Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware

Michael Carbin Sasa Misailovic Martin C. Rinard
MIT CSAIL
{mcarbin, misailo, rinard}@csail.mit.edu

Abstract

Emerging high-performance architectures are anticipated to contain unreliable components that may exhibit *soft errors*, which silently corrupt the results of computations. Full detection and masking of soft errors is challenging, expensive, and, for some applications, unnecessary. For example, approximate computing applications (such as multimedia processing, machine learning, and big data analytics) can often naturally tolerate soft errors.

We present Rely, a programming language that enables developers to reason about the quantitative reliability of an application – namely, the probability that it produces the correct result when executed on unreliable hardware. Rely allows developers to specify the reliability requirements for each value that a function produces.

We present a static quantitative reliability analysis that verifies quantitative requirements on the reliability of an application, enabling a developer to perform sound and verified reliability engineering. The analysis takes a Rely program with a reliability specification and a hardware specification that characterizes the reliability of the underlying hardware components and verifies that the program satisfies its reliability specification when executed on the underlying unreliable hardware platform. We demonstrate the application of quantitative reliability analysis on six computations implemented in Rely.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Reliability, Verification

Keywords Approximate Computing

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2374-1/13/10.
http://dx.doi.org/10.1145/2509136.2509546

1. Introduction

System reliability is a major challenge in the design of emerging architectures. Energy efficiency and circuit scaling are becoming major goals when designing new devices. However, aggressively pursuing these design goals can often increase the frequency of *soft errors* in small [67] and large systems [10] alike. Researchers have developed numerous techniques for detecting and masking soft errors in both hardware [23] and software [20, 53, 57, 64]. These techniques typically come at the price of increased execution time, increased energy consumption, or both.

Many computations, however, can tolerate occasional unmasked errors. An *approximate computation* (including many multimedia, financial, machine learning, and big data analytics applications) can often acceptably tolerate occasional errors in its execution and/or the data that it manipulates [16, 44, 59]. A *checkable computation* can be augmented with an efficient checker that verifies the acceptability of the computation's results [8, 9, 35, 55]. If the checker does detect an error, it can reexecute the computation to obtain an acceptable result.

For both approximate and checkable computations, operating without (or with at most selectively applied) mechanisms that detect and mask soft errors can produce 1) fast and energy efficient execution that 2) delivers acceptably accurate results often enough to satisfy the needs of their users despite the presence of unmasked soft errors.

1.1 Background

Researchers have identified a range of both approximate computations [1, 2, 18, 29, 42–44, 59, 60, 64, 68, 72] and checkable computations [8, 9, 35, 55]. Their results show that it is possible to exploit these properties for a variety of purposes — increased performance, reduced energy consumption, increased adaptability, and increased fault tolerance. One key aspect of such computations is that they typically contain *critical* regions (which must execute without error) and *approximate* regions (which can execute acceptably even in the presence of occasional errors) [16, 59].

To support such computations, researchers have proposed energy-efficient architectures that, because they omit some

error detection and correction mechanisms, may expose soft errors to the computation [20, 23–25, 64]. A key aspect of these architectures is that they contain both reliable and (more efficient) unreliable components for executing the critical and approximate regions of a computation, respectively. The rationale behind this design is that developers can identify and separate the critical regions of the computation (which must execute on reliable hardware) from the approximate regions of the computation (which may execute on more efficient unreliable hardware).

Existing systems, tools, and type systems have focused on helping developers identify, separate, and reason about the binary distinction between critical and approximate regions [16, 24, 38, 59, 60, 64, 66]. However, in practice, no computation can tolerate an unbounded accumulation of soft errors — to execute acceptably, even the approximate regions must execute correctly with some minimum reliability.

1.2 Quantitative Reliability

We present a new programming language, Rely, and an associated program analysis that computes the *quantitative reliability* of the computation — i.e., the probability with which the computation produces a correct result when its approximate regions execute on unreliable hardware. More specifically, given a hardware specification and a Rely program, the analysis computes, for each value that the computation produces, a conservative probability that the value is computed correctly despite the possibility of soft errors.

In contrast to existing approaches, which support only a binary distinction between critical and approximate regions, quantitative reliability can provide precise static probabilistic acceptability guarantees for computations that execute on unreliable hardware platforms.

1.3 Rely

Rely is an imperative language that enables developers to specify and verify quantitative reliability specifications for programs that allocate data in unreliable memory regions and incorporate unreliable arithmetic/logical operations.

Quantitative Reliability Specifications. Rely supports quantitative reliability specifications for the results that functions produce. For example, a developer can declare a function signature `int<0.99*R(x, y)> f(int x, int y, int z)`, where `0.99*R(x, y)` is the reliability specification for `f`'s return value. The symbolic expression `R(x, y)` is the *joint reliability* of `x` and `y` – namely, the probability that they both simultaneously have the correct value on entry to the function. This specification states that the reliability of the return value of `f` must be *at least* 99% of `x` and `y`'s reliability when the function was called.

Joint reliabilities serve as an abstraction of a function's input distribution, which enables Rely's analysis to be both modular and oblivious to the exact shape of the distributions. This is important because 1) such exact shapes can be difficult for developers to identify and specify and 2) known tractable classes of probability distributions are not closed under many operations found in standard programming languages, which can complicate attempts to develop compositional program analyses that work with such exact shapes [27, 43, 65, 72].

Machine Model. Rely assumes a simple machine model that consists of a processor (with a register file and an arithmetic/logic unit) and a main memory. The model includes unreliable arithmetic/logical operations (which return an incorrect value with non-negligible probability [20, 24, 25, 64]) and unreliable physical memories (in which data may be written or read incorrectly with non-negligible probability [24, 38, 64]). Rely works with a *hardware reliability specification* that lists the probability with which each operation in the machine model executes correctly.

Language. Rely is an imperative language with integer, logical, and floating point expressions, arrays, conditionals, while loops, and function calls. In addition to these standard language features, Rely also allows a developer to allocate data in unreliable memories and write code that uses unreliable arithmetic/logical operations. For example, the declaration `int x in urel` allocates the variable `x` in an unreliable memory named `urel` where both reads and writes of `x` may fail with some probability. A developer can also write an expression `a +. b`, which is an unreliable addition of the values `a` and `b` that may produce an incorrect result.

Semantics. We have designed the semantics of Rely to exploit the full availability of unreliable computation in an application. Specifically, Rely only requires reliable computation at points where doing so ensures that programs are memory safe and exhibit control flow integrity.

Rely's semantics models an abstract machine that consists of a heap and a stack. The stack consists of frames that contain references to the locations of each invoked function's variables (which are allocated in the heap). To protect references from corruption, the stack is allocated in a reliable memory region and stack operations – i.e., pushing and popping frames – execute reliably. To prevent out-of-bounds memory accesses that may occur as a consequence of an unreliable array index computation, Rely requires that each array read and write includes a bounds check; these bounds check computations also execute reliably. Rely does not require a specific underlying mechanism to execute these operations reliably; one can use any applicable software or hardware technique [20, 26, 28, 51, 53, 57, 66, 70].

To prevent the execution from taking control flow edges that are not in the program's static control flow graph, Rely assumes that 1) instructions are stored, fetched, and decoded reliably (as is supported by existing unreliable processor architectures [24, 64]) and 2) control flow branch targets are computed reliably.

1.4 Quantitative Reliability Analysis

Given a Rely program and a hardware reliability specification, Rely's analysis uses a precondition generation approach to generate a symbolic *reliability precondition* for each function. A reliability precondition captures a set of constraints that is sufficient to ensure that a function satisfies its reliability specification when executed on the underlying unreliable hardware platform. The reliability precondition is a conjunction of predicates of the form $A_{out} \leq r \cdot \mathcal{R}(X)$, where A_{out} is a placeholder for a developer-provided reliability specification for an output named *out*, r is a real number between 0 and 1, and the term $\mathcal{R}(X)$ is the joint reliability of a set of parameters X.

Conceptually, each predicate specifies that the reliability given in the specification (given by A_{out}) should be less than or equal to the reliability of a path that the program may take to compute the result (given by $r \cdot \mathcal{R}(X)$). The analysis computes the reliability of a path from the probability that all operations along the path execute reliably.

The specification is valid if the probabilities for all paths to computing a result exceed that of the result's specification. To avoid the inherent intractability of considering all possible paths, Rely uses a simplification procedure to reduce the precondition to one that characterizes the least reliable path(s) through the function.

Loops. One of the core challenges in designing Rely and its analysis is dealing with unreliable computation within loops. The reliability of variables updated within a loop may depend on the number of iterations that the loop executes. Specifically, if a variable has a loop-carried dependence and updates to that variable involve unreliable operations, then the variable's reliability is a monotonically decreasing function of the number of iterations of the loop – on each loop iteration the reliability of the variable degrades relative to its previous reliability. If a loop does not have a compile-time bound on the maximum number of iterations, then the reliability of such a variable can, in principle, degrade arbitrarily, and the only conservative approximation of the reliability of such a variable is zero.

To provide specification and verification flexibility, Rely provides two types of loop constructs: statically *bounded* `while` loops and statically *unbounded* `while` loops. Statically bounded `while` loops allow a developer to provide a static bound on the *maximum* number of iterations of a loop. The dynamic semantics of such a loop is to exit if the number of executed iterations reaches this bound. This bound allows Rely's analysis to soundly construct constraints on the reliability of variables modified within the loop by unrolling the loop for its maximum bound.

Statically unbounded `while` loops have the same dynamic semantics as standard `while` loops. In the absence of a static bound on the number of executed loop iterations, however, Rely's analysis constructs a dependence graph of the loop's body to identify variables that are *reliably updated* – specifically, all operations that influence these variables' values are reliable. Because the execution of the loop does not decrease the reliability of these variables, the analysis identifies that their reliabilities are unchanged. For the remaining, unreliably updated variables, Rely's analysis conservatively sets their reliability to zero.

Specification Checking. In the last step of the analysis of a function, Rely verifies that the function's specifications are consistent with its reliability precondition. Because reliability specifications are also of the form $r \cdot \mathcal{R}(X)$, the final precondition is a conjunction of predicates of the form $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$, where $r_1 \cdot \mathcal{R}(X_1)$ is a reliability specification and $r_2 \cdot \mathcal{R}(X_2)$ is a path reliability. If these predicates are valid, then the reliability of each computed output is greater than that given by its specification.

The validity problem for these predicates has a sound mapping to the conjunction of two simple constraint validity problems: inequalities between real numbers ($r_1 \leq r_2$) and set inclusion constraints over finite sets ($X_2 \subseteq X_1$). Checking the validity of a reliability precondition is therefore decidable and efficiently checkable.

1.5 Case Studies

We have used Rely to build unreliable versions of six building block computations for media processing, machine learning, and data analytics applications. Our case studies illustrate how quantitative reliability enables a developer to use principled reasoning to relax the semantics of both approximate and checkable computations.

For approximate computations, quantitative reliability allows a developer to reify and verify the results of the fault injection and accuracy explorations that are typically used to identify the minimum acceptable reliability of a computation [43, 44, 68, 72]. For checkable computations, quantitative reliability allows a developer to use the performance specifications of both the computation and its checker to determine the computation's overall performance given that – with some probability – it may produce an incorrect answer and therefore needs to be reexecuted.

1.6 Contributions

This paper presents the following contributions:

Quantitative Reliability Specifications. We present quantitative reliability specifications, which characterize the probability that a program executed on unreliable hardware produces the correct result, as a constructive method for developing applications. Quantitative reliability specifications enable developers who build applications for unreliable hardware architectures to perform sound and verified reliability engineering.

Language and Semantics. We present Rely, a language that allows developers to specify reliability requirements for programs that allocate data in unreliable memory regions and use unreliable arithmetic/logical operations.

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
n & \in & \text{Int}_{\text{M}} & e \in \mathit{Exp} & \to & n \mid x \mid \texttt{(}\mathit{Exp}\texttt{)} \mid \mathit{Exp}\ \mathit{iop}\ \mathit{Exp} & m & \in & \text{MVar} \\
r & \in & \mathbb{R} & b \in \mathit{BExp} & \to & \texttt{true} \mid \texttt{false} \mid \mathit{Exp}\ \mathit{cmp}\ \mathit{Exp} \mid \texttt{(}\mathit{BExp}\texttt{)} \mid & V & \to & x \mid a \mid V, x \mid V, a \\
x, \ell & \in & \text{Var} & & & \mathit{BExp}\ \mathit{lop}\ \mathit{BExp} \mid \texttt{!}\mathit{BExp} \mid \texttt{!.}\mathit{BExp} & \mathit{RSpec} & \to & r \mid \texttt{R(}V\texttt{)} \mid r \texttt{*} \texttt{R(}V\texttt{)} \\
a & \in & \text{ArrVar} & \mathit{CExp} & \to & e \mid a & T & \to & \texttt{int} \mid \texttt{int<}\mathit{RSpec}\texttt{>}
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
 & & & D & \to & D_0\ [\texttt{in}\ m] \\
F & \to & (T \mid \texttt{void})\ \mathit{ID}\ \texttt{(}P^*\texttt{)}\ \{\ S\ \} & D_0 & \to & \texttt{int}\ x\ [= \mathit{Exp}] \mid \texttt{int}\ a\texttt{[}n^+\texttt{]} \\
P & \to & P_0\ [\texttt{in}\ m] & S_s & \to & \texttt{skip} \mid x\ \texttt{=}\ \mathit{Exp} \mid\ x\ \texttt{=}\ a\texttt{[}\mathit{Exp}^+\texttt{]} \mid a\texttt{[}\mathit{Exp}^+\texttt{]}\ \texttt{=}\ \mathit{Exp} \mid \\
P_0 & \to & \texttt{int}\ x \mid T\ a\texttt{(}n\texttt{)} & & & \mathit{ID}\texttt{(}\mathit{CExp}^*\texttt{)} \mid x\ \texttt{=}\ \mathit{ID}\texttt{(}\mathit{CExp}^*\texttt{)} \mid \texttt{if}_\ell\ \mathit{BExp}\ S\ S \mid S\ \texttt{;}\ S \\
S & \to & D^*\ S_s\ S_r^? & & & \texttt{while}_\ell\ \mathit{BExp}\ [\texttt{:}\ n]\ S\ \mid\ \texttt{repeat}_\ell\ n\ S \\
 & & & S_r & \to & \texttt{return}\ \mathit{Exp}
\end{array}
$$

Figure 1. Rely's Language Syntax

We present a dynamic semantics for Rely via a probabilistic small-step operational semantics. This semantics is parameterized by a hardware reliability specification that characterizes the probability that an unreliable arithmetic/logical or memory read/write operation executes correctly.

Semantics of Quantitative Reliability. We formalize the semantics of quantitative reliability as it relates to the probabilistic dynamic semantics of a Rely program. Specifically, we define the quantitative reliability of a variable as the probability that its value in an unreliable execution of the program is the same as that in a fully reliable execution. We also define the semantics of a logical predicate language that can characterize the reliability of variables in a program.

Quantitative Reliability Analysis. We present a program analysis that verifies that the dynamic semantics of a Rely program satisfies its quantitative reliability specifications. For each function in the program, the analysis computes a symbolic reliability precondition that characterizes the set of valid specifications for the function. The analysis then verifies that the developer-provided specifications are valid according to the reliability precondition.

Case Studies. We have used our Rely implementation to develop unreliable versions of six building block computations for media processing, machine learning, and data analytics applications. These case studies illustrate how to use quantitative reliability to develop and reason about both approximate and checkable computations in a principled way.

2. Example

Figure 1 presents the syntax of the Rely language. Rely is an imperative language for computations over integers, floats (not presented), and multidimensional arrays. To illustrate how a developer can use Rely, Figure 2 presents a Rely-based implementation of a pixel block search algorithm derived from that in the x264 video encoder [71].

The function `search_ref` searches a region (`pblocks`) of a previously encoded video frame to find the block of pixels that is most similar to a given block of pixels (`cblock`) in the current frame. The motion estimation algorithm uses the results of `search_ref` to encode `cblock` as a function of the identified block.

```
1  #define nblocks 20
2  #define height  16
3  #define width   16
4
5  int<0.99*R(pblocks, cblock)> search_ref (
6      int<R(pblocks)> pblocks(3) in urel,
7      int<R(cblock)>  cblock(2) in urel)
8  {
9    int minssd = INT_MAX,
10       minblock = -1 in urel;
11   int ssd, t, t1, t2 in urel;
12   int i = 0, j, k;
13
14   repeat nblocks  {
15     ssd = 0;
16     j = 0;
17     repeat height {
18       k = 0;
19       repeat width {
20         t1 = pblocks[i,j,k];
21         t2 = cblock[j,k];
22         t = t1 -. t2;
23         ssd = ssd +. t *. t;
24         k = k + 1;
25       }
26       j = j + 1;
27     }
28
29     if (ssd <. minssd) {
30       minssd = ssd;
31       minblock = i;
32     }
33
34     i = i + 1;
35   }
36   return minblock;
37 }
```

Figure 2. Rely Code for Motion Estimation Computation

This is an approximate computation that can trade correctness for more efficient execution by approximating the search to find a block. If `search_ref` returns a block that is not the most similar, then the encoder may require more bits to encode `cblock`, potentially decreasing the video's peak signal-to-noise ratio or increasing its size. However, previous studies on soft error injection [20] and more aggressive transformations like loop perforation [44, 68] have demonstrated that the quality of x264's final result is only slightly affected by perturbations of this computation.

2.1 Reliability Specifications

The function declaration on Line 5 specifies the types and reliabilities of `search_ref`'s parameters and return value. The parameters of the function are `pblocks(3)`, a three-dimensional array of pixels, and `cblock(2)`, a two-dimensional array of pixels. In addition to the standard signature, the function declaration contains *reliability specifications* for each result that the function produces.

Rely's reliability specifications express the reliability of a function's results – when executed on an unreliable hardware platform – as a function of the reliabilities of its inputs. The specification for the reliability of `search_ref`'s return value is `int<0.99*R(pblocks,cblock)>`. This states that the return value is an integer with a reliability that is at least 99% of the *joint reliability* of the parameters `pblocks` and `cblock` (denoted by `R(pblocks, cblock)`). The joint reliability of a set of parameters is the probability that they all have the correct value when passed in from the caller. This specification holds for all possible values of the joint reliability of `pblocks` and `cblock`. For instance, if the contents of the arrays `pblocks` and `cblock` are fully reliable (correct with probability one), then the return value is correct with probability 0.99.

In Rely, arrays are passed by reference and the execution of a function can, as a side effect, modify an array's contents. The reliability specification of an array therefore allows a developer to constrain the *reliability degradation* of its contents. Here `pblocks` has an output reliability specification of `R(pblocks)` (and similarly for `cblock`), meaning that all of `pblock`'s elements are at least as reliable when the function exits as they were on entry to the function.

2.2 Unreliable Computation

Rely targets hardware architectures that expose both reliable operations (which always execute correctly) and more energy-efficient unreliable operations (which execute correctly with only some probability). Specifically, Rely supports reasoning about reads and writes of unreliable memory regions and unreliable arithmetic/logical operations.

Memory Region Specification. Each parameter declaration also specifies the memory region in which the data of the parameter is allocated. Memory regions correspond to the physical partitioning of memory at the hardware level into regions of varying reliability. Here `pblocks` and `cblock` are allocated in an unreliable memory region named `urel`.

Lines 9-12 declare the local variables of the function. By default, variables in Rely are allocated in a default, fully reliable memory region. However, a developer can also optionally specify a memory region for each local variable. For example, the variables declared on Lines 9-11 reside in `urel`.

Unreliable Operations. The operations on Lines 22, 23, and 29 are unreliable arithmetic/logical operations. In Rely, every arithmetic/logical operation has an unreliable counterpart that is denoted by suffixing a period after the operation symbol. For example, "`-.`" denotes unreliable subtraction and "`<.`" denotes unreliable comparison.

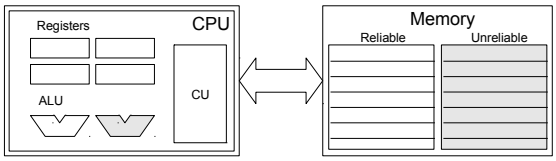

Figure 3. Machine Model Illustration. Gray boxes represent unreliable components

Using these operations, `search_ref`'s implementation *approximately* computes the index (`minblock`) of the most similar block, i.e. the block with the minimum distance from `cblock`. The `repeat` statement on line 14, iterates a constant `nblock` number of times, enumerating over all previously encoded blocks. For each encoded block, the `repeat` statements on lines 17 and 19 iterate over the `height*width` pixels of the block and compute the sum of the squared differences (`ssd`) between each pixel value and the corresponding pixel value in the current block `cblock`. Finally, the computation on lines 29 through 32 selects the block that is – approximately – the most similar to `cblock`.

2.3 Hardware Semantics

Figure 3 illustrates the conceptual machine model behind Rely's reliable and unreliable operations; the model consists of a CPU and a memory.

CPU. The CPU consists of 1) a register file, 2) arithmetic logical units that perform operations on data in registers, and 3) a control unit that manages the program's execution.

The arithmetic-logical unit can execute reliably or unreliably. We have represented this in Figure 3 by physically separate reliable and unreliable functional units, but this distinction can be achieved through other mechanisms, such as dual-voltage architectures [24]. Unreliable functional units may omit additional checking logic, enabling the unit to execute more efficiently but also allowing for soft errors that may occur due to, for example, power variations within the ALU's combinatorial circuits or particle strikes. As is provided by existing computer architecture proposals [24, 64], the control unit of the CPU reliably fetches, decodes, and schedules instructions; given a virtual address in the application, the control unit correctly computes a physical address and operates only on that physical address.

Memory. Rely supports machines with memories that consist of an arbitrary number of memory partitions (each potentially of different reliability), but for simplicity Figure 3 partitions memory into two regions: reliable and unreliable. Unreliable memories can, for example, use decreased DRAM refresh rates to reduce power consumption at the expense of increased soft error rates [38, 64].

2.4 Reliability Analysis

Given a Rely program, Rely's reliability analysis verifies that each function in the program satisfies its reliability specification when executed on unreliable hardware. Figure 4 presents an overview of Rely's analysis. It takes as input a Rely program and a *hardware reliability specification*.

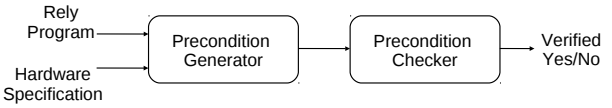

Figure 4. Rely's Analysis Overview

The analysis consists of two components: the *precondition generator* and the *precondition checker*. For each function, the precondition generator produces a precondition that characterizes the reliability of the function's results given a hardware reliability specification that characterizes the reliability of each unreliable operation. The precondition checker then determines if the function's specifications satisfy the constraint. If so, then the function satisfies its reliability specification when executed on the underlying unreliable hardware in that the reliability of its results exceed their specifications.

Design. As a key design point, the analysis generates preconditions according to a conservative approximation of the semantics of the function. Specifically, it characterizes the reliability of a function's result according to the probability that the function computes that result fully reliably.

To illustrate the intuition behind this design point, consider the evaluation of an integer expression e. The reliability of e is the probability that it evaluates to the same value n in an unreliable evaluation as in the fully reliable evaluation. There are two ways that an unreliable evaluation can return n: 1) the unreliable evaluation of e encounters no faults and 2) the unreliable evaluation possibly encounters faults, but still returns n by chance.

Rely's analysis conservatively approximates the reliability of a computation by only considering the first scenario. This design point simplifies our reasoning to the task of computing the probability that a result is reliably computed as opposed to reasoning about a computation's input distribution and the probabilities of all executions that produce the correct result. As a consequence, the analysis requires as input only a hardware reliability specification that gives the probability with which each arithmetic/logical operation and memory operation executes correctly. Our analysis is therefore oblivious to a computation's input distribution and does not require a full model of how soft errors affect its result.

2.4.1 Hardware Reliability Specification

Rely's analysis works with a hardware reliability specification that specifies the reliability of arithmetic/logical and memory operations. Figure 5 presents a hardware reliability specification that we have created using results from existing

```
reliability spec {
  operator (+.) = 1 - 10^-7;
  operator (-.) = 1 - 10^-7;
  operator (*.) = 1 - 10^-7;
  operator (<.) = 1 - 10^-7;
  memory rel {rd = 1, wr = 1};
  memory urel {rd = 1 - 10^-7, wr = 1};
}
```

Figure 5. Hardware Reliability Specification

computer architecture literature [23, 38]. Each entry specifies the reliability – the probability of a correct execution – of arithmetic operations (e.g., `+.`) and memory read/write operations.

For ALU operations, the presented reliability specification uses the reliability of an unreliable multiplication operation from [23, Figure 9]. For memory operations, the specification uses the probability of a bit flip in a memory cell from [38, Figure 4] with extrapolation to the probability of a bit flip within a 32-bit word. Note that a memory region specification includes two reliabilities: the reliability of a read (`rd`) and the reliability of a write (`wr`).

2.4.2 Precondition Generator

For each function, Rely's analysis generates a *reliability precondition* that conservatively bounds the set of valid specifications for the function. A reliability precondition is a conjunction of predicates of the form $A_{out} \leq r \cdot \mathcal{R}(X)$, where A_{out} is a placeholder for a developer-provided reliability specification for an output with name *out*, r is a numerical value between 0 and 1, and the term $\mathcal{R}(X)$ is the joint reliability of the set of variables X on entry to the function.

The analysis starts at the end of the function from a postcondition that must be true when the function returns and then works backward to produce a precondition such that if the precondition holds before execution of the function, then the postcondition holds at the end of the function.

Postcondition. The postcondition for a function is the constraint that the reliability of each array argument exceeds that given in its specification. For our example function `search_ref`, the postcondition Q_0 is

$$Q_0 = A_{\texttt{pblocks}} \leq \mathcal{R}(\texttt{pblocks}) \wedge A_{\texttt{cblock}} \leq \mathcal{R}(\texttt{cblock}),$$

which specifies that the reliability of the arrays `pblocks` and `cblock` – $\mathcal{R}(\texttt{pblocks})$ and $\mathcal{R}(\texttt{cblock})$ – should be at least that specified by the developer – $A_{\texttt{pblocks}}$ and $A_{\texttt{cblock}}$.

Precondition Generation. The analysis of the body of the `search_ref` function starts at the `return` statement. Given the postcondition Q_0, the analysis creates a new precondition Q_1 by conjoining to Q_0 a predicate that states that reliability of the return value ($r_0 \cdot \mathcal{R}(\texttt{minblock})$) is at least that of its specification (A_{ret}):

$$Q_1 = Q_0 \wedge A_{ret} \leq r_0 \cdot \mathcal{R}(\texttt{minblock}).$$

```
(3)   {Q0 ∧ A_ret ≤ r0^4 · R(i, ssd, minssd)
           ∧ A_ret ≤ r0^4 · R(minblock, ssd, minssd)}
      if (ssd <. minssd) {
(2)     {Q0 ∧ A_ret ≤ r0 · R(i, ℓ29)}
        minssd = ssd;
        {Q0 ∧ A_ret ≤ r0 · R(i, ℓ29)}
        minblock = i;
        {Q0 ∧ A_ret ≤ r0 · R(minblock, ℓ29)}
      } else {
(2)     {Q0 ∧ A_ret ≤ r0 · R(minblock, ℓ29)}
        skip;
        {Q0 ∧ A_ret ≤ r0 · R(minblock, ℓ29)}
      }
(1)   {Q0 ∧ A_ret ≤ r0 · R(minblock, ℓ29)}
```

Figure 6. `if` Statement Analysis in the Last Loop Iteration

The reliability of the return value comes from our design principle for reliability approximation. Specifically, this reliability is the probability of correctly reading `minblock` from unreliable memory – which is $r_0 = 1 - 10^{-7}$ according to the hardware reliability specification – multiplied by $\mathcal{R}(\texttt{minblock})$, the probability that the preceding computation correctly computed and stored `minblock`.

Loops. The statement that precedes the `return` statement is the `repeat` statement on Line 14. A key difficulty with reasoning about the reliability of variables modified within a loop is that if a variable is updated unreliably and has a loop-carried dependence then its reliability monotonically decreases as a function of the number of loop iterations. Because the reliability of such variables can, in principle, decrease arbitrarily in an unbounded loop, Rely provides both an unbounded loop statement (with an associated analysis, Section 5.2.3) and an alternative *bounded loop* statement that lets a developer specify a compile-time bound on the maximum number of its iterations that therefore bounds the reliability degradation of modified variables. The loop on Line 14 iterates `nblocks` times and therefore decreases the reliability of any modified variables `nblocks` times. Because the reliability degradation is bounded, Rely's analysis uses unrolling to reason about the effects of a bounded loop.

Conditionals. The analysis of the body of the loop on Line 14 encounters the `if` statement on Line 29.[1] This `if` statement uses an unreliable comparison operation on `ssd` and `minssd`, both of which reside in unreliable memory. The reliability of `minblock` when modified on Line 31 therefore also depends on the reliability of this expression because faults may force the execution down a different path.

Figure 6 presents a Hoare logic style presentation of the analysis of the conditional statement. The analysis works in three steps; the preconditions generated by each step are numbered with the corresponding step.

[1] This happens after encountering the increment of `i` on Line 34, which does not modify the current precondition because it does not reference `i`.

Step 1. To capture the implicit dependence of a variable on an unreliable condition, Rely's analysis first uses latent *control flow variables* to make these dependencies explicit. A control flow variable is a unique program variable (one for each statement) that records whether the conditional evaluated to *true* or *false*. We denote the control flow variable for the `if` statement on Line 29 by ℓ_{29}.

To make the control flow dependence explicit, the analysis adds the control flow variable to all joint reliability terms in Q_1 that contain variables modified within the body of the `if` conditional (`minssd` and `minblock`).

Step 2. The analysis next recursively analyses both the "then" and "else" branches of the conditional, producing one precondition for each branch. As in a standard precondition generator (e.g., weakest-preconditions) the assignment of `i` to `minblock` in the "then" branch replaces `minblock` with `i` in the precondition. Because reads from `i` and writes to `minblock` are reliable (according to the specification) the analysis does not introduce any new r_0 factors.

Step 3. In the final step, the analysis leaves the scope of the conditional and conjoins the two preconditions for its branches after transforming them to include the direct dependence of the control flow variable on the reliability of the `if` statement's condition expression.

The reliability of the `if` statement's expression is greater than or equal to the product of 1) the reliability of the `<.` operator (r_0), 2) the reliability of reading both `ssd` and `minssd` from unreliable memory (r_0^2), and 3) the reliability of the computation that produced `ssd` and `minssd` ($\mathcal{R}(\texttt{ssd}, \texttt{minssd})$). The analysis therefore transforms each predicate that contains the variable ℓ_{29}, by multiplying the right-hand side of the inequality with r_0^3 and replacing the variable ℓ_{29} with `ssd` and `minssd`. This produces the precondition Q_2:

$$Q_2 = Q_0 \wedge A_{ret} \leq r_0^4 \cdot \mathcal{R}(\texttt{i}, \texttt{ssd}, \texttt{minssd}) \wedge A_{ret} \leq r_0^4 \cdot \mathcal{R}(\texttt{minblock}, \texttt{ssd}, \texttt{minssd}).$$

Simplification. After unrolling a single iteration of the loop that begins at Line 14, the analysis produces $Q_0 \wedge A_{ret} \leq r_0^{2564} \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock}, \texttt{i}, \texttt{ssd}, \texttt{minssd})$ as the precondition for a single iteration of the loop's body. The constant 2564 represents the number of unreliable operations within a single loop iteration.

Note that there is one less predicate in this precondition than in Q_2. As the analysis works backwards through the program, it uses a simplification technique that identifies that a predicate $A_{ret} \leq r_1 \cdot \mathcal{R}(X_1)$ *subsumes* another predicate $A_{ret} \leq r_2 \cdot \mathcal{R}(X_2)$. Specifically, the analysis identifies that $r_1 \leq r_2$ and $X_2 \subseteq X_1$, which together mean that the second predicate is a weaker constraint on A_{ret} than the first and can therefore be removed. This follows from the fact that the joint reliability of a set of variables is less than or equal to the joint reliability of any subset of the variables – regardless of the distribution of their values.

This simplification is how Rely's analysis achieves scalability when there are multiple paths in the program; specifically a simplified precondition characterizes the least reliable path(s) through the program.

Final Precondition. When the analysis reaches the beginning of the function after fully unrolling the loop on Line 14, it has a precondition that bounds the set of valid specifications as a function of the reliability of the parameters of the function. For `search_ref`, the analysis generates the precondition $A_{ret} \leq 0.994885 \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock}) \wedge A_{\texttt{pblocks}} \leq \mathcal{R}(\texttt{pblocks}) \wedge A_{\texttt{cblock}} \leq \mathcal{R}(\texttt{cblock})$.

2.4.3 Precondition Checker

The final precondition is a conjunction of predicates of the form $A_{out} \leq r \cdot \mathcal{R}(X)$, where A_{out} is a placeholder for the reliability specification of an output. Because reliability specifications are all of the form $r \cdot \mathcal{R}(X)$ (Figure 1), each predicate in the final precondition (where each A_{out} is replaced with its specification) is of the form form $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$, where $r_1 \cdot \mathcal{R}(X_1)$ is a reliability specification and $r_2 \cdot \mathcal{R}(X_2)$ is computed by the analysis.

Similar to the analysis's simplifier (Section 2.4.2), the precondition checker verifies the validity of each predicate by checking that 1) r_1 is less than r_2 and 2) $X_2 \subseteq X_1$.

For `search_ref`, the analysis computes the predicates $0.99 \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock}) \leq 0.994885 \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock})$, $\mathcal{R}(\texttt{pblocks}) \leq \mathcal{R}(\texttt{pblocks})$, and $\mathcal{R}(\texttt{cblock}) \leq \mathcal{R}(\texttt{cblock})$. Because these predicates are valid according to our checking procedure, `search_ref` satisfies its reliability specification when executed on the specified unreliable hardware.

3. Language Semantics

Because soft errors may probabilistically change the execution path of a program, we model the semantics of a Rely program with a probabilistic, non-deterministic transition system. Specifically, the dynamic semantics defines probabilistic transition rules for each arithmetic/logical operation and each read/write on an unreliable memory region.

Over the next several sections we develop a small-step semantics that specifies the probability of each individual transition of an execution. In Section 3.4 we provide big-step definitions that specify the probability of an entire execution.

3.1 Preliminaries

Rely's semantics models an abstract machine that consists of a heap and a stack. The heap is an abstraction over the physical memory of the concrete machine, including its various reliable and unreliable memory regions. Each variable (both scalar and array) is allocated in the heap. The stack consists of frames – one for each function invocation – which contain references to the locations of each allocated variable.

Hardware Reliability Specification. A hardware reliability specification $\psi \in \Psi = (\mathit{iop} + \mathit{cmp} + \mathit{lop} + M_{op}) \to \mathbb{R}$ is a finite map from arithmetic/logical operations (*iop*, *cmp*, *lop*) and *memory region operations* (M_{op}) to reliabilities (i.e., the probability that the operation executes correctly).

Arithmetic/logical operations *iop*, *cmp*, and *lop* include both reliable and unreliable versions of each integer, comparison, and logical operation. The reliability of each reliable operation is 1 and the reliability of an unreliable operation is as provided by a specification (Section 2.4.1).

The finite maps $\mathit{rd} \in M \to M_{op}$ and $\mathit{wr} \in M \to M_{op}$ define memory region operations as reads and writes (respectively) on memory regions $m \in M$, where M is the set of all memory regions in the reliability specification.

The hardware reliability specification 1_ψ denotes the specification for fully reliable hardware in which all arithmetic/logical and memory operations have reliability 1.

References. A *reference* is a tuple $\langle n_b, \langle n_1, \ldots, n_k \rangle, m \rangle \in \mathrm{Ref}$ consisting of a base address $n_b \in \mathrm{Loc}$, a dimension descriptor $\langle n_1, \ldots, n_k \rangle$, and a memory region m. The address space Loc is finite. A base address and the components of a dimension descriptor are machine integers $n \in \mathrm{Int_M}$, which have finite bit width and therefore create a finite set.

References describe the location, dimensions, and memory region of variables in the heap. For scalars, the dimension descriptor is the single-dimension, single-element descriptor $\langle 1 \rangle$. We use the projections π_{base} and π_{dim} to select the base address and the dimension descriptor of a reference, respectively.

Frames, Stacks, Heaps, and Environments. A *frame* $\sigma \in \Sigma = \mathrm{Var} \to \mathrm{Ref}$ is a finite map from variables to references. A *stack* $\delta \in \Delta ::= \sigma \mid \sigma :: \Delta$ is a non-empty list of frames. A *heap* $h \in H = \mathrm{Loc} \to \mathrm{Int_M}$ is a finite map from addresses to machine integers. An *environment* $\varepsilon \in \mathrm{E} = \Delta \times H$ is a stack and heap pair, $\langle \delta, h \rangle$.

Memory Allocator. The abstract memory allocator *new* is a potentially non-deterministic partial function that executes reliably. It takes a heap h, a memory region m, and a dimension descriptor and returns a fresh address n_b that resides in memory region m and a new heap h' that reflects updates to the internal memory allocation data structures.

Auxiliary Probability Distributions. Each nondeterministic choice in Rely's semantics must have an underlying probability distribution so that the set of possible transitions at any given small step of an execution creates a probability distribution – i.e., the sum of the probabilities of each possibility is one. In Rely, there are two points at which an execution can make a nondeterministic choice: 1) the result of an incorrect execution of an unreliable operation and 2) the result of allocating a new variable in the heap.

The discrete probability distribution $P_f(n_f \mid \mathit{op}, n_1, ..., n_k)$ models the manifestation of a soft error during an incorrect execution of an operation. Specifically, it gives the prob-

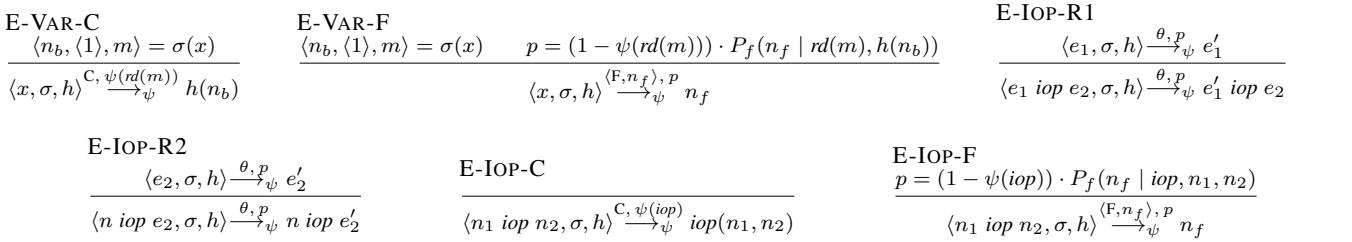

Figure 7. Dynamic Semantics of Integer Expressions

ability that an incorrect execution of an operation *op* on operands $n_1, \ldots, n_k$ produces a value n_f that is different from the correct result of the operation. This distribution is inherently tied to the properties of the underlying hardware.

The discrete probability distribution $P_m(n_b, h' \mid h, m, d)$ models the semantics of a nondeterministic memory allocator. It gives the probability that a memory allocator returns a fresh address n_b and an updated heap h' given an initial heap h, a memory region m, and a dimension descriptor d.

We define these distributions only to support a precise formalization of the dynamic semantics of a program; they do not need to be specified for a given hardware platform or a given memory allocator to use Rely's reliability analysis.

3.2 Semantics of Expressions

Figure 7 presents a selection of the rules for the dynamic semantics of integer expressions. The labeled probabilistic small-step evaluation relation $\langle e, \sigma, h\rangle \xrightarrow{\theta, p}_{\psi} e'$ states that from a frame σ and a heap h, an expression e evaluates in one step with probability p to an expression e' given a hardware reliability specification ψ. The label $\theta \in \{\mathrm{C}, \langle \mathrm{C}, n\rangle, \langle \mathrm{F}, n_f\rangle\}$ denotes whether the transition corresponds to a correct (C or $\langle \mathrm{C}, n\rangle$) or a faulty ($\langle \mathrm{F}, n_f\rangle$) evaluation of that step. For a correct transition $\langle \mathrm{C}, n\rangle$, $n \in \mathrm{Int_M}$ records a nondeterministic choice made for that step. For a faulty transition $\langle \mathrm{F}, n_f\rangle$, $n_f \in \mathrm{Int_M}$ represents the value that the fault introduced in the semantics of the operation.

To illustrate the meaning of the rules, consider the rules for variable reference expressions. A variable reference x reads the value stored in the memory address for x. The are two possibilities for the evaluation of a variable reference:

- Correct [E-VAR-C]. The variable reference evaluates correctly and successfully returns the integer stored in x. This happens with probability $\psi(rd(m))$, where m is the memory region in which x allocated. This probability is the reliability of reading from x's memory region.
- Faulty [E-VAR-F]. The variable reference experiences a fault and returns another integer n_f. The probability that the faulty execution returns a specific integer n_f is $(1 - \psi(rd(m))) \cdot P_f(n_f \mid rd(m), h(n_b))$. P_f is the distribution that gives the probability that a failed memory read operation returns a value n_f instead of the true stored value $h(n_b)$ (Section 3.1).

3.3 Semantics of Statements

Figure 8 presents the scalar and control flow fragment of Rely. The labeled probabilistic small-step execution relation $\langle s, \varepsilon\rangle \xrightarrow{\theta, p}_{\psi} \langle s', \varepsilon'\rangle$ states that execution of the statement s in the environment ε takes one step yielding a statement s' and an environment ε' with probability p under the hardware reliability specification ψ. As in the dynamic semantics for expressions, a label θ denotes whether the transition evaluated correctly (C or $\langle \mathrm{C}, n\rangle$) or experienced a fault ($\langle \mathrm{F}, n_f\rangle$). The semantics of the statements in our language are largely similar to that of traditional presentations except that the statements have the ability to encounter faults during execution.

The semantics we present here and in the Appendix [14] (which includes a semantics for arrays and functions) is designed to allow unreliable computation at all points in the application – subject to the constraint that the application is still memory safe and exhibits control flow integrity.

Memory Safety. To protect references that point to memory locations from corruption, the stack is allocated in a reliable memory region and stack operations – i.e., pushing and popping frames – execute reliably (see Appendix). To prevent out-of-bounds memory accesses that may occur due to an unreliable array index computation, Rely requires that each array read and write include a bounds check. These bounds check computations execute reliably (see Appendix).

Control Flow Integrity. To prevent execution from taking control flow edges that do not exist in the program's static control flow graph, Rely assumes that 1) instructions are stored, fetched, and decoded reliably (as supported by existing unreliable processor architectures [24, 64]) and 2) targets of control flow branches are reliably computed. These two properties allow for the control flow transfers in the rules [E-IF-TRUE], [E-IF-FALSE], and [E-SEQ-R2] to execute reliably with probability 1.

We note that the semantics does not require a specific underlying mechanism to achieve reliable execution and, therefore, an implementation can use any applicable software or hardware technique [20, 26, 28, 51, 53, 57, 66, 70].

3.4 Big-step Notations

We use the following big-step execution relations in the remainder of the paper.

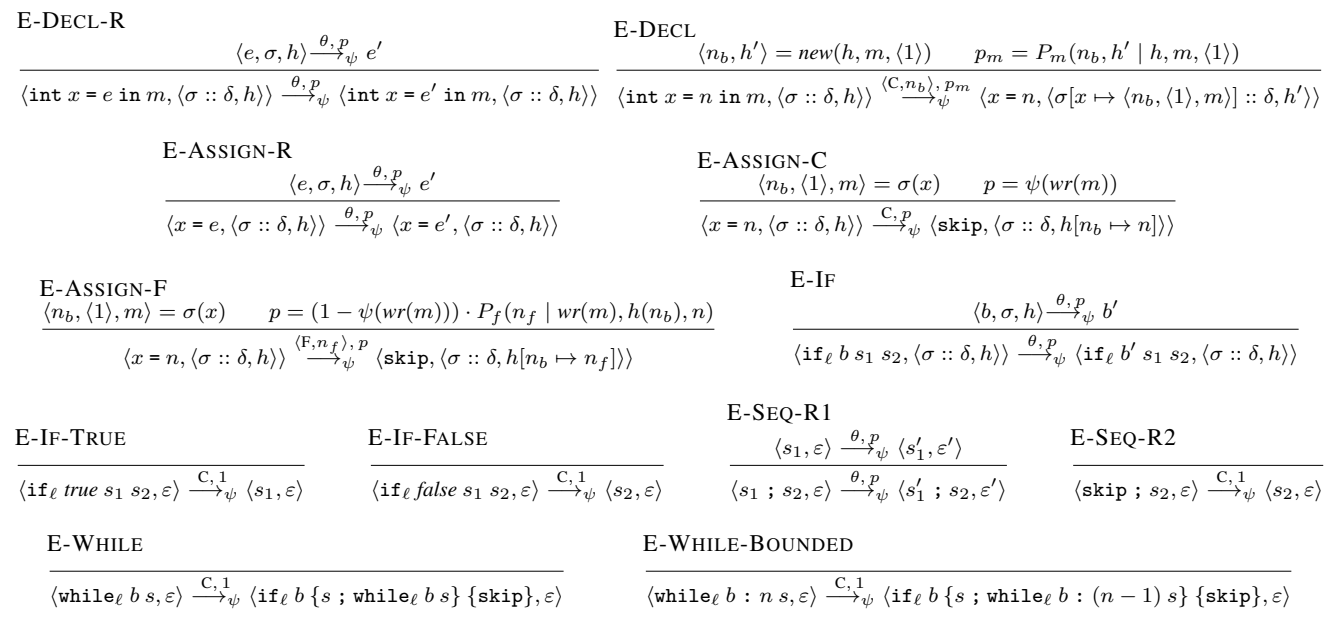

Figure 8. Dynamic Semantics of Statements

Definition 1 (Big-step Trace Semantics)**.**

$$\langle s,\varepsilon\rangle \xRightarrow{\tau,p}_{\psi} \varepsilon' \equiv \langle s,\varepsilon\rangle \xrightarrow{\theta_1,p_1}_{\psi} \ \ldots\ \xrightarrow{\theta_n,p_n}_{\psi} \langle \texttt{skip}, \varepsilon'\rangle$$
$$\textit{where } \tau = \theta_1,\ldots,\theta_n \textit{ and } p = \prod_i p_i$$

The big-step trace semantics is a reflexive transitive closure of the small-step execution relation that records a trace of the execution. A *trace* $\tau \in \mathrm{T} ::= \cdot \mid \theta :: \mathrm{T}$ is a sequence of small-step transition labels. The *probability of a trace*, p, is the product of the probabilities of each transition.

Definition 2 (Big-step Aggregate Semantics)**.**

$$\langle s,\varepsilon\rangle \xRightarrow{p}_{\psi} \varepsilon' \textit{ where } p = \sum_{\tau\in\mathrm{T}} p_\tau \textit{ such that } \langle s,\varepsilon\rangle \xRightarrow{\tau,p_\tau}_{\psi} \varepsilon'$$

The big-step aggregate semantics enumerates over the set of all finite length traces and collects the aggregate probability that a statement s evaluates to an environment ε' from an environment ε given a hardware reliability specification ψ. The big-step aggregate semantics therefore gives the total probability that a statement s starts from an environment ε and terminates in an environment ε'.[2]

Termination and Errors. An unreliable execution of a statement may experience a run-time error (due to an out-of-bounds array access) or not terminate at all. The big-step aggregate semantics does not collect such executions. Therefore, the sum of the probabilities of the big-step transitions from an environment ε may not equal to 1. Specifically, let $p \in \mathrm{E} \to \mathbb{R}$ be a measure for the set of environments reachable from ε, i.e., $\forall\varepsilon'.\langle s,\varepsilon\rangle \xRightarrow{p(\varepsilon')}_{\psi} \varepsilon'$. Then p is *subprobability* measure, i.e., $0 \le \sum_{\varepsilon'\in\mathrm{E}} p(\varepsilon') \le 1$ [32].

[2] The inductive (versus co-inductive) interpretation of T yields a countable set of finite-length traces and therefore the sum over T is well-defined.

4. Semantics of Quantitative Reliability

We next present the basic definitions that give a semantic meaning to the reliability of a Rely program.

4.1 Paired Execution

The *paired execution* semantics is the primary execution relation that enables us to reason about the reliability of a program. Specifically, the relation pairs the semantics of the program when executed reliably with its semantics when executed unreliably.

Definition 3 (Paired Execution)**.** $\varphi \in \Phi = \mathrm{E} \to \mathbb{R}$

$$\langle s,\langle\varepsilon,\varphi\rangle\rangle \Downarrow_{\psi} \langle\varepsilon',\varphi'\rangle \textit{ such that } \langle s,\varepsilon\rangle \xRightarrow{\tau,p_\tau}_{1_\psi} \varepsilon' \textit{ and}$$
$$\varphi'(\varepsilon_u') = \sum_{\varepsilon_u\in\mathrm{E}} \varphi(\varepsilon_u)\cdot p_u \textit{ where } \langle s,\varepsilon_u\rangle \xRightarrow{p_u}_{\psi} \varepsilon_u'$$

The relation states that from a *configuration* $\langle\varepsilon,\varphi\rangle$ consisting of an environment ε and an *unreliable environment distribution* φ, the paired execution of a statement s yields a new configuration $\langle\varepsilon',\varphi'\rangle$.

The environments ε and ε' are related by the fully reliable execution of s. Namely, an execution of s from an environment ε yields ε' under the fully reliable hardware model 1_ψ.

The unreliable environment distributions φ and φ' are probability mass functions that map an environment to the probability that the unreliable execution of the program is in that environment. In particular, φ is a distribution on environments before the unreliable execution of s whereas φ' is the distribution on environments after executing s. These distributions specify the probability of reaching a specific environment as a result of faults during the execution.

The unreliable environment distributions are discrete because E is a countable set. Therefore, φ' can be defined

pointwise: for any environment $\varepsilon'_u \in \mathrm{E}$, the value of $\varphi'(\varepsilon'_u)$ is the probability that the unreliable execution of the statement s results in the environment ε'_u given the distribution on possible starting environments, φ, and the aggregate probability p_u of reaching ε'_u from any starting environment $\varepsilon_u \in \mathrm{E}$ according to the big-step aggregate semantics. In general, φ' is a subprobability measure because it is defined using the big-step aggregate semantics, which is also a subprobability measure (Section 3.4).

4.2 Reliability Predicates and Transformers

The paired execution semantics enables us to define the semantics of statements as transformers on *reliability predicates* that bound the reliability of program variables. A reliability predicate P is a predicate of the form:

$$\begin{array}{lcl} P & \rightarrow & \texttt{true} \mid \texttt{false} \mid R \leq R \mid P \wedge P \\ R & \rightarrow & r \mid \mathcal{R}(X) \mid R \cdot R \end{array}$$

A predicate can either be the constant `true`, the constant `false`, a comparison between *reliability factors* (R), or a conjunction of predicates. A reliability factor is real-valued quantity that is either a constant r in the range $[0, 1]$; a joint reliability factor $\mathcal{R}(X)$ that gives the probability that all program variables in the set X have the same value in the unreliable execution as they have in the reliable execution; or a product of reliability factors, $R \cdot R$.

This combination of predicates and reliability factors enables us to specify bounds on the reliability of variables in the program, such as $0.99999 \leq \mathcal{R}(\{x\})$, which states the probability that x has the correct value in an unreliable execution is at least 0.99999.

4.2.1 Semantics of Reliability Predicates.

Figure 9 presents the denotational semantics of reliability predicates via the semantic function $[\![P]\!]$. The denotation of a reliability predicate is the set configurations that satisfy the predicate. We elide a discussion of the semantics of reliability predicates themselves because they are standard and instead focus on the semantics of joint reliability factors.

Joint Reliability Factor. A joint reliability factor $\mathcal{R}(X)$ represents the probability that an unreliable environment ε_u sampled from the unreliable environment distribution φ has the same values for all variables in the set X as that in the reliable environment ε. To define this probability, we use the function $\mathcal{E}(X, \varepsilon)$, which gives the set of environments that have the same values for all variables in X as in the environment ε. The denotation of a joint reliability factor is then the sum of the probabilities of each of these environments according to φ.

Auxiliary Definitions. We define predicate satisfaction and validity, respectively, as follows:

$$\begin{array}{rcl} \langle \varepsilon, \varphi \rangle \models P & \equiv & \langle \varepsilon, \varphi \rangle \in [\![P]\!] \\ \models P & \equiv & \forall \varepsilon. \forall \varphi. \langle \varepsilon, \varphi \rangle \models P \end{array}$$

4.2.2 Reliability Transformer

Given a semantics for predicates, we can now view the paired execution of a program as a *reliability transformer* – namely, a transformer on reliability predicates that is reminiscent of Dijkstra's Predicate Transformer Semantics [22].

Definition 4 (Reliability Transformer)**.**
$\psi \models \{P\}\ s\ \{Q\} \equiv$
$\forall \varepsilon. \forall \varphi. \forall \varepsilon'. \forall \varphi'. (\langle \varepsilon, \varphi \rangle \models P \wedge \langle s, \langle \varepsilon, \varphi \rangle \rangle \Downarrow_\psi \langle \varepsilon', \varphi' \rangle) \Rightarrow$
$\langle \varepsilon', \varphi' \rangle \models Q$

The paired execution of a statement s is a transformer on reliability predicates, denoted $\psi \models \{P\}\ s\ \{Q\}$. Specifically, the paired execution of s *transforms* P to Q if for all $\langle \varepsilon, \varphi \rangle$ that satisfy P and for all $\langle \varepsilon', \varphi' \rangle$ yielded by the paired execution of s from $\langle \varepsilon, \varphi \rangle$, $\langle \varepsilon', \varphi' \rangle$ satisfies Q. The paired execution of s transforms P to Q for any P and Q where this relationship holds.

Reliability predicates and reliability transformers allow us to use symbolic predicates to characterize and constrain the shape of the unreliable environment distributions before and after execution of a statement. This approach provides a well-defined domain in which to express Rely's reliability analysis as a generator of constraints on the shape of the unreliable environment distributions for which a function satisfies its reliability specification.

5. Reliability Analysis

For each function in a program, Rely's reliability analysis generates a symbolic *reliability precondition* with a precondition generator style analysis. The reliability precondition is a reliability predicate that constrains the set of specifications that are valid for the function. Specifically, the reliability precondition is of the form $\bigwedge_{i,j} R_i \leq R_j$ where R_i is the reliability factor for a developer-provided specification of a function output and R_j is a reliability factor that gives a conservative lower bound on the reliability of that output. If the reliability precondition is valid, then the developer-provided specifications are valid for the function.

5.1 Preliminaries

Transformed Semantics. We formalize Rely's analysis over a transformed semantics of the program that we produce via a source-to-source transformation function $\mathcal{T}$ that performs two transformations:

- **Conditional Flattening.** Each conditional has a unique *control flow variable* ℓ associated with it that we use to flatten a conditional of the form $\texttt{if}_\ell\ (b)\ \{s_1\}\ \{s_2\}$ to the sequence $\ell = b\ ;\ \texttt{if}_\ell\ (\ell)\ \{s_1\}\ \{s_2\}$. This transformation reifies the control flow variable as an explicit program variable that records the value of the conditional.
- **SSA.** We transform a Rely program to a SSA renamed version of the program. The ϕ-nodes for a conditional include a reference to the control flow variable for the

$$[\![P]\!] \in \mathcal{P}(\mathrm{E} \times \Phi) \qquad [\![\texttt{true}]\!] = \mathrm{E} \times \Phi \qquad [\![\texttt{false}]\!] = \varnothing \qquad [\![P_1 \wedge P_2]\!] = [\![P_1]\!] \cap [\![P_2]\!]$$

$$[\![R_1 \ \leq \ R_2]\!] = \{\langle \varepsilon, \varphi \rangle \mid [\![R_1]\!](\varepsilon, \varphi) \leq [\![R_2]\!](\varepsilon, \varphi)\}$$

$$[\![R]\!] \in \mathrm{E} \times \Phi \to \mathbb{R} \qquad [\![r]\!](\varepsilon, \varphi) = r \qquad [\![R_1 \cdot R_2]\!](\varepsilon, \varphi) = [\![R_1]\!](\varepsilon, \varphi) \cdot [\![R_2]\!](\varepsilon, \varphi) \qquad [\![\mathcal{R}(X)]\!](\varepsilon, \varphi) = \sum_{\varepsilon_u \in \mathcal{E}(X, \varepsilon)} \varphi(\varepsilon_u)$$

$$\mathcal{E} \in \mathcal{P}(\mathrm{Var} + \mathrm{ArrVar}) \times \mathrm{E} \to \mathcal{P}(\mathrm{E}) \qquad \mathcal{E}(X, \varepsilon) = \{\varepsilon' \mid \varepsilon' \in \mathrm{E} \wedge \forall v.\, v \in X \Rightarrow \mathrm{equiv}(\varepsilon', \varepsilon, v)\})$$

$$\mathrm{equiv}(\langle \sigma' :: \delta', h' \rangle, \langle \sigma :: \delta, h \rangle, v) = \forall i \,.\, 0 \leq i < \mathrm{len}(v, \sigma) \Rightarrow h'(\pi_{\mathsf{base}}(\sigma'(v)) + i) = h(\pi_{\mathsf{base}}(\sigma(v)) + i)$$

$$\mathrm{len}(v, \sigma) = \text{let } \langle n_0, \ldots, n_k \rangle = \pi_{\mathsf{dim}}(\sigma(v)) \text{ in } \prod_{0 \leq i \leq k} n_i$$

Figure 9. Predicate Semantics

conditional. For example, we transform a sequence of the form ℓ = b ; $\texttt{if}_\ell$ (ℓ) $\{x = 1\}$ $\{x = 2\}$ to the sequence ℓ = b ; $\texttt{if}_\ell$ (ℓ) $\{x_1 = 1\}$ $\{x_2 = 2\}$; $x = \phi(\ell, x_1, x_2)$. We rely on standard treatments for the semantics of ϕ-nodes [4]. For arrays, we use a simple Array SSA [31].

We also note that we apply the SSA transformation such that a reference of a parameter at any point in the body of the function refers to its initial value on entry to the function. This property naturally gives a function's reliability specifications a semantics that refers to the reliability of variables on entry to the function.

These two transformations together allow us to make explicit the dependence between the reliability of a conditional's control flow variable and the reliability of variables modified within.

Auxiliary Maps. The map $\Lambda \in \mathrm{Var} \to M$ is a map from program variables to their declared memory regions. We compute this map by inspecting the parameter and variable declarations in the function. The map $\Gamma \in \mathrm{Var} \to R$ is a unique map from the outputs of a function – namely, the return value and arrays passed as parameters – to the reliability factors (Section 4.2) for the developer-provided specification of each output. We allocate a fresh variable named ret that represents the return value of the program.

Substitution. A substitution $e_0[e_2/e_1]$ replaces all occurrences of the expression e_1 with the expression e_2 within the expression e_0. Multiple substitution operations are applied from left to right. The substitution matches set patterns. For instance, the pattern $\mathcal{R}(\{x\} \cup X)$ represents a joint reliability factor that contains the variable x, alongside with the remaining variables in the set X. Then, the result of the substitution $r_1 \cdot \mathcal{R}(\{x, z\})[r_2 \cdot \mathcal{R}(\{y\} \cup X)/\mathcal{R}(\{x\} \cup X)]$ is the expression $r_1 \cdot r_2 \cdot \mathcal{R}(\{y, z\})$.

5.2 Precondition Generation

The analysis generates preconditions according to a conservative approximation of the paired execution semantics. Specifically, it characterizes the reliability of a value in a function according to the probability that the function computes that value – including its dependencies – fully reliably given a hardware specification.

Figure 10 presents a selection of Rely's reliability precondition generation rules. The generator takes as input a statement s, a postcondition Q, and (implicitly) the maps Λ and Γ. The generator produces as output a precondition P, such that if P holds before the paired execution of s, then Q holds after.

We have crafted the analysis so that Q is the constraint over the developer-provided specifications that must hold at the end of execution of a function. Because arrays are passed by reference in Rely and can therefore be modified, one property that must hold at the end of execution of a function is that each array must be at least as reliable as implied by its specification. Our analysis captures this property by setting the initial Q for the body of a function to

$$\bigwedge_{a_i} \Gamma(a_i) \leq \mathcal{R}(a_i')$$

where a_i is the ith array parameter of the function and a_i' is an SSA renamed version of the array that contains the appropriate value of a_i at the end of the function. This constraint therefore states that the reliability implied by the specifications must be less than or equal to the actual reliability of each input array at the end of the function. As the precondition generator works backwards through the function, it generates a new precondition that – if valid at the beginning of the function – ensures that Q holds at the end.

5.2.1 Reasoning about Expressions

The topmost part of Figure 10 first presents our rules for reasoning about the reliability of evaluating an expression. The reliability of evaluating an expression depends on two factors: 1) the reliability of the operations in the expression and 2) the reliability of the variables referenced in the expression. The function $\rho \in (\mathit{Exp} + \mathit{BExp}) \to \mathbb{R} \times \mathcal{P}(\mathrm{Var})$ computes the core components of these two factors. It returns a pair consisting of 1) the probability of correctly executing all operations in the expression and 2) the set of variables referenced by the expression. The projections ρ_1 and ρ_2 return each component, respectively. Using these projections, the

$$\rho \in (\mathit{Exp} + \mathit{BExp}) \to \mathbb{R} \times \mathcal{P}(\mathrm{Var}) \qquad \rho(n) = (1, \varnothing) \qquad \rho(x) = (\psi(\mathit{rd}(\Lambda(x))), \{x\})$$

$$\rho(e_1 \mathit{\ iop\ } e_2) = (\rho_1(e_1) \cdot \rho_1(e_2) \cdot \psi(\mathit{iop}),\ \ \rho_2(e_1) \cup \rho_2(e_2)) \qquad \rho_1(e) = \pi_1(\rho(e)) \qquad \rho_2(e) = \pi_2(\rho(e))$$

$$\begin{array}{rcl}
\mathit{RP}_\psi & \in & S \times P \to P \\
\mathit{RP}_\psi(\texttt{return}\ e, Q) & = & Q \wedge \Gamma(\mathrm{ret}) \leq \rho_1(e) \cdot \mathcal{R}(\rho_2(e)) \\
\mathit{RP}_\psi(x = e, Q) & = & Q\ [(\rho_1(e) \cdot \psi(\mathit{wr}(\Lambda(x))) \cdot \mathcal{R}(\rho_2(e) \cup X))/\mathcal{R}(\{x\} \cup X)] \\
\mathit{RP}_\psi(x = a[e_1, \ldots, e_n], Q) & = & Q\ [((\prod_i \rho_1(e_i)) \cdot \psi(\mathit{rd}(\Lambda(a))) \cdot \psi(\mathit{wr}(\Lambda(x))) \cdot \mathcal{R}(\{a\} \cup (\bigcup_i \rho_2(e_i)) \cup X))/\mathcal{R}(\{x\} \cup X)] \\
\mathit{RP}_\psi(a[e_1, \ldots, e_n] = e, Q) & = & Q\ [(\rho_1(e) \cdot (\prod_i \rho_1(e_i)) \cdot \psi(\mathit{wr}(\Lambda(a))) \cdot \mathcal{R}(\rho_2(e) \cup (\bigcup_i \rho_2(e_i)) \cup \{a\} \cup X))/\ \mathcal{R}(\{a\} \cup X)] \\
\mathit{RP}_\psi(\texttt{skip}, Q) & = & Q \\
\mathit{RP}_\psi(s_1\ ;\ s_2, Q) & = & \mathit{RP}_\psi(s_1, \mathit{RP}_\psi(s_2, Q)) \\
\mathit{RP}_\psi(\texttt{if}_\ell\ \ell\ s_1\ s_2, Q) & = & \mathit{RP}_\psi(s_1, Q) \wedge \mathit{RP}_\psi(s_2, Q) \\
\mathit{RP}_\psi(x = \phi(\ell, x_1, x_2), Q) & = & Q\ [\mathcal{R}(\{\ell, x_1\} \cup X)/\mathcal{R}(\{x\} \cup X)] \wedge Q\ [\mathcal{R}(\{\ell, x_2\} \cup X)/\mathcal{R}(\{x\} \cup X)] \\
\mathit{RP}_\psi(\texttt{while}_\ell\ b : 0\ s, Q) & = & Q \\
\mathit{RP}_\psi(\texttt{while}_\ell\ b : n\ s, Q) & = & \mathit{RP}_\psi(\mathcal{T}(\texttt{if}_{\ell_n}\ b\ \{s\ ;\ \texttt{while}_\ell\ b : (n-1)\ s\}\ \texttt{skip}), Q) \\
\mathit{RP}_\psi(\texttt{int}\ x = e\ \texttt{in}\ m, Q) & = & \mathit{RP}_\psi(x = e, Q) \\
\mathit{RP}_\psi(\texttt{int}\ a[n_0, \ldots, n_k]\ \texttt{in}\ m, Q) & = & Q\ [\mathcal{R}(X)/\mathcal{R}(\{a\} \cup X)]
\end{array}$$

Figure 10. Reliability Precondition Generation

reliability of an expression e – given any reliable environment and unreliable environment distribution – is therefore *at least* $\rho_1(e) \cdot \mathcal{R}(\rho_2(e))$, where $\mathcal{R}(\rho_2(e))$ is the joint reliability of all the variables referenced in e. We elide the rules for boolean and relational operations, but they are defined analogously.

5.2.2 Generation Rules for Statements

We next present the precondition generation rules for Rely statements. As in a precondition generator, the analysis works backwards from the end of the program towards the beginning. We have therefore structured our discussion of the statements starting with function returns.

Function Returns. When execution reaches a function return, `return` e, the analysis must verify that the reliability of the return value is greater than the reliability that the developer specified. To verify this, the analysis rule generates the additional constraint $\Gamma(\mathrm{ret}) \leq \rho_1(e) \cdot \mathcal{R}(\rho_2(e))$. This constrains the reliability of the return value, where $\Gamma(\mathrm{ret})$ is the reliability specification for the return value.

Assignment. For the program to satisfy a predicate Q after the execution of an assignment statement $x = e$, then Q must hold given a substitution of the reliability of the expression e for the reliability of x. The substitution $Q[(\rho_1(e) \cdot \psi(\mathit{wr}(\Lambda(x))) \cdot \mathcal{R}(\rho_2(e) \cup X))/\mathcal{R}(\{x\} \cup X)]$ binds each reliability factor in which x occurs – $\mathcal{R}(\{x\} \cup X)$ – and replaces the factor with a new reliability factor $\mathcal{R}(\rho_2(e) \cup X)$ where $\rho_2(e)$ is the set of variables referenced by e.

The substitution also multiplies the reliability factor by $\rho_1(e) \cdot \psi(\mathit{wr}(\Lambda(x)))$, which is the probability that e evaluates fully reliably and its value is reliably written to the memory location for x.

Array loads and stores. The reliability of a load statement $x = a[e_1, \ldots, e_n]$ depends on the reliability of the indices $e_1, \ldots, e_n$, the reliability of the values stored in a, and the reliability of reading from a's memory region. The rule's implementation is similar to that for assignment.

The reliability of an array store $a[e_1, \ldots, e_n] = e$ depends on the reliability of the source expression e, the reliability of the indices $e_1, \ldots, e_n$, and the reliability of writing to a. Note that the rule preserves the presence of a within the reliability term. By doing so, the rule ensures that it tracks the full reliability of all the elements within a.

Conditional. For the program to satisfy a predicate Q after a conditional statement $\texttt{if}_\ell\ b\ s_1\ s_2$, each branch must satisfy Q. The rule therefore generates a precondition that is a conjunction of the results of the analysis of each branch.

Phi-nodes. The rule for a ϕ-node $x = \phi(\ell, x_1, x_2)$ captures the implicit dependence of the effects of control flow on the value of a variable x. For the merged value x, the rule establishes Q by generating a precondition that ensures that Q holds independently for both x_1 and x_2, given an appropriate substitution. Note that the rule also includes ℓ in the substitution; this explicitly captures x's dependence on ℓ. The flattening statement inserted before a conditional (Section 5.1), later replaces the reliability of ℓ with its dependencies.

Bounded `while` and `repeat`. Bounded `while` loops, $\texttt{while}_\ell\ b : n\ s$, and `repeat` loops, `repeat` n s, execute their bodies at most n times. Execution of such a loop therefore satisfies Q if P holds beforehand, where P is the result of invoking the analysis on n sequential copies of the body. The rule implements this approach via a sequence of bounded recursive calls to transformed versions of itself.

Unbounded `while`. We present the analysis for unbounded `while` loops in Section 5.2.3.

Function calls. The analysis for functions is modular and takes the reliability specification from the function declaration and substitutes the reliabilities of the function's formal arguments with the reliabilities of the expressions that represent the corresponding actual arguments of the function. We present the rule for function calls in the Appendix [14].

Note that this modular approach supports reasoning about recursion. When we analyze a function, if we assume that the specification of a recursive invocation is valid, then the result of the recursive call is no more reliable then the specification we're trying to verify. If we perform any unreliable computation on that result, then it is less reliable then our specification and therefore cannot be verified unless the given specification is zero. This is consistent with our analysis of unbounded `while` loops (Section 5.2.3).

5.2.3 Unbounded `while` Loops.

An unbounded loop, $\texttt{while}_\ell\ b\ s$, may execute for a number of iterations that is not bounded statically. The reliability of a variable that is modified unreliably within a loop and has a loop-carried dependence is a monotonically decreasing function of the number of loop iterations. The only sound approximation of the reliability of such a variable is therefore zero. However, unbounded loops may also update a variable reliably. In this case, the reliability of the variable is the joint reliability of its dependencies. We have implemented an analysis for unbounded `while` loops to distinguish these two cases as follows:

Dependence Graph. Our analysis first constructs a dependence graph for the loop. Each node in the dependence graph corresponds to a variable that is read or written within the condition or body of the loop. There is a directed edge from the node for a variable x to the node for a variable y if the value of y depends on the value of x. We additionally classify each edge as reliable or unreliable meaning that a reliable or unreliable operation creates the dependence.

There is an edge from the node for a variable x to the node for the variable y if one of the following holds:

- **Assignment:** there is an assignment to y where x occurs in the expression on the right hand side of the assignment; this condition captures direct data dependencies. We classify such an edge as reliable if every operation in the assignment (i.e., the operations in the expression and the write to memory itself) are reliable. Otherwise, we mark the edge as unreliable. The rules for array load and store statements are similar, and include dependencies induced by the computation of array indices.
- **Control Flow Side Effects:** y is assigned within an `if` statement and the `if` statement's control flow variable is named x; this condition captures control dependencies. We classify each such edge as reliable.

The analysis uses the dependence graph to identify the set of variables in the loop that are *reliably updated*. A variable x is reliably updated if all simple paths (and simple cycles) to x in the dependence graph contain only reliable edges.

Fixpoint Analysis. Given a set of reliably updated variables X_r, the analysis next splits the postcondition Q into two parts. For each predicate $R_i \leq r \cdot \mathcal{R}(X)$ in Q (where R_i is a developer-provided specification), the analysis checks if the property $\forall x \in X.\, x \in \text{modset}(s) \Rightarrow x \in X_r$ holds, where $\text{modset}(s)$ computes the set of variables that may be modified by s. If this holds, then all the variables in X are either modified reliably or not modified at all within the body of the loop. The analysis conjoins the set of predicates that satisfy this property to create the postcondition Q_r and conjoins the remaining predicates to create Q_u.

The analysis next iterates the function $F(A)$ starting from `true`, where $F(A) = Q_r \wedge RP_\psi(\mathcal{T}(\texttt{if}_\ell\ b\ s\ \texttt{skip}), A)$, until it reaches a fixpoint. The resulting predicate Q'_r is a translation of Q_r such the joint reliability of a set of variables is replaced by the joint reliability of its dependencies.

Lemma 1 (Termination)**.** *Iteration of $F(A)$ terminates.*

This follows by induction on iterations, the monotonicity of RP and the fact that the range of $F(A)$ (given a simplifier that removes redundant predicates, Section 5.4) is finite (together, finite descending chains). The key intuition is that the set of real-valued constants in the precondition before and after an iteration does not change (because all variables are reliably updated) and the set of variables that can occur in a joint reliability factor is finite. Therefore, there are a finite number of unique preconditions in the range of $F(A)$. We present a proof sketch in the Appendix [14].

Final Precondition. In the last step, the analysis produces a final precondition that preserves the reliability of variables that are reliably updated by conjoining Q'_r with the predicate $Q_u[(R_i \leq 0)/(R_i \leq R_j)]$, where R_i and R_j are joint reliability factors. The substitution on Q_u sets the joint reliability factors that contain unreliably updated variables to zero.

5.2.4 Properties

Rely's reliability analysis is sound with respect to the transformer semantics laid out in Section 4.

Theorem 2 (Soundness)**.** $\psi \models \{RP_\psi(s, Q)\}\ s\ \{Q\}$

This theorem states that if a configuration $\langle \varepsilon, \varphi \rangle$ satisfies a generated precondition and the paired execution of s yields a configuration $\langle \varepsilon', \varphi' \rangle$, then $\langle \varepsilon', \varphi' \rangle$ satisfies Q. Alternatively, s transforms the precondition generated by our analysis to Q. We present a proof sketch in the Appendix [14].

5.3 Specification Checking

As the last step of the analysis for a function, the analysis checks the developer-provided reliability specifications for the function's outputs as captured by the precondition generator's final precondition. Because each specification has

the form $r \cdot \mathcal{R}(X)$ (Figure 1) the precondition is a conjunction of predicates of the form $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$. While these joint reliability factors represent arbitrary and potentially complex distributions of the values of X_1 and X_2, there is simple and sound (though not complete) procedure to check the validity of each predicate in a precondition that follows from the *ordering* of joint reliability factors.

Proposition 1 (Ordering)**.** *For two sets of variables X and Y, if $X \subseteq Y$ then $\mathcal{R}(Y) \leq \mathcal{R}(X)$.*

This follows from the fact that the joint reliability of a set of variables Y is less than or equal to the joint reliability of any subset of the variables – regardless of the distribution of their values. As a consequence of the ordering of joint reliability factors, there is a simple and sound method to check the validity of a predicate.

Corollary 1 (Predicate Validity)**.** *If $r_1 \leq r_2$ and $X_2 \subseteq X_1$ then $\models r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$.*

The constraint $r_1 \leq r_2$ is a comparison of two real numbers and the constraint $X_2 \subseteq X_1$ is an inclusion of finite sets. Note that both types of constraints are decidable and efficiently checkable.

Checking. Because the predicates in the precondition generator's output are mutually independent, we can use Corollary 1 to check the validity of the full precondition by checking the validity of each predicate in turn.

5.4 Implementation

We implemented the parser for the Rely language, the precondition generator, and the precondition checker in OCaml. The implementation consists of 2500 lines of code. The analysis can operate on numerical or symbolic hardware reliability specifications. Our implementation performs simplification transformations after each precondition generator step to simplify numerical expressions and remove predicates that are trivially valid or *subsumed* by another predicate.

Proposition 2 (Predicate Subsumption)**.** *A reliability predicate $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$ subsumes (i.e., soundly replaces) a predicate $r_1' \cdot \mathcal{R}(X_1') \leq r_2' \cdot \mathcal{R}(X_2')$ if $r_1' \cdot \mathcal{R}(X_1') \leq r_1 \cdot \mathcal{R}(X_1)$ and $r_2 \cdot \mathcal{R}(X_2) \leq r_2' \cdot \mathcal{R}(X_2')$*

6. Case Studies

We next discuss six computations (three checkable, three approximate) that we implemented and analyzed with Rely.

6.1 Analysis Summary

Table 11 presents our benchmark computations and Rely's analysis results. For each benchmark, we present the type of the computation (checkable or approximate), its length in lines of code (LOC), the execution time of the analysis, and the number of inequality predicates in the final precondition produced by the precondition generator.

Benchmark	Type	LOC	Time (ms)	Predicates
newton	Checkable	21	8	1
secant	Checkable	30	7	2
coord	Checkable	36	19	1
search_ref	Approximate	37	348	3
matvec	Approximate	32	110	4
hadamard	Approximate	87	18	3

Figure 11. Benchmark Analysis Summary

Benchmarks. We analyze the following six computations:

- **newton:** This computation searches for a root of a univariate function using Newton's Method.
- **secant:** This computation searches for a root of a univariate function using the Secant Method.
- **coord:** This computation calculates the Cartesian coordinates from the polar coordinates passed as the input.
- **search_ref:** This computation performs a simple motion estimation. We presented this computation in Section 2.
- **mat_vec:** This computation multiplies a matrix and a vector and stores the result in another vector.
- **hadamard:** This computation takes as input two blocks of 4x4 pixels and computes the sum of differences between the pixels in the frequency domain.

We provide a detailed description of the benchmarks, including the Rely source code, in the Appendix [14].

Analysis Time. We executed Rely's analysis on an Intel Xeon E5520 machine with 16 GB of main memory. The analysis times for all benchmarks are under one second.

Number of Predicates. We used the hardware reliability specification from Figure 5 to generate a reliability precondition for each benchmark. The number of predicates in each benchmark's precondition is small (all less than five) because simplification removes most of the additional predicates introduced by the rules for conditionals. Specifically, these predicates are often subsumed by another predicate.

6.2 Reliability and Accuracy

A developer's choice of reliability specifications is typically influenced by the perceived effect that the unreliable execution of the computation may have on the accuracy of its result and its execution time and energy consumption. We present two case studies that illustrate how developers can use Rely to reason about the tradeoffs between accuracy and performance that are available for checkable computations and approximate computations.

6.2.1 Checkable Computations

Checkable computations are those that can be augmented with an efficient checker that dynamically verifies the correctness of the computation's result. If the checker detects an error, then it reexecutes the computation or executes an alternative reliable implementation. We next present how a developer can use Rely to build and reason about the performance of a *checked* implementation of Newton's method.

```
1  #define tolerance 0.000001
2  #define maxsteps 40
3
4  float<0.9999*R(x)> F(float x in urel);
5  float<0.9999*R(x)> dF(float x in urel);
6
7  float <0.99*R(xs)> newton(float xs in urel){
8     float x, xprim in urel;
9     float t1, t2 in urel;
10
11    x = xs;
12    xprim = xs +. 2*.tolerance;
13
14    while ((x -. xprim >=. tolerance)
15           ||. (x -. xprim <=. -tolerance)
16          ) : maxsteps {
17       xprim = x;
18       t1 = F(x);
19       t2 = dF(x);
20       x = x -. t1 /. t2;
21    }
22
23    if (!.((x -. xprim <=. tolerance)
24         &&. (x -. xprim >=. -tolerance))) {
25        x = INFTY;
26    }
27    return x;
28 }
```

Figure 12. Newton's Method Implementation

Newton's method. Figure 12 presents an unreliable implementation of Newton's method in Rely. Newton's method searches for a root of a function; given a differentiable function $f(x)$, its derivative $f'(x)$, and a starting point x_s, it computes a value x_0 such that $f(x_0) = 0$.

This is an example of a fixed-point iteration computation that executes a `while` loop at most `maxstep` steps. The computation within each loop iteration of the method can execute unreliably: each iteration updates the estimate of the root x by computing the value of the function f and the derivative f'. If the computation converges in the maximum number of steps, the function returns the produced value. Otherwise it returns the error value (infinity). The reliability of the computation depends on the reliability of the starting value x_s and the reliability of the functions f and f'. If the reliability specification of f is `float<0.9999*R(x)> F(float x)` (and similar for f'), then the analysis verifies that the reliability of the whole computation is at least `0.99*R(xs)`.

Checked Implementation. A developer can build a checked implementation of `newton` with the following code:

```
float root = newton(xs);
float ezero = f(root);
if (ezero < -tolerance || ezero > tolerance)
  root = newton_r(xs);
```

To check the reliability of the root x_0 that `newton` produces, it is sufficient to evaluate the function $f(x_0)$ and check if the result is zero (within some tolerance). If the checker detects that the result is not a zero, then the computation calls `newton_r`, a fully reliable alternative implementation.

Reliability/Execution Time Tradeoff. Quantitative reliability allows us to model the performance of this checked implementation of Newton's method. Let τ_u be the expected execution time of `newton`, τ_{um} the expected execution time of `newton` when executed for its maximum number of internal loop iterations, τ_c the expected execution time of the checker, and τ_r the expected execution time of `newton_r`.

The expected execution time of the checked computation when `newton` produces a correct result is $T_1 = \tau_u + \tau_c$. In the case when `newton` produces an incorrect result, the expected execution time is at most $T_2 = \tau_{um} + \tau_c + \tau_r$ (i.e., the maximum expected execution time of `newton` plus the expected execution time of both the checker and the reexecution via `newton_r`). This formula is conservative because it assumes that a fault causes `newton` to execute for its maximum number of iterations.

If we denote the reliability of `newton` by r, then the expected execution time of the checked computation as a whole is $T' = r \cdot T_1 + (1 - r) \cdot T_2$, which produces a projected speedup s, where $s = \tau_r / T'$. These formulas allow a developer to find the reliability r that meets the developer's performance improvement goal and can be analogously applied for alternative resource usage measures, such as energy consumption and throughput.

Example. As an illustration, let us assume that the computation executes on unreliable hardware and its reliable version is obtained using software-level replication. Using the replication approach proposed in [57], the replicated implementation is 40% slower than the unreliable version – i.e, $\tau_r = 1.4\tau_u$. Furthermore, let the reliable Newton's method computation converge on average in a half of the maximum number of steps (i.e., $\tau_u = \tau_{um}/2$) and let the execution time of the checker be half of the time of a single iteration of Newton's method. For a projected speedup of 1.32, the developer can use the previous formulas to compute the target reliability $r = 0.99$. Rely can verify that `newton_u` executes with this reliability (given the hardware reliability specification from Figure 5) when the input x_s is fully reliable.

6.2.2 Approximate Computations

Many applications can tolerate inaccuracies in the results of their internal approximate computations, which are computations that can produce a range of results of different quality. The approximate computations in these applications can be transformed to trade accuracy of their results for increased performance by producing lower quality results.

In building and transforming approximate computations, there are two correctness properties that a developer or transformation system must consider: *integrity* – which ensures that the computation produces a valid result that the remaining computation can consume without failing – and *accuracy* – which determines the quality of the result itself [12].

In this case study, we present how a developer can use Rely in combination with other techniques to reason about the integrity and accuracy of `search_ref` (Section 2).

Integrity. Recall that `search_ref` searches for the index of the block within the array of pixel blocks `pblocks` that is the minimum distance from the block of pixels `cblock`. An important integrity property for `search_ref` is therefore that it returns a valid index: an index that is at least `0` and at most `nblocks - 1`. However, this property may not hold in an unreliable execution because `search_ref`'s unreliable operations may cause it to return an arbitrary result. To guard against this, a developer has several options.

One of the developer's options is to modify `search_ref` to dynamically check that its result is valid and reexecute itself if the check fails. This case is analogous to that for checkable computations (Section 6.2.1), with the distinction that the checker implements a partial correctness specification.

Another option is to modify `search_ref` to dynamically check that its result is valid and instead of reexecuting itself if the check fails, it *rectifies* [39, 58] its result by returning a valid index at random. This enables `search_ref` to produce a valid – but still approximate – result.

Alternatively, the developer can place `minblock` in reliable memory and set its initial value to a fixed, valid index (e.g., `0`); this implements a fixed rectification approach. Because `i` is also stored in reliable memory, `minblock` will always contain a valid index despite `search_ref`'s other unreliable operations. The developer can establish this fact either informally or formally with relational verification [11].

Accuracy. A computation's reliability bound states how often the computation returns a correct result and therefore also states a conservative bound on the computation's accuracy. To determine an acceptable reliability bound (and therefore an acceptable accuracy), the developer can perform local or end-to-end accuracy experiments [29, 44, 68]. As an illustration of an end-to-end experiment, we present a simulation approach for `search_ref`.

To estimate the effects of the unreliable execution, we modified a fully reliable version of `search_ref` (one without unreliable operations) to produce the correct minimum distance block with probability p and produce the maximum distance block with probability $1 - p$. This modification provides a conservative estimate of the bound on `search_ref`'s accuracy loss given a reliability p (when inputs to the computation are reliable) and the assumption that a fault causes `search_ref` to return the worst possible result.

We then implemented two versions of the x264 video encoder [71]: one with the reliable version of `search_ref` and one with the modified version. For several values of p, we then compared the videos produced by the reliable and modified encoders on 17 HD video sequences (each 200 frames in length) from the Xiph.org Video Test Media repository [40]. We quantified the difference between the quality of the resulting videos via the Quality Loss Metric (QLM), previously used in [44]. This metric computes a relative difference between the quality scores of the two videos, each of which is computed as a weighted sum of the peak signal to noise ratio and the encoded video size.

p	0.90	0.95	0.97	0.98	0.99
QLM	0.041	0.021	0.012	0.009	0.004

Figure 13. `search_ref` Simulation Result

Table 13 presents the average QLM as a function of the reliability of `search_ref`. A developer can use the results of the simulation to identify an acceptable amount of quality loss for the encoded video. For example, if the developer is willing to accept at most 1% quality loss (which corresponds to the QLM value 0.01), then the developer can select 0.98 from Table 13 as the target reliability for an unreliable version of `search_ref`. The reliability specification that the developer then writes for an unreliable version is `0.98*R(pblocks,cblock)`. As demonstrated in Section 2, Rely's analysis can verify that the presented unreliable implementation satisfies an even tighter reliability (i.e., 0.99).

7. Extensions

In this section we discuss several extensions to the research we presented in this paper.

Unreliable Inputs. Some computations (for example, computations that read unreliable sensors or work with noisy data) may process unreliable inputs. A straightforward extension of Rely's analysis would incorporate this additional source of unreliability into the analysis framework. The resulting enhanced system would make it possible to reason about how unreliable inputs propagate through the program to affect the reliability of the outputs, enabling a global understanding of the impact of unreliable inputs on the reliability of the computation as a whole.

Optimization. It is possible to extend Rely to optimize the placement of unreliable operations to maximize the performance of the computation subject to a reliability specification. One approach would be to extend the precondition generator to produce a distinct symbolic constant (instead of $\psi(op)$) for each location that contains an arithmetic operation. The final precondition can be used to select arithmetic operations that can execute unreliably while still satisfying the given reliability bound.

The precondition generator could be similarly extended to generate symbolic expressions for upper bounds on the number of iterations of bounded loops. These expressions could then help the developer select loop bounds to satisfy the reliability specification of a function.

Dynamic Reliability Analysis. Rely's static analysis produces sound but conservative reliability estimates. A dynamic analysis could use fault injection to obtain potentially more precise (but unsound) reliability estimates. It would, in principle, be possible to combine these analyses. One approach would use a dynamic analysis to obtain reliability specifications for selected functions, then the static analysis for the remaining part of the program. In this scenario the static analysis of callers would use the dynamically derived reliability specifications for callees.

Precise Array Analysis. A more precise array analysis can be used to distinguish reads and writes to different array elements. We anticipate that existing techniques for analyzing data dependencies in loops can be used to produce more precise sets of array locations that an array read or write operation can affect. For example, we can extend Rely with array alias specifications which would allow for more precise reasoning for loops where elements of one array are unreliably computed from elements of a separate array.

Reliability Preconditions. The function's reliability specifications that Rely's analysis verifies are valid for all calling context. Developers may, however, desire specify conditions that are valid only for some calling context, such as a precondition that the function is called only on fully reliable inputs. We anticipate that Rely's general predicate logic will make it possible to extend Rely with preconditions without modifying the underlying analysis domain and structure.

8. Related Work

8.1 Critical and Approximate Regions

Almost all approximate computations have critical regions that must execute without error for the computation as a whole to execute acceptably.

Dynamic Criticality Analysis. One way to identify critical and approximate regions is to change different regions of the computation or data in some way and observe the effect. To the best of our knowledge, Rinard was the first to present a technique (task skipping) designed to identify critical and approximate regions in approximate computations [59, 60]. Carbin and Rinard subsequently presented a technique that uses directed fuzzing to identify critical and approximate computations, program data, and input data [16]. Other techniques use program transformations [44] and input fuzzing [3].

Static Criticality Analysis. Researchers have also developed several specification-based static analyses that let the developer identify and separate critical and approximate regions. Flikker [38] is a set of language extensions with a runtime and hardware support to enable more energy efficient execution of programs on inherently unreliable memories. It allows a developer to partition data into critical and approximate regions (but does not enforce full separation between the regions). Based on these annotations, the Flikker runtime allocates and stores data in a reliable or unreliable memory. Sampson et al. [64] present EnerJ, a programming language with an information-flow type system that allows a developer to partition program's data into approximate and critical data and ensures that operations on approximate data do not affect critical data or memory safety of programs.

All of this prior research focuses on the binary distinction between reliable and approximate computations. In contrast, the research presented in this paper allows a developer to specify and verify that even approximate computations produce the correct result most of the time. Overall, this additional information can help developers better understand the effects of deploying their computations on unreliable hardware and exploit the benefits that unreliable hardware offers.

8.2 Relational Reasoning for Approximate Programs

Carbin et al. [11] present a verification system for relaxed approximate programs based on a relational Hoare logic. The system enables rigorous reasoning about the integrity and worst-case accuracy properties of a program's approximate regions. Their work builds upon the relational verification techniques originated in Rinard et al.'s Credible Compilation [63], Pnueli et al.'s Translation Validation [54], and later by Benton's Relational Hoare Logic [7].

Rely differs from these approaches because of its probabilistic relational reasoning: specifically, the probability that an unreliable implementation returns the correct result. However, these non-probabilistic approaches are complementary to Rely in that they enable reasoning about the non-probabilistic properties of an approximate computation.

8.3 Accuracy Analysis

In addition to reasoning about how often a computation may produce a correct result, it may also be desirable to reason about the accuracy of the result that the computation produces. Dynamic techniques observe the accuracy impact of program transformations [1, 2, 29, 41, 42, 44, 59, 60, 62, 68], or injected soft errors [20, 37, 38, 64, 70]. Researchers have developed static techniques that use probabilistic reasoning to characterize the accuracy impact of various phenomena [6, 18, 43, 47, 56, 72]. And of course, the accuracy impact of the floating point approximation to real arithmetic has been extensively studied by numerical analysts [17].

8.4 Probabilistic Program Analysis

Kozen's work [32] was the first to propose the analysis of probabilistic programs as *transformers* of discrete probability distributions. Researchers have since developed a number of program analyses for probabilistic programs, including those based on axiomatic reasoning [5, 6, 46], abstract interpretation [19, 21, 45, 69], and symbolic execution [27, 65].

The language features that introduce probabilistic nondeterminism in programs that this previous research studied include probabilistic sampling, `x = random()` [5, 6, 32, 45], probabilistic choice between statements, `s1` $\oplus_p$ `s2` [19, 21, 46], and specifications of the distributions of computation's inputs [69]. Rely refines the probabilistic operators by defining a set of unreliable arithmetic and memory read/write operations that model faults in the underlying hardware.

Morgan et al. [46] propose a weakest-precondition style analysis for probabilistic programs that treats the programs as *expectation transformers*. Preconditions and postconditions are defined as bounds on probabilities that particular logical predicates hold at a specified location in the program. Rely's analysis, like [46], constructs precondition predicates for program statements. However, Rely's predicates are relational, relating the states of the reliable and unreliable ex-

ecutions of the program. Moreover, Rely's analysis is more precise as it uses direct multiplicative lower bounds on reliability as opposed to additive upper bounds on error.

Barthe et al. [5] define a probabilistic relational Hoare logic for a simple probabilistic imperative language that is similar to Kozen's. The relational predicates are arbitrary conjunctions or disjunctions of relational expressions over program variables, each of which is endowed with a probability of being true. While general, this approach requires manual proofs or an SMT solver to verify the validity of predicates. In comparison, Rely presents a semantics for reliability predicates that incorporates joint reliability factors, which create a simple and efficient checking procedure.

Reliability Analysis. Analyzing the reliability of complex physical and software systems is a classical research problem [36]. More recently researchers have presented symbolic execution techniques for checking complex probabilistic assertions. Filieri et al. [27] present analysis of finite-state Java programs. Sankaranarayanan et al. [65] present analysis of computations with linear expressions and potentially unbounded loops. These techniques require knowledge of the distributions of the inputs to the computation. Rely's analysis requires only the probability with which each operation in the computation executes correctly.

8.5 Fault Tolerance and Resilience

Researchers have developed various software, hardware, or mixed approaches for detection and recovery from specific types of soft errors that guarantee a reliable program execution [13, 20, 26, 28, 30, 51–53, 57, 61, 66, 70]. For example, Reis et al. [57] present a compiler that replicates a computation to detect and recover from single event upset errors.

These techniques are complementary to Rely in that each can provide implementations of operations that need to be reliable, as either specified by the developer or as required by Rely, to preserve memory safety and control flow integrity.

8.6 Emerging Hardware Architectures

Recently researchers have proposed multiple hardware architectures to trade reliability for additional energy or performance savings. Some of the recent research efforts include probabilistic CMOS chips [50], stochastic processors [48], error resilient architecture [34], unreliable memories [33, 38, 49, 64], and the Truffle architecture [24]. These techniques typically use voltage scaling at different granularities.

This previous research demonstrated that for specific classes of applications, such as multimedia processing and machine learning, the proposed architectures provided energy or time savings profitable to the user. Rely aims to help developers better understand and control the behavior of their applications on such platforms.

9. Conclusion

Driven by hardware technology trends, future computational platforms are projected to contain unreliable hardware components. To safely exploit the benefits (such as reduced energy consumption) that such unreliable components may provide, developers need to understand the effect that these components may have on the overall reliability of the approximate computations that execute on them.

We present a language, Rely, for exploiting unreliable hardware and an associated analysis that provides probabilistic reliability guarantees for Rely computations executing on unreliable hardware. By enabling developers to better understand the probabilities with which this hardware enables approximate computations to produce correct results, these guarantees can help developers safely exploit the significant benefits that unreliable hardware platforms offer.

Acknowledgements

We thank Deokhwan Kim, Hank Hoffmann, Vladimir Kiriansky, Stelios Sidiroglou, Rishabh Singh, and the anonymous referees for their useful comments. We also note our previous technical report [15].

This research was supported in part by the National Science Foundation (Grants CCF-0905244, CCF-1036241, CCF-1138967, CCF-1138967, and IIS-0835652), the United States Department of Energy (Grant DE-SC0008923), and DARPA (Grants FA8650-11-C-7192, FA8750-12-2-0110).

References

[1] J. Ansel, Y. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. CGO, 2011.

[2] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. PLDI, 2010.

[3] T. Bao, Y. Zheng, and X. Zhang. White box sampling in uncertain data processing enabled by program analysis. OOPSLA, 2012.

[4] G. Barthe, D. Demange, and D. Pichardie. A formally verified ssa-based middle-end: Static single assignment meets compcert. ESOP, 2012.

[5] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. POPL, 2009.

[6] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic reasoning for differential privacy. POPL, 2012.

[7] N. Benton. Simple relational correctness proofs for static analyses and program transformations. POPL, 2004.

[8] M. Blum and S. Kanna. Designing programs that check their work. STOC, 1989.

[9] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of computer and system sciences*, 1993.

[10] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 2009.

[11] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. PLDI, 2012.

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

[13] M. Carbin, S. Misailovic, M. Kling, and M. Rinard. Detecting and escaping infinite loops with Jolt. ECOOP, 2011.

[14] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability of programs that execute on unreliable hardware (appendix). http://groups.csail.mit.edu/pac/rely.

[15] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability of programs that execute on unreliable hardware. Technical Report MIT-CSAIL-TR-2013-014, MIT, 2013.

[16] M. Carbin and M. Rinard. Automatically identifying critical input regions and code in applications. ISSTA, 2010.

[17] F. Chaitin-Chatelin and V. Fraysse. *Lectures on finite precision computations*. 1996.

[18] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. FSE, 2011.

[19] P. Cousot and M. Monerau. Probabilistic abstract interpretation. ESOP, 2012.

[20] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. ISCA '10.

[21] A. Di Pierro and H. Wiklicky. Concurrent constraint programming: Towards probabilistic abstract interpretation. PPDP, 2000.

[22] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8), August 1975.

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

[24] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. ASPLOS, 2012.

[25] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. MICRO, 2012.

[26] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. ASPLOS, 2010.

[27] A. Filieri, C. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. ICSE, 2013.

[28] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. DSN, 2002.

[29] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. ASPLOS, 2011.

[30] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. OOPSLA, 2012.

[31] K. Knobe and V. Sarkar. Array ssa form and its use in parallelization. POPL, 1998.

[32] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 1981.

[33] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Mitigating soft error failures for multimedia applications by selective data protection. CASES, 2006.

[34] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra. Ersa: error resilient system architecture for probabilistic applications. DATE, 2010.

[35] N. Leveson, S. Cha, J. C. Knight, and T. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE TSE*, 1990.

[36] N. Leveson and P. Harvey. Software fault tree analysis. *Journal of Systems and Software*, 3(2), 1983.

[37] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. HPCA, 2007.

[38] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn. Flikker: saving dram refresh-power through critical data partitioning. ASPLOS, 2011.

[39] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and Martin Rinard. Automatic input rectification. ICSE, 2012.

[40] Xiph.org Video Test Media. http://media.xiph.org/video/derf.

[41] J. Meng, A. Raghunathan, S. Chakradhar, and S. Byna. Exploiting the forgiving nature of applications for scalable parallel execution. IPDPS, 2010.

[42] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM TECS Special Issue on Probabilistic Embedded Computing*, 2013.

[43] S. Misailovic, D. Roy, and M. Rinard. Probabilistically accurate program transformations. SAS, 2011.

[44] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. ICSE, 2010.

[45] D. Monniaux. Abstract interpretation of probabilistic semantics. SAS, 2000.

[46] C. Morgan, A. McIver, and K. Seidel. Probabilistic predicate transformers. *TOPLAS*, 1996.

[47] D. Murta and J. N. Oliveira. Calculating fault propagation in functional programs. Technical report, Univ. Minho, 2013.

[48] S. Narayanan, J. Sartori, R. Kumar, and D. Jones. Scalable stochastic processors. DATE, 2010.

[49] J. Nelson, A. Sampson, and L. Ceze. Dense approximate storage in phase-change memory. ASPLOS Ideas & Perspectives, 2011.

[50] K. Palem. Energy aware computing through probabilistic switching: A study of limits. *IEEE Transactions on Computers*, 2005.

[51] K. Pattabiraman, V. Grover, and B. Zorn. Samurai: protecting critical data in unsafe languages. EuroSys, 2008.

[52] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. Ernst, and M. Rinard. Automatically patching errors in deployed software. SOSP, 2009.

[53] F. Perry, L. Mackey, G.A. Reis, J. Ligatti, D.I. August, and D. Walker. Fault-tolerant typed assembly language. PLDI, 2007.

[54] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. TACAS, 1998.

[55] P. Prata and J. Silva. Algorithm based fault tolerance versus result-checking for matrix computations. FTCS, 1999.

[56] J. Reed and B. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. ICFP, 2010.

[57] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. Swift: Software implemented fault tolerance. CGO 05, 2005.

[58] M. Rinard. Acceptability-oriented computing. OOPSLA, 2003.

[59] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. ICS, 2006.

[60] M. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. OOPSLA, 2007.

[61] M. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S. Beebee Jr. Enhancing server availability and security through failure-oblivious computing. OSDI, 2004.

[62] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. OOPSLA Onwards!, 2010.

[63] M. Rinard and D. Marinov. Credible compilation with pointers. RTRV, 1999.

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

[65] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. PLDI, 2013.

[66] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. Zorn. Yarra: An extension to c for data integrity and partial safety. CSF '11.

[67] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. DSN, 2002.

[68] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. FSE, 2011.

[69] M. Smith. Probabilistic abstract interpretation of imperative programs using truncated normal distributions. *Electronic Notes in Theoretical Computer Science*, 2008.

[70] A. Thomas and K. Pattabiraman. Error detector placement for soft computation. DSN, 2013.

[71] x264. http://www.videolan.org/x264.html.

[72] Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. POPL, 2012.