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

Keyur Joshi, Vimuth Fernando, and Sasa Misailovic
University of Illinois at Urbana-Champaign

CGO 2020
Unreliable Hardware – Transient Errors

Architects make great efforts to minimize errors

Some errors slip through the cracks – silently corrupt computation results

Process size vs. error rate

Image from “Inter-Agency Workshop on HPC Resilience at Extreme Scale”, DoD, ‘12
Big systems fail due to scale

Heterogeneous systems have components with varying reliability

Small systems fail due to low voltage/power

Rugged environments radiation, temperature, etc.

Transient Errors are Everywhere

Images from Wikipedia and publicdomainvectors.org
Reliability is the probability of obtaining the *exact* answer.
100% Exactness Is Not Always Required!

Media Processing

Approximations for NP-Complete Problems

Machine Learning

Large-Scale Graph Processing

Images from Wikipedia
But We **Do** Need Quality Control...
How do we increase reliability of programs on unreliable hardware?
Lightweight Check and Recover

\[ z = x \times y \]
\[ z' = x \times y \]
\[ z == z' ? \]

Code Re-Execution (SWIFT, DRIFT, Shoestring)

\[ y = \text{foo}(x) \]
\[ \text{DNN}(x, y) ? \]

Anomaly Detection (Topaz, Rumba)

\[ y = \text{foo}(x) \]
\[ \text{hw_err_flg} ? \]

Hardware Error Flag (Relax)

\[ s = \text{SAT}(p) \]
\[ \text{verify}(s, p) ? \]

Verification (NP-Complete)
The Try-Check-Recover Mechanism

Some research languages\textsuperscript{1,2} expose \textit{Try-Check-Recover mechanisms}:

\begin{verbatim}
try { solution = SATSolve(problem) }
\end{verbatim} \textit{Unreliable code}

\begin{verbatim}
check { satisfies(problem, solution) }
\end{verbatim} \textit{Checks for errors}

\begin{verbatim}
recover { solution = SATSolve(problem) }
\end{verbatim} \textit{Recovery code}

\textsuperscript{1}“Relax”, M. de Kruijf, S. Nomura, and K. Sankaralingam, ISCA ’10 \hspace{1cm} \textsuperscript{2}“Topaz”, S. Achour and M. Rinard, OOPSLA ‘15
How do we analyze programs to ensure that they are sufficiently reliable?
Static Reliability Analysis of Programs\textsuperscript{1}

output = program(input)

Prove:
\( \{R(output) \geq 0.99 \cdot R(input)\} \)

\textsuperscript{1}“Rely”, M. Carbin, S. Misailovic, and M. Rinard, OOPSLA ‘13

Does not contain try-check-recover
How do we do reliability analysis of programs with checks and recovery mechanisms in a formal manner?
Aloe

The first static reliability analysis of programs with recover blocks

Supports recovery blocks that re-execute the `try` computation

Supports arrays, conditionals, and bounded loops

Supports various types of error checkers
Aloe Syntax

\[
\begin{align*}
n &\in \mathbb{N} & \text{quantities} & \text{recovery} &\rightarrow \\
m &\in \mathbb{N} \cup \mathbb{F} & \text{values} & \text{redo}[n] &\rightarrow \\
r &\in [0, 1.0] & \text{probability} & \text{redo} &\rightarrow \\
x, b &\in \text{Var} & \text{variables} & \psi &\rightarrow \\
a &\in \text{ArrVar} & \text{array variables} & S &\rightarrow \\
f &\in \text{Func} & \text{external functions} & S &\rightarrow \\
op &\in \{+,-,\ldots\} & \text{arithmetic operators} & \text{skip} &\rightarrow \\
\text{Exp} &\rightarrow m \mid x \mid f(\text{Exp}^*) \mid \text{Exp} \mid \text{Exp op Exp} & \text{expressions} & x = \text{Exp} &\rightarrow \\
\text{Exp}^* &\rightarrow (\text{Exp}) \mid \text{Exp op Exp} & & x = \text{Exp} [] r \text{ Exp} &\rightarrow \\
\text{Exp}^+ &\rightarrow a[\text{Exp}^+] \mid a[\text{Exp}^+] = \text{Exp} & & x = a[\text{Exp}^+] &\rightarrow \\
\text{int<n>} \mid \text{float<n>} &\rightarrow \text{basic types} & & \text{if } \text{Exp} \{S\} \text{ else } \{S\} &\rightarrow \\
\text{tx} \mid \text{ta}[n^+] \mid \text{D;}\text{D} &\rightarrow \text{variable} & & \text{repeat } n \{S\} &\rightarrow \\
\text{D;}\text{D} &\rightarrow \text{declarations} & & x = (\text{T})\text{Exp} &\rightarrow \\
P &\rightarrow \text{D;}\text{S} & \text{program} & \text{try } \{S\} \text{ check } \{\text{Exp}\} \text{ recover } \{\text{recovery}\} &\rightarrow \\
\end{align*}
\]
Modelling Unreliable Computations

Aloe models unreliable computations using probabilistic choice:

\[ \text{var} = \begin{cases} 
\text{e\_exact} & \text{with probability } p \\
\text{e\_inexact} & \text{with probability } 1-p 
\end{cases} \]

\[ z = x+y \quad \text{[p]} \quad \text{rnd}() \quad // \text{instruction level}^1 \]
\[ z = \text{foo}(x) \quad \text{[p]} \quad \text{foo\_err}(x) \quad // \text{function level}^2 \]
\[ z = 1.0 \quad \text{[p]} \quad \text{rnd}() \quad // \text{unreliable memory operations}^3 \]

^1“EnerJ”, A. Sampson et al., PLDI ’11
^2“Rumba”, D. Khudia et al., ISCA ’15
^3“Replica”, V. Fernando et al., ASPLOS ’19
## Hardware Specifications (Example)

<table>
<thead>
<tr>
<th>Specification</th>
<th>Mild</th>
<th>Medium</th>
<th>Aggressive</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM refresh: per-second bit flip probability</td>
<td>$10^{-9}$</td>
<td>$10^{-5}$</td>
<td>$10^{-3}$</td>
</tr>
<tr>
<td>Memory power saved</td>
<td>17%</td>
<td>22%</td>
<td>24%</td>
</tr>
<tr>
<td>SRAM read upset probability</td>
<td>$10^{-16.7}$</td>
<td>$10^{-7.4}$</td>
<td>$10^{-3}$</td>
</tr>
<tr>
<td>SRAM write failure probability</td>
<td>$10^{-5.59}$</td>
<td>$10^{-4.94}$</td>
<td>$10^{-3}$</td>
</tr>
<tr>
<td>Supply power saved</td>
<td>70%</td>
<td>80%</td>
<td>90%*</td>
</tr>
<tr>
<td>float mantissa bits</td>
<td>16</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>double mantissa bits</td>
<td>32</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Energy saved per operation</td>
<td>32%</td>
<td>78%</td>
<td>85%*</td>
</tr>
<tr>
<td>Arithmetic timing error probability</td>
<td>$10^{-6}$</td>
<td>$10^{-4}$</td>
<td>$10^{-2}$</td>
</tr>
<tr>
<td>Energy saved per operation</td>
<td>12%*</td>
<td>22%</td>
<td>30%</td>
</tr>
</tbody>
</table>

**Table 2.** Approximation strategies simulated in our evaluation. Numbers marked with * are educated guesses by the authors; the others are taken from the sources described in Section 4.2. Note that all values for the Medium level are taken from the literature.

---

1“EnerJ”, A. Sampson et al., PLDI ‘11
Aloe Reliability Analysis

Aloe’s analysis is based on that of Rely\(^1\)

\(\{0.999 \cdot \mathcal{R}(x, y) \geq 0.99\}\)

\(z = x \cdot y \cdot 0.999 \text{ rnd}();\)

\(\{\mathcal{R}(z) \geq 0.99\}\)

\(^1\)M. Carbin, S. Misailovic, and M. Rinard, OOPSLA ‘13
Example – Sorting on Unreliable Hardware

```java
try {
    output = quicksort(arr) \[p_{\text{try}}\] scramble(arr);
}

cHECK { sorted(output) }

recover {
    output = quicksort(arr) \[p_{\text{rec}}\] scramble(arr);
}
```

We want output to be correctly sorted with probability $\geq r$
Possible Execution Paths

```
output = quicksort(arr)
try
    output = quicksort(arr) [p_{try}]
    scramble(arr);
    recover
    check fail
    error
    (1 - p_{try}) \cdot (1 - p_{rec})
    try + check
    correct
    check pass
    (1 - p_{try}) \cdot p_{rec}
```
Aloe Precondition Generation

```plaintext
try {
    output = quicksort(arr) [p_{try}] scramble(arr);
}
check { sorted(output) }
recover {
    output = quicksort(arr) [p_{rec}] scramble(arr);
    {R(output) \geq r}
}
{R(output) \geq r}
```
Detour – Error-Free Rate of **try**

```plaintext
try {
    \{0.99 \cdot \mathcal{R}(w,y) \geq r\}
    x = y \times 2 \ [0.99] \text{ rnd}();
    z = w + y \ [0.99] \text{ rnd}();
    \{\mathcal{R}(z) \geq r\}
}
check \{ f(w,x,y,z) \}
```

correctness

**check** detects errors in any part of **try**

Unreliable computation of **x** affects the probability that **check** passes!

Aloe separately analyses the probability that **try** executes correctly *in its entirety*
Aloe Precondition Generation

\[\{(p_{\text{try}} + (1 - p_{\text{try}}) \cdot p_{\text{rec}}) \cdot \mathcal{R}(\text{arr}) \geq r\}\]

```plaintext
try {
    output = quicksort(arr)[p_{\text{try}}] scramble(arr);
}
check { sorted(output) }
recover {
    \{p_{\text{rec}} \cdot \mathcal{R}(\text{arr}) \geq r\}
    output = quicksort(arr)[p_{\text{rec}}] scramble(arr);
    \{\mathcal{R}(\text{output}) \geq r\}
}
\{\mathcal{R}(\text{output}) \geq r\}
```

Error-free rate of \text{try}: \(p_{\text{try}}\)
Possible Execution Paths ($p_{try} = p_{rec} = 0.99$)

output = quicksort(arr)
[0.99]
scramble(arr); try +
check

correct check pass

correct

0.99

0.01 \cdot 0.99

error check fail

Aloe calculates total probability of correct output: $0.99 + 0.0099 = 0.9999$
Combining Preconditions

classification { recover { if (*) { x = y * w [0.99] rnd(); } else { x = y + z [0.999] rnd(); } } {R(x) ≥ r} }
Complex Postconditions

\[
\{0.9999 \cdot p_1 \cdot \mathcal{R}(y, z) \geq r_1 \} \land \{p_2 \cdot \mathcal{R}(y) \geq r_2 \}
\]

```java
try {
    x = y * z [0.99] rnd();
}
check { f(x, y, z) }
recover {
    x = y * z [0.99] rnd();
}
```

\[
\{ p_1 \cdot \mathcal{R}(x) \geq r_1 \} \land \{ p_2 \cdot \mathcal{R}(y) \geq r_2 \}
\]
Aloe Assumptions – Re-execution

Aloe expects that `recover` re-executes the code in `try`

The reliability of statements in `try` and `recover` may differ

Why? Impossible to prove using Rely’s logic that `try` and `recover` perform the same computation

If such a proof is already available, then Aloe’s analysis remains valid even for syntactically distinct `try` and `recover`
Aloe Assumptions – Idempotence

Aloe expects that the computation in `try` is *idempotent*

Idempotent – can be run multiple times without changing the correct result

E.g. \[ x = y + z \quad \checkmark \quad x = x + z \quad \times \]

Why? Otherwise `try` can modify the result of executing `recover`
Handling Control Flow – Same as in Rely

\[ RP_\psi(\text{if}_\ell \ell \ s_1 \ s_2, Q) = RP_\psi(s_1, Q) \land RP_\psi(s_2, Q) \]

\[ RP_\psi(\text{while}_\ell \ b: 0 \ s, Q) = Q \]
\[ RP_\psi(\text{while}_\ell \ b: n \ s, Q) = RP_\psi(\mathcal{T}(\text{if}_{\ell_n} \ b \{ s ; \text{while}_\ell \ b: (n - 1) \ s \} \text{skip}), Q) \]

Rely Precondition Generation for Control Flow
Using If-Then for Recovery Mechanisms

Prior analyses (Rely) expressed recovery mechanisms using if-then statements

```python
output = quicksort(list) [p_{\text{try}}] scramble(list);
if (! sorted(output) )
{
    output = quicksort(list) [p_{\text{rec}}] scramble(list);
}
```
Using If-Then for Recovery Mechanisms

Rely treats if-then as a nondeterministic choice

Case 1:

\[
output = \text{quicksort(list)} \ [p_{\text{try}]} \ \text{scramble(list)};
\]

Case 2:

\[
output = \text{quicksort(list)} \ [p_{\text{try}]} \ \text{scramble(list)};
\]

\[
output = \text{quicksort(list)} \ [p_{\text{rec}]} \ \text{scramble(list)};
\]
Using If-Then for Recovery Mechanisms

Rely analyses the reliability of each case separately

Case 1: output sorted correctly with probability $p_{try}$
\[
\text{output} = \text{quicksort}(\text{list}) [p_{try}] \text{ scramble}(\text{list});
\]

Case 2: output sorted correctly with probability $p_{rec}$
\[
\text{output} = \text{quicksort}(\text{list}) [p_{try}] \text{ scramble}(\text{list});
\]
\[
\text{output} = \text{quicksort}(\text{list}) [p_{rec}] \text{ scramble}(\text{list});
\]
Using If-Then for Recovery Mechanisms

Rely then retains the most conservative case

Overall reliability: \( \min(p_{\text{try}}, p_{\text{rec}}) \)

Compare to Aloe’s calculated reliability using try-check-recover:

\[
p_{\text{try}} + (1 - p_{\text{try}}) \cdot p_{\text{rec}}
\]
Imperfect Checkers

Many checkers are imperfect – may not precisely detect errors

Code re-execution and comparison
• “SWIFT”, G. Reis et al., CGO ‘05
• “Shoestring”, S. Feng et al., ASPLOS ’10

Error Prediction
• “Rumba”, D. Khudia et al., ISCA ‘15

Anomaly detection
• “Topaz”, S. Achour and M. Rinard, OOPSLA ‘15

May detect nonexistent errors

May not detect actual errors, may detect nonexistent errors
False Positives / False Negatives

- **try** executes
  - No Error: Check executes, Check Pass
  - Error: Check executes, Check Fail

- Probabilities Provided to Aloe
  - True Negative
  - False Positive
  - False Negative
  - True Positive
False Positive / False Negative Rates

For some checkers, these rates can be determined analytically
• E.g. approximate sorted-ness checks provide statistical guarantees

For other checkers, these rates must be determined empirically
• E.g. outlier detection¹, DNNs² which require pre-training
• Probabilities of false positives/negatives are estimated from training/testing data
• Aloe’s analysis is only valid for similar distribution of input data

¹“Topaz”, S. Achour and M. Rinard, OOPSLA ‘15
²“Approximate Checkers”, A. Mahmoud et al., WAX ‘19
Example – Unreliable Multiplier Hardware

```java
try {
    z = x*y \[p_{try}\] rnd();
}
check {
    z == (x*y \[p_{try}\] rnd());
}
recover {
    z = x*y \[p_{rec}\] rnd();
}
```

- **try** multiplies x and y in an unreliable manner
- **check** re-executes the computation on same hardware
- We want z to be exact with probability \( \geq r \)
Execution Paths

\[
\begin{align*}
z &= x \times y \\
&\quad [p_{try}] \\
&\quad \text{try} \\
&\quad \text{pass} \\
&\quad (p_{try})^2 \\
&\quad \text{correct}
\end{align*}
\]

\[
\begin{align*}
z &= x \times y \\
&\quad [p_{try}] \\
&\quad \text{try} \\
&\quad \text{fail} \\
&\quad (1 - (p_{try})^2 - \epsilon) \cdot p_{rec}
\end{align*}
\]

\[
\begin{align*}
z &= x \times y \\
&\quad [p_{try}] \\
&\quad \text{try} \\
&\quad \text{error} \\
&\quad \epsilon \\
&\quad \left(1 - (p_{try})^2 - \epsilon\right) \cdot (1 - p_{rec})
\end{align*}
\]
Aloe Precondition Generation

\[ \left\{ \left( \left( p_{\text{try}} \right)^2 + \left( 1 - \left( p_{\text{try}} \right)^2 - \epsilon \right) \cdot p_{\text{rec}} \right) \cdot R(x, y) \geq r \right\} \]

\begin{align*}
\text{try} & \{ \\
& \quad z = x \times y \cdot \text{[p_{\text{try}}]} \cdot \text{rnd}(); \\
\} \\
\text{check} & \{ \\
& \quad z = (x \times y \cdot \text{[p_{\text{try}}]} \cdot \text{rnd}()); \\
\} \\
\text{recover} & \{ \\
& \quad \{ p_{\text{rec}} \cdot R(x, y) \geq r \} \\
& \quad z = x \times y \cdot \text{[p_{\text{rec}}]} \cdot \text{rnd}(); \\
\} \\
& \quad \{ R(z) \geq r \} \\
\end{align*}

Error-free rate of \text{try}: \quad p_{\text{try}}

True Negative: \quad p_{\text{try}}
False Positive: \quad 1 - p_{\text{try}}
False Negative: \quad \epsilon \ (\approx 0)
True Positive: \quad 1 - \epsilon
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>End-to-End Computation</th>
<th>Kernel Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank</td>
<td>PageRanks of graph nodes</td>
<td>Update PageRank of one node</td>
</tr>
<tr>
<td>Scale</td>
<td>Upscale an image</td>
<td>One pixel of upscaled image</td>
</tr>
<tr>
<td>Blackscholes</td>
<td>Prices of stock options</td>
<td>Price of one stock option</td>
</tr>
<tr>
<td>SSSP</td>
<td>Single Source Shortest Path</td>
<td>One iteration for one node</td>
</tr>
<tr>
<td>BFS</td>
<td>Breadth First Search</td>
<td>One search iteration for one node</td>
</tr>
<tr>
<td>SOR</td>
<td>Successive Over-Relaxation</td>
<td>One update for one element</td>
</tr>
<tr>
<td>Motion</td>
<td>Motion estimation</td>
<td>Similarity calculation for one block</td>
</tr>
<tr>
<td>Sobel</td>
<td>Edge detection filter</td>
<td>One pixel of filtered image</td>
</tr>
</tbody>
</table>

try-check-recover
Methodology

We model an architecture having multiple available reliability levels\(^1\)

Reliability of arithmetic operations:
\[\text{try} - 0.999 \quad \text{\color{red}{\text{\footnotesize{\textsuperscript{1}}}}}\]
\[\text{recover} - 0.9999 \quad \text{\color{red}{\text{\footnotesize{\textsuperscript{1}}}}}\]

\(^1\)“EnerJ”, A. Sampson et al., PLDI ‘11
Methodology

Perfect checkers: we simulate hardware support for detecting errors\textsuperscript{1,2}

Imperfect checkers: we experiment with different false positive/negative rates from Topaz\textsuperscript{3}

We compare Aloe’s analysis results to Rely

Rely uses if-then instead of try-check-recover

\textsuperscript{1}“Relax”, M. de Kruijf et al., ISCA ’10  \textsuperscript{2}“Argus”, A. Meixner et al., MICRO ’07  \textsuperscript{3}S. Achour and M. Rinard, OOPSLA ’15
## Reliability Calculated by Aloe (Perfect Checker)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Kernel-level Reliability</th>
<th>End-to-End Reliability</th>
<th>Aloe Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Aloe</td>
<td>Rely</td>
<td>Aloe</td>
</tr>
<tr>
<td>PageRank</td>
<td>0.9999</td>
<td>0.9531</td>
<td>≥ 0.99</td>
</tr>
<tr>
<td>Scale</td>
<td>0.9999</td>
<td>0.9891</td>
<td>≥ 0.99</td>
</tr>
<tr>
<td>Blackscholes</td>
<td>0.9999</td>
<td>0.9871</td>
<td>≥ 0.99</td>
</tr>
<tr>
<td>SSSP</td>
<td>0.999999</td>
<td>0.9920</td>
<td>≥ 0.99</td>
</tr>
<tr>
<td>BFS</td>
<td>0.99999</td>
<td>0.9227</td>
<td>≥ 0.99</td>
</tr>
<tr>
<td>SOR</td>
<td>0.99999</td>
<td>0.9950</td>
<td>≥ 0.99</td>
</tr>
<tr>
<td>Motion</td>
<td>0.9999</td>
<td>≈ 0.00</td>
<td>≥ 0.99</td>
</tr>
<tr>
<td>Sobel</td>
<td>0.9999</td>
<td>0.9930</td>
<td>≥ 0.99</td>
</tr>
</tbody>
</table>
More in the Paper

• error-free rate analysis of \texttt{try}

• Several additional examples

• Additional evaluation details
  • Testing setup
  • Unreliable checker and empirical analysis results

• [Appendix] Semantics and Aloe soundness proof
Conclusion

Aloe is the first static analysis of reliability of programs with recovery mechanisms

We analyzed eight kernels and end-to-end benchmarks with recovery mechanisms

Aloe can verify useful reliability bounds for all benchmarks