software recovery of hardware faults. In *ISCA*.

[14] M. A. de Kruijf, K. Sankaralingam, and S. Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In *PLDI*.

[15] N. DeBardeleben, J. Laros, J. T. Daly, S. L. Scott, C. Engelmann, and B. Harrod. 2009. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. *Whitepaper* (2009).

[16] M. Dimitrov and H. Zhou. 2009. Anomaly-based Bug Prediction, Isolation, and Validation: An Automated Approach for Software Debugging. In *ASPLOS*.

[17] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. 2003. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO*.

[18] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. 2010. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *ASPLOS*.

[19] V. Fernando, A. Franques, S. Abadal, S. Misailovic, and J. Torrellas. 2019. Replica: A Wireless Manycore for Communication-Intensive and Approximate Data. In *ASPLOS*.

[20] V. Fernando, K. Joshi, and S. Misailovic. 2019. Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization. In *OOPSLA*.

[21] O. Goldreich. 2017. *Introduction to property testing*. Cambridge University Press.

[22] S. Hari, S. Adve, and H. Naeimi. 2012. Low-cost Program-level Detectors for Reducing Silent Data Corruptions. In *DSN*.

[23] S. He, S. K. Lahiri, and Z. Rakamarić. 2018. Verifying relative safety, accuracy, and termination for program approximations. *Journal of Automated Reasoning* 60, 1 (2018).

[24] K. Joshi, V. Fernando, and S. Misailovic. 2020. Appendix to Aloe. https://kpjoshi.com/papers/CGO2020_appendix.pdf

[25] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang. 2016. Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation. In *PLDI*.

[26] G. Li, S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. Keckler. 2017. Understanding Error Propagation in Deep-Learning Neural Networks (DNN) Accelerators and Applications. In *SC*.

[27] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. 2011. Flikker: Saving DRAM Refresh-Power through Critical Data Partitioning. In *ASPLOS*.

[28] G. Lyle, S. Cheny, K. Pattabiraman, Z. Kalbarczyk, and R. Iyer. 2009. An End-to-end Approach for the Automatic Derivation of Application-Aware Error Detectors. In *DSN*.

[29] M. R. Lyu. 1995. *Software Fault Tolerance*. John Wiley & Sons.

[30] A. Mahmoud, P. Reckampm, P. Tang, C. Fletcher, and S. Adve. 2019. Approximate Checkers. In *WAX*.

[31] A. Mahmoud, R. Venkatagiri, K. Ahmed, S. Misailovic, D. Marinov, C. W. Fletcher, and S. Adve. 2019. Minotaur: Adapting Software Testing Techniques for Hardware Errors. In *ASPLOS*.

[32] A. Meixner, M. E. Bauer, and D. Sorin. 2007. Argus: Low-cost, comprehensive error detection in simple cores. In *MICRO*.

[33] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. 2014. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In *OOPSLA*.

[34] S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi. 2017. Phase-aware optimization in approximate computing. In *CGO*.

[35] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. 2002. Detailed design and evaluation of redundant multi-threading alternatives. In *ISCA*.

[36] R. P. 2013. European exascale software initiative EESI2 - towards exascale roadmap implementation. *2nd IS-ENES workshop on high-performance computing for climate models* (2013).

[37] L. Page, S. Brin, R. Motwani, and T. Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report.

[38] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris. 2015. FLEXJAVA: Language support for safe and modular approximate programming. In *FSE*.

[39] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer. 2008. SymPLFIED: Symbolic program-level fault injection and error detection framework. In *DSN*.

[40] M. Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS*.

[41] S. Sahoo, M. Li, P. Ramchandran, S. Adve, V. Adve, and Y. Zhou. 2008. Using Likely Program Invariants to Detect Hardware Errors. In *DSN*.

[42] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*.

[43] N. Saxena. 2016. Autonomous Car is the New Driver for Resilent Computing and Design-for-test. In *NASA Electronic Parts and Packaging (NEPP) Program 2016 Electronics Technology Workshop*.

[44] J. R. Sklaroff. 1976. Redundancy Management Technique for Space Shuttle Computers. *IBM J. Res. Dev.* 20 (1976).

[45] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, S. Misailovic, and S. Bagchi. 2018. Videochef: efficient approximation for streaming video processing pipelines. In *USENIX ATC*.

[46] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. 2017. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design Test* 34, 2 (2017).