Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2013-014 June 19, 2013

Verifying Quantitative Reliability of Programs That Execute on Unreliable Hardware

Michael Carbin, Sasa Misailovic, and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Verifying Quantitative Reliability of 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 recovery from 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.

In this paper 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.

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 [49] and large systems [8] alike. Researchers have developed numerous techniques for detecting and recovering from soft errors in both hardware [18] and software [15, 38, 42, 47]. These techniques typically come at the price of increased execution time, increased energy consumption, or both.

Many computations, however, can tolerate occasional unmasked errors. *Approximate computations* (including many multimedia, financial, machine learning, and big data analytics applications) can often acceptably tolerate occasional errors that occur in approximate parts of the computation and/or the data that it manipulates [11, 32, 43]. *Checkable computations* can be augmented with an efficient checker that ensures that the computation produced a correct result. If the checker does detect an error, it can reexecute the computation to obtain a correct result.

For both approximate and checkable computations, the benefits of fast and energy efficient execution without (or with selectively applied) fault tolerance mechanisms may outweigh the drawbacks of exposed soft errors.

1.1 Background

Researchers have identified a range of both approximate computations [1, 2, 13, 23, 30–32, 43, 44, 47, 50, 55] and checkable computations [6, 7, 27, 40]. 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* parts (which must execute without error) and *approximate* parts (which can execute acceptably even in the presence of occasional errors).

To support such computations, researchers have proposed energy-efficient architectures that, because they omit some error detection and correction mechanisms, may expose some soft errors to the computation [15, 18–20, 47]. A key aspect of these architectures is that they contain both reliable and (more efficient) unreliable components that facilitate executing the critical and approximate parts of a computation, respectively. The rationale behind this design is that developers can identify and separate the critical parts of the computation (which must execute on the reliable hardware) from the approximate parts of the computation (which may execute on the more efficient unreliable components).

Existing systems, tools, and type systems have focused on helping developers identify, separate, and reason about the binary distinction between critical and approximate parts of a computation [11, 19, 29, 43, 44, 47, 48]. However, in practice, no computation can tolerate an unbounded accumulation of soft errors — to execute acceptably, even the approximate parts of a computation must execute correctly with some minimum probability.

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 parts 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 parts of a computation, 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 with a signature `int<0.99*R(x)> f(int x)`, where `0.99*R(x)` is the reliability specification for `f`'s return value. The symbolic expression `R(x)` stands for the reliability of the parameter `x` upon entrance to `f`. This reliability specification therefore quantifies the reliability of `f`'s return value as a function of the reliability of its inputs. Specifically, this specification means that the reliability of `f`'s return value is *at least* `0.99` times `x`'s reliability upon entry.

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 [15, 19, 20, 47] – and unreliable physical memories – in which data may be written or read incorrectly with non-negligible probability [19, 29, 47]. Rely works with a *hardware reliability specification* that lists the probability with which each operation in the machine model executes correctly.

Rely 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` can fail with some probability. A developer can also write an expression `a +. b`, which is an unreliable addition of the values `a` and `b`.

1.4 Quantitative Reliability Analysis

Given a Rely program and the hardware reliability specification, Rely's analysis uses a weakest-precondition approach to generate a symbolic *reliability constraint* for each function that captures a set of conditions that is sufficient to ensure that the function satisfies its reliability specification when executed on the underlying unreliable hardware platform. Conceptually, for each result that a function produces, these conditions conservatively approximate the reliability of the least reliable path that the program may take to compute the result. The analysis computes the probability that all steps along this path evaluate reliably and checks if this probability exceeds the probability specified in the developer-provided reliability specification.

One of the core challenges in designing Rely's analysis is dealing with unreliable computation that influences the execution of control flow constructs.

Conditionals. Unreliable computation of the boolean condition of an `if` statement introduces uncertainty into branch that the `if` statement may take – i.e., it can execute either the "then" or "else" branch when the opposite branch should have been executed. If the computation within the branches can update variables, then the reliability of the condition itself must be factored into the reliability of each such variable to account for a missing or an incorrect update of the variable due to an incorrect execution of the `if` statement.

Rely's analysis infers the set of variables that may be updated on either branch of an `if` statement and incorporates the reliability of the `if` statement's condition when checking if the reliabilities of these variables satisfy their specifications.

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 conservative static reliability of such a variable is zero.

To provide specification and verification flexibility, Rely provides two constructs: statically *unbounded* `while` loops and statically *bounded* `while` loops. 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, Rely's analysis checks if the reliability of each variable modified within the loop depends on the number of iterations of the loop (i.e., it has a loop-carried dependence and is unreliably modified), and if so the analysis conservatively sets the variable's reliability to zero.

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

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

Figure 1: Rely's Language Syntax

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.

1.5 Contributions

This paper presents the following contributions:

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

Language and Semantics. We present Rely, an imperative language that allows developers to write programs that use unreliable arithmetic/logical operations and allocate data in unreliable memory regions. Rely also enables developers to write quantitative reliability specifications for the results of a program.

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 operation (arithmetic/logical or memory read/write) 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 describe and constrain 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 constraint 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 constraint.

The validity problem for predicates generated by the analysis has an injective mapping to the conjunction of two validity problems: one in the theory of real-closed fields and one in the theory of set inclusion constraints. This problem is therefore decidable and – given the form of the generated predicates – checkable in linear time.

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. Using a quantitative hardware reliability specification derived from previous unreliable hardware architecture projects, we use Rely's analysis to obtain probabilistic reliability bounds for these computations.

2. Example

Figure 1 presents a selection of Rely's syntax. 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 core component of the motion estimation algorithm from the x264 video encoder [54].

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.

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 the next frame, increasing the size of the resulting encoding. However, previous studies on soft error injection [15] and more aggressive transformations like loop

```
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

perforation [32, 50] have demonstrated that the quality of the final result of the program is only slightly affected by the perturbation of this computation.

2.1 Reliability and Memory Region 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 and *memory region specifications* for each of its arguments.

Reliability Specification. Rely's reliability specifications express the reliability of the function's results as a function of the reliabilities of its inputs. For example, 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` on the entry to the function.

Rely uses the `R(pblocks, cblock)` notation to denote the joint reliability of the input parameters. 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`. Therefore, if the contents of `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 as references 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.

Memory Region Specification. Each parameter declaration also specifies the memory region in which the data of the array is allocated. Memory regions correspond to the physical partitioning of memory at the hardware level into regions of varying reliability. Here the elements of both `pblocks` and `cblock` are allocated in the unreliable memory region `urel`. A developer can specify the memory region of local variables in an analogous manner.

2.2 Unreliable Computation

Lines 9-12 declare the local variables of the function. Like parameter declarations, variable declarations specify the memory region of each variable. In Rely, if a developer does not provide a memory region (as is the case for the variables `i`, `j`, and `k` on Line 12) then the variables are allocated in a default, fully reliable memory region.

The body of the function computes the value (`minssd`) and 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 most similar to the currently encoded block.

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.

```
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 3: Hardware Reliability Specification

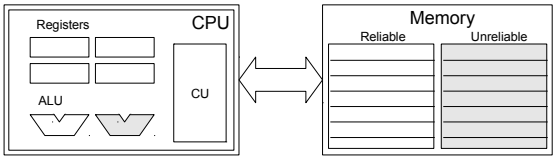

Figure 4: Machine Model Illustration. Gray boxes represent unreliable components

2.3 Hardware Reliability Specification

Rely's analysis works with a *hardware reliability specification* that specifies the reliability of arithmetic/logical and memory operations. Figure 3 presents a hardware reliability specification that we have created using results from existing computer architecture literature [18, 29]. The specification consists of a set of individual specifications for the reliability – the probability of a correct execution – of arithmetic operations (e.g., `+.`) and read/write operations on memory regions.

Figure 4 illustrates the conceptual machine model for our hardware reliability specifications. The machine 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 4 by physically separate reliable and unreliable functional units, but this distinction can be achieved through other mechanisms, such as dual-voltage architectures [19].

Due to soft errors, an execution of an arithmetic/logical operation on an unreliable functional unit produces the correct result with only some probability (as opposed to computing the correct result for all executions). These soft errors may occur due to, for example, power variations within the ALU's combinatorial circuits or particle strikes. For the presented hardware model, we use have used the probability of failure of an unreliable multiplication operation from [18, Figure 9]. We assume the same error rate for the remaining ALU operations.

As is provided by existing computer architecture proposals [19, 47], 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. The memory regions listed in the hardware specification are exactly those that can be referred by a parameter or variable's memory region specification. Rely supports machine models that have an arbitrary number of memory partitions (each potentially of different reliability), but for simplicity we have partitioned memory into two regions: reliable and unreliable. Note that the specification for a memory region includes two probabilities: the probability that a read is successful (`rd`) and the probability that a write is successful (`wr`).

Due to soft errors, reading (writing) from unreliable memory yields (stores) the correct value with only some probability. These soft errors may occur due to, for example, decreased refresh rate of DRAM cells or particle strikes. For the presented hardware model, we have used the probability of a bit flip in a memory cell from [29, Figure 4] and then extrapolated that value to produce the probability of a bit flip within a 32-bit word.

2.4 Reliability Analysis

Rely's analysis system takes as inputs a program written in Rely and a hardware reliability specification. For each function, the analysis verifies that the reliability of the function's return value and the output reliabilities of array arguments under the hardware model always exceed the developer's reliability specifications.

The analysis works by generating constraints that characterize the set of all valid specifications for the function. The analysis then verifies that the developer-provided specifications satisfy these constraints.

Reliability Approximation. The analysis generates constraints according to a conservative approximation of the semantics of the function. Specifically, it characterizes the reliability of an output of a function according to the probability that the statements in the function that compute the output execute fully reliably.

To illustrate the intuition behind this design point, consider execution of an assignment statement $x = e$. The reliability of x after this statement is the probability that x contains the same value in an unreliable execution as in the fully reliable execution. There are two ways that x can contain the same value in both the reliable and unreliable executions: 1) the reliable and unreliable executions have the same values for all variables referenced in e and both the evaluation of e and the assignment to x encounter no faults, or 2) the unreliable execution encounters faults during the evaluation of e and the assignment to x, and by chance, the value assigned to x is the same as in the reliable execution.

Our analysis conservatively approximates the reliability of the assignments by only considering the first scenario. This design point simplifies our reasoning to the task of computing the probability that an execution is fully reliable

as opposed to reasoning about all possible executions of an expression, a variable assignment, or by extension, a program. As a consequence, the analysis requires as input only the probability that an arithmetic/logical operation or a memory operation execute correctly. It does not require a full characterization of how soft errors manifest during the execution of an operation, such as a probability that power variation or a particle strike affects a particular bit in a word.

Constraint Generation. The analysis of a function generates a reliability constraint that conservatively bounds the set of valid specifications for the function. As a weakest preconditions generator, the analysis starts at the end of the function from an initial constraint that must be true when the function returns. The analysis starts with the constraint that the actual reliability of each function output must be at least that given in its specification. The analysis then works backwards producing a new constraint such that if the new constraint holds before a sequence of analyzed statements, then the initial constraint holds at the end of the function, after execution of the statements.

For `search_ref` the analysis starts at the `return` statement. It produces three constraints, one for each function output. Each constraint has 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 *out*, r is a numerical value between 0 and 1, and $\mathcal{R}(X)$ the joint reliability of the set of variables X.

The constraint for the return value is $A_{ret} \leq (1 - 10^{-7}) \cdot \mathcal{R}(\texttt{minblock})$. This constraint means that the reliability specified by the developer must be less than or equal to the reliability of reading `minblock` from unreliable memory – which is $1 - 10^{-7}$ (according the hardware reliability specification) – multiplied by the probability that the address where `minblock` is located contains the correct value. The constraints for `pblocks` and `cblock` are $A_{\texttt{pblocks}} \leq \mathcal{R}(\texttt{pblocks})$ and $A_{\texttt{cblock}} \leq \mathcal{R}(\texttt{cblock})$, respectively.

Constraint Propagation. The analysis next propagates the constraints backwards through the function, updating the constraints to reflect the effects of a program's statements on the reliability of the function's results. For example, consider the statement `minblock = i` on Line 31. The analysis will update the constraint $A_{ret} \leq (1-10^{-7}) \cdot \mathcal{R}(\texttt{minblock})$ to reflect the dependence of `minblock` on `i`. At a first approximation, the new constraint is $A_{ret} \leq (1 - 10^{-7}) \cdot \mathcal{R}(\texttt{i})$ because reads from `i` are fully reliable – as it is allocated in reliable memory – and writes to `minblock` are fully reliable – as specified in the hardware specification. However, the analysis must also consider additional dependencies for this update because it occurs within an `if` statement (Line 29) and a loop (Line 14) and is therefore dependent on the reliability of the expressions that control the execution of those constructs.

Conditionals. The condition of an `if` statement may encounter faults that force the execution of the program down a different path. For example, the `if` statement on Line 29 uses an unreliable comparison operation on `ssd` and `minssd`, which both reside in unreliable memory. Because this expression is unreliable, the reliability of `minblock` when modified on Line 31 must also depend on the reliability of this conditional.

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

The analysis first constructs a constraint that is a conjunction of two constraints: one that must hold for the "then" branch and one that must hold for the "else" branch. The analysis of the `minblock = i` statement in the "then" branch will add the dependence on ℓ_{29} by transforming the constraint $A_{ret} \leq (1 - 10^{-7}) \cdot \mathcal{R}(\texttt{minblock})$ to the constraint $A_{ret} \leq (1 - 10^{-7}) \cdot \mathcal{R}(\ell_{29}, \texttt{i})$, which includes the dependence of `minblock` on both ℓ_{29} and `i`. The "else" branch of the conditional is empty – therefore its constraint is $A_{ret} \leq (1 - 10^{-7}) \cdot \mathcal{R}(\texttt{minblock})$, which reflects the fact that `minblock` keeps its previous values.

As a final step, when the analysis leaves the scope of the conditional, it will again transform the constraint to include the direct dependence of the control flow variable on the reliability of the `if` statement's condition. Namely, the new constraint for the "then" branch will be $A_{ret} \leq (1 - 10^{-7})^2 \cdot \mathcal{R}(\texttt{i}, \texttt{ssd}, \texttt{minssd})$ where the reliability of the operation `<.` has been incorporated along with the reliabilities of the variables `ssd` and `minssd`.

While `minblock` is not modified in the "else" branch, the probability of executing the "else" branch instead of the "then" branch – which would incorrectly modify `minblock` – depends on the reliability of the conditional. The analysis therefore updates the constraint for the "else" branch include this dependence, producing the constraint $A_{ret} \leq (1 - 10^{-7}) \cdot \mathcal{R}(\texttt{minblock}, \texttt{ssd}, \texttt{minssd})$. In general, if a variable is modified on either branch of a conditional, then its reliability depends on the reliability of the condition.

Loops. This update to `minblock` also occurs within a loop. As with an `if` statement, the condition of a loop can be unreliable and therefore, the analysis uses a control flow variable to incorporate the reliability of the loop's condition.

A key difficulty with reasoning about the reliability of variables modified within a loop is the fact that the reliability of a variable that is updated unreliably and has a loop-carried dependence monotonically decreases as a function of the number of loop iterations. To enable straightforward reasoning about loops, Rely offers two types of loop constructs:

- **Unbounded Loops.** An unbounded loop is a loop that does not have a compile-time bound on the number of iterations. For these loops, Rely's analysis conservatively sets the reliability of modified variables to 0, as the reliability of variables modified within the loop can, in principle, decrease arbitrarily.
- **Bounded Loops.** A bounded loop is a loop that has a compile-time bound on the number of loop iterations. The `repeat` loop on Line 14 is a bounded loop that iterates `nblocks` times – where `nblocks` is a compile-time constant – and therefore decreases the reliability of any modified variables `nblocks` times. Because the reliability decrease is bounded, Rely's analysis uses unrolling to reason about the effects of a bounded loop.

After unrolling a single iteration of the loop that begins at Line 14 and performing analysis on its body, Rely produces $A_{ret} \leq (1-10^{-7})^{2564} \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock}, \texttt{i}, \texttt{ssd}, \texttt{minssd}, \ell_{14})$ as a constraint, where ℓ_{14} is the loop's control flow variable.

Specification Checking. Once the analysis reaches the beginning of the function after fully unrolling the loop on Line 14, it has a set of constraints that bound the set of valid specifications as a function of the reliability of the parameters of the function. For `search_ref`, the analysis generates the constraints $A_{ret} \leq 0.994885125493 \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock})$, $A_{\texttt{pblocks}} \leq \mathcal{R}(\texttt{pblocks})$, and $A_{\texttt{cblock}} \leq \mathcal{R}(\texttt{cblock})$ meaning that any specifications that can be substituted for A_{ret}, $A_{\texttt{pblocks}}$, and $A_{\texttt{cblock}}$ and still satisfy the inequality are valid specifications. In the example, these specifications are $0.99 \cdot \mathcal{R}(\texttt{pblocks,cblock})$, $\mathcal{R}(\texttt{pblocks})$, and $\mathcal{R}(\texttt{cblock})$, respectively.

The constraint checker verifies the validity of each predicate by checking that 1) the numerical constant in the specification is less than or equal to the numerical constant computed by the analysis and 2) the set of variables in the reliability factor computed by the analysis is a subset of the set of variables in the reliability factor provided by the specification. For the predicate on the return value, the checker verifies that 1) $0.99 \leq 0.994885125493$ and 2) both the specification and the reliability factor generated by the analysis reference the `pblocks` and `cblock` variables. For the predicates for the `pblocks` and `cblock` variables, the checker verifies that 1) the numerical constant in the specification and that generated by the analysis are the same (equal to 1.0) and 2) the set of variables in the reliability factor generated by the analysis are the same as that in the specification ($\{\texttt{pblocks}\}$ and $\{\texttt{cblock}\}$, respectively).

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 3.5 we provide big-step definitions that specify the probability of an entire program execution.

3.1 Design

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) of a function 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. The stack is allocated in a reliable memory region of the concrete machine.

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

To prevent out-of-bounds memory accesses that may occur as consequence of an unreliable array index computation, each array read and write includes a bounds check. These bounds check computations execute reliably.

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 instructions are 1) stored in reliable memory and 2) fetched and decoded reliably – as is supported by existing unreliable computer architectures [19, 47].

3.2 Preliminaries

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

Arithmetic/logical operations *iop*, *cmp*, and *lop* include both reliable and unreliable versions of each integer, comparison, and logical operation. For each reliable operation, the probability that the operation executes correctly is 1.0. We define the maps $\mathit{rd} : M \rightarrow M_{\mathit{op}}$ and $\mathit{wr} : M \rightarrow M_{\mathit{op}}$, which provide the memory operations for reads and writes (respectively) on memory regions ($m \in M$). M is the set of all memory regions given in the hardware reliability specification.

We use 1_ψ to denote the hardware reliability specification for fully reliable hardware in which all arithmetic/logical and memory operations have reliability 1.0 – i.e., $\forall \mathit{op}.\, 1_\psi(\mathit{op}) = 1$.

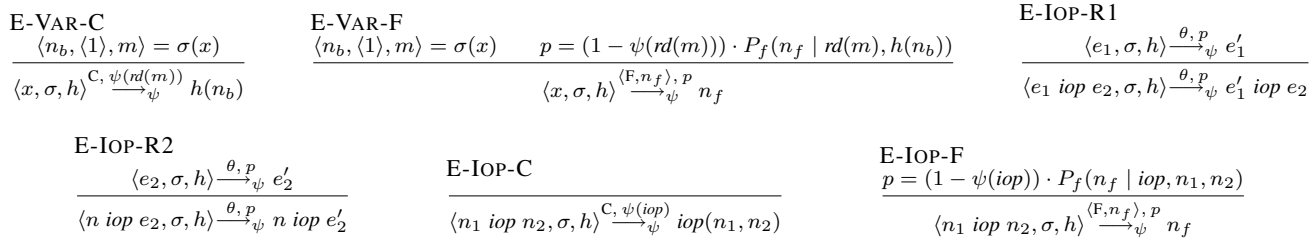

Figure 5: Dynamic Semantics of Integer Expressions

Unreliable Result Distribution. An unreliable result distribution $P_f(n_f \mid op, n_1, \ldots, n_k)$ models the manifestation of a soft error during an incorrect execution of an operation. Specifically, it provides the probability that an incorrect execution of an operation *op* on operands $n_1, \ldots, n_k$ produces a value n_f when it encounters a soft error. An unreliable result distribution is a probability mass function (conditioned on the operation and its operands) with the additional constraint that it assigns zero probability to the correct result of an operation.

An unreliable result distribution function is inherently tied to the properties of the underlying hardware. We use these distributions only to specify the dynamic semantics of a program. Rely's reliability analysis, on the other hand, does not require these distributions because their shape is independent of the reliability of an operation (i.e., they ascribe zero probability to the correct result). These distributions therefore do not need to be explicitly specified in the hardware reliability specification.

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. An address is a finite non-negative integer (Loc) and each value is an integer. References describe the location, dimensions, and memory region of scalars and arrays that are allocated in the heap. A base address and the components of a dimension descriptor are finite non-negative integers. In the case of scalars, the dimension descriptor is always the single-dimension, single-element descriptor $\langle 1\rangle$.

For notational convenience, we use projections π_{base} and π_{dim} to select the base address and the dimension descriptor of the array, respectively.

Frames, Stacks, and Heaps. A *frame* σ is an element of the domain $\Sigma = \mathrm{Var} \to \mathrm{Ref}$ which is the set of finite maps from program variables to references. A *stack* $\delta \in \Delta$ is a list of frames. Stacks are constructed by the standard operator :: for list construction. A *heap* $h \in H = \mathrm{Loc} \to \mathbb{Z}$ is a finite map from addresses to integer values stored at these addresses. Floating point and boolean values are stored in memory as encoded integers.

For notational convenience, we also define an *environment* $\varepsilon \in \mathrm{E} = \Delta \times H$ to be a pair $\langle \delta, h\rangle$ consisting of a stack δ and a heap h. We use the projections π_{heap} and π_{frame} to select the heap and the top frame of the stack from an environment variable, respectively.

3.3 Semantics of Expressions

Figure 5 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 $\theta \in \{\mathrm{C}, \langle \mathrm{F}, n_f\rangle\}$ denotes whether the transition corresponds to a correct (C) or faulty ($\langle \mathrm{F}, n_f\rangle$) evaluation of that step. For a faulty transition, n_f represents the value that the fault introduced in the semantics of the operation.

Variable Reference. 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 unreliable result distribution that gives the probability that a failed memory read operation returns a value n_f instead of the true stored value $h(n_b)$. As aforementioned, the shape of this distribution comes from the properties of the underlying unreliable hardware, but this shape does not need to be specified to support Rely's analysis. We use this distribution only to support a precise formalization of the dynamic semantics of a program.

Binary Integer Operation. A binary integer operation e_1 *iop* e_2 evaluates its two subexpressions e_1 and e_2 and returns the result of the operation. An operation may experience a fault while evaluating e_1, evaluating e_2, or while applying the operation to the reduced values of the two subexpressions.

Evaluation of an integer operation proceeds by first reducing e_1 [E-IOP-R1] to the value n_1 and then reducing e_2 [E-IOP-R2] to n_2. Once each subexpression has been fully reduced, evaluation continues with one of two possibilities:

- Correct [E-IOP-C]. The integer operation evaluates correctly with probability $\psi(\mathit{iop})$ and successfully returns the result of applying the operation on the reduced values of the two subexpressions.
- Faulty [E-IOP-F]. The integer operation experiences a fault with probability $(1 - \psi(\mathit{iop}))$ and returns another integer n_f with probability $P_f(n_f \mid \mathit{iop}, n_1, n_2)$. As with faulty variable references, P_f is a distribution over possible values of n_f given a fault while executing the operation *iop* on n_1 and n_2.

We elide a presentation of the dynamic semantics of boolean and floating point expressions, which follows closely that of the dynamic semantics of integer expressions.

3.4 Semantics of Statements

Figure 6 presents the scalar manipulation 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 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 additional ability to encounter faults during evaluation.

Integer Variable Declarations. An integer variable declaration $\texttt{int}\ x = e\ \texttt{in}\ m$ declares a new variable to be used in a function. The semantics first reduces the declaration's initializer [E-DECL-R], then allocates a new variable in the heap using the memory allocator function *new* [E-DECL]. The allocator *new* is a potentially non-deterministic function. It takes a heap h, a memory region m, and a dimension descriptor (in this case a single-dimension, single-element array) 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.

To express the semantics of potentially non-deterministic memory allocators, we define the probability distribution $P_m(n_b, h' \mid h, m, n_d)$, which returns the probability that a memory allocator returns a location n_b and an updated heap h', given the initial heap h, a memory region m, and a dimension descriptor n_d. Note that this distribution is inherently tied to the semantics of the memory allocator and is defined only to support a precise formalization of the dynamic semantics of a program, and therefore does not need to be specified by the developer.

The semantics finally sets the new reference for x within the current frame and then reduces to an assignment $x = n$, ensuring that x is appropriately initialized.

Integer Variable Assignment. An assignment statement $x = e$ assigns the result of evaluating e to the variable x. Evaluation of the statement proceeds by first reducing e to an integer n [E-ASSIGN-R]. After reducing e, there are then two possibilities for evaluation:

- Correct [E-ASSIGN-C]. The write to the variable succeeds with probability $\psi(\mathit{wr}(m))$ and then n is stored into the memory address of x.
- Faulty [E-ASSIGN-F]. The write to the variable encounters a fault with probability $1 - \psi(\mathit{wr}(m))$ and instead stores an arbitrary value n_f into x with probability $P_f(n_f \mid \mathit{wr}(m), n)$, where P_f is an unreliable result distribution given a fault during writing the value n to memory.

Conditionals. Evaluation of an `if` condition $\texttt{if}_\ell\ b\ s_1\ s_2$ proceeds by first reducing b to a boolean value *true* or *false* [E-IF]. After fully reducing the boolean expression, the statement transfers control either to s_1 (if the condition is *true* [E-IF-TRUE]) or to s_2 (if the condition is *false* [E-IF-FALSE]). [E-IF-TRUE] and [E-IF-FALSE] always transfer control to s_1 and s_2, respectively.

We have annotated each conditional with a unique *control flow variable* ℓ. This control flow variable records whether the condition evaluated to *true* ($\ell = 1$) or *false* ($\ell = 0$), respectively. Control flow variables are latent variables that are not visible to the developer. Control variables are instead additional semantic instrumentation that Rely uses in its analysis of a program to reason about the reliability of program variables that are modified under conditionals. Specifically, the reliability of those variables depends on the reliability of the conditional's expression (Section 2.4 and Section 5). To support the dynamic semantics, control flow variables are implicitly declared and allocated in reliable memory on entry to a function.

Sequential Composition. Evaluation of a sequential composition of statements $s_1\ ;\ s_2$ proceeds by first reducing s_1 to a `skip` statement [E-SEQ-R1] and then transferring control to s_2 [E-SEQ-R2].

Unbounded while loops. The semantics of an unbounded `while` loop, $\texttt{while}_\ell\ b\ s$, is similar to that in traditional presentations. The statement reduces to an `if` conditional consisting of a test of b and, if true, an execution of the loop body s followed by another instance of the `while` loop [E-WHILE].

Bounded while and repeat loops. Bounded `while` loops, $\texttt{while}_\ell\ b : n\ s$, (and $\texttt{repeat}_\ell\ n\ s$) are `while` loops that execute at most n iterations. The semantics of the bounded `while` is similar to that for unbounded loops except that 1) it first initializes the control flow variable ℓ to zero, 2) before

E-DECL-R

$$\frac{\langle e, \sigma, h\rangle \xrightarrow{\theta, p}_{\psi} e'}{\langle \texttt{int}\ x = e\ \texttt{in}\ m, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta, p}_{\psi} \langle \texttt{int}\ x = e'\ \texttt{in}\ m, \langle \sigma :: \delta, h\rangle\rangle}$$

E-DECL

$$\frac{\langle n_b, h'\rangle = \mathit{new}(h, m, \langle 1\rangle) \qquad p_m = P_m(n_b, h' \mid h, m, \langle 1\rangle)}{\langle \texttt{int}\ x = n\ \texttt{in}\ m, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\mathrm{C}, p_m}_{\psi} \langle x = n, \langle \sigma[x \mapsto \langle n_b, \langle 1\rangle, m\rangle] :: \delta, h'\rangle\rangle}$$

E-ASSIGN-R

$$\frac{\langle e, \sigma, h\rangle \xrightarrow{\theta, p}_{\psi} e'}{\langle x = e, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta, p}_{\psi} \langle x = e', \langle \sigma :: \delta, h\rangle\rangle}$$

E-ASSIGN-C

$$\frac{\langle n_b, \langle 1\rangle, m\rangle = \sigma(x) \qquad p = \psi(wr(m))}{\langle x = n, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\mathrm{C}, p}_{\psi} \langle \texttt{skip}, \langle \sigma :: \delta, h[n_b \mapsto n]\rangle\rangle}$$

E-ASSIGN-F

$$\frac{\langle n_b, \langle 1\rangle, m\rangle = \sigma(x) \qquad p = (1 - \psi(wr(m))) \cdot P_f(n_f \mid wr(m), n)}{\langle x = n, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\langle \mathrm{F}, n_f\rangle, p}_{\psi} \langle \texttt{skip}, \langle \sigma :: \delta, h[n_b \mapsto n_f]\rangle\rangle}$$

E-IF

$$\frac{\langle b, \sigma, h\rangle \xrightarrow{\theta, p}_{\psi} b'}{\langle \texttt{if}_\ell\ b\ s_1\ s_2, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta, p}_{\psi} \langle \texttt{if}_\ell\ b'\ s_1\ s_2, \langle \sigma :: \delta, h\rangle\rangle}$$

E-IF-TRUE

$$\frac{}{\langle \texttt{if}_\ell\ \mathit{true}\ s_1\ s_2, \varepsilon\rangle \xrightarrow{\mathrm{C}, 1}_{\psi} \langle \ell = 1\ ;\ s_1, \varepsilon\rangle}$$

E-IF-FALSE

$$\frac{}{\langle \texttt{if}_\ell\ \mathit{false}\ s_1\ s_2, \varepsilon\rangle \xrightarrow{\mathrm{C}, 1}_{\psi} \langle \ell = 0\ ;\ s_2, \varepsilon\rangle}$$

E-SEQ-R1

$$\frac{\langle s_1, \varepsilon\rangle \xrightarrow{\theta, p}_{\psi} \langle s_1', \varepsilon'\rangle}{\langle s_1\ ;\ s_2, \varepsilon\rangle \xrightarrow{\theta, p}_{\psi} \langle s_1'\ ;\ s_2, \varepsilon'\rangle}$$

E-SEQ-R2

$$\frac{}{\langle \texttt{skip}\ ;\ s_2, \varepsilon\rangle \xrightarrow{\mathrm{C}, 1}_{\psi} \langle s_2, \varepsilon\rangle}$$

E-WHILE

$$\frac{}{\langle \texttt{while}_\ell\ b\ s, \varepsilon\rangle \xrightarrow{\mathrm{C}, 1}_{\psi} \langle \texttt{if}_\ell\ b\ \{s\ ;\ \texttt{while}_\ell\ b\ s\}\ \{\texttt{skip}\}, \varepsilon\rangle}$$

E-WHILE-N

$$\frac{}{\langle \texttt{while}_\ell\ b : n\ s, \varepsilon\rangle \xrightarrow{\mathrm{C}, 1}_{\psi} \langle \ell = 0\ ;\ \texttt{while}_\ell\ (\ell < n\ \&\&\ b)\ \{s\ ;\ \ell = \ell + 1\}, \varepsilon\rangle}$$

E-REPEAT

$$\frac{}{\langle \texttt{repeat}_\ell\ s\ n, \varepsilon\rangle \xrightarrow{\mathrm{C}, 1}_{\psi} \langle \texttt{while}_\ell\ \mathit{true} : n\ s, \varepsilon\rangle}$$

Figure 6: Dynamic Semantics of Statements

each iteration, the loop checks if $\ell < n$ and exits if otherwise, and 3) after each iteration, it increments ℓ by one. All these operations are done reliably.

3.4.1 Arrays and Functions

We have included the semantics for arrays and functions in the paper's Appendix[1].

3.5 Big-step Notations

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

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

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

The big-step trace semantics for statements is the reflexive transitive closure of the small-step execution relation that records a trace of the execution. A *trace* τ is the sequence of all small-step transition labels. The *probability of a trace*, p, is the product of the probabilities of each small-step transition.

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

$$\langle s, \varepsilon\rangle \overset{p}{\Longrightarrow}_{\psi} \varepsilon' \textit{ where } p = \sum_{\tau} p_i \textit{ such that } \langle s, \varepsilon\rangle \overset{\tau, p_i}{\Longrightarrow}_{\psi} \varepsilon'$$

[1] Appendix is available at `http://groups.csail.mit.edu/pac/rely/`

A big-step aggregate semantics enumerates over 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 its execution terminates in an environment ε'.

4. Semantics of Quantitative Reliability

Given our language semantics, 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)**.** $\quad \varphi \in \Phi = \mathrm{E} \to \mathbb{R}$

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

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. The unreliable environment distribution φ gives the distribution on starting environments for the unreliable execution of s whereas φ' gives the distribution on possible environments after executing s. Unreliable environment distribution functions therefore describe the probability of reaching an environment as a result of a fault.

The unreliable environment distribution φ' is specified pointwise: $\varphi'(\varepsilon'_u)$ 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 of reaching ε'_u from any starting environment $\varepsilon_u \in \mathrm{E}$ according to the big-step aggregate semantics. We note that, in general, φ' is a subprobability measure, i.e. $\Sigma_{\varepsilon'_u}\varphi'(\varepsilon'_u) \leq 1$, since the big-step aggregate semantics (Definition 2) enumerates only over terminating traces.

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:

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

A predicate can either be the constant `true`, the constant `false`, a comparison between *reliability factors* (R_f), or a conjunction of predicates.

Reliability Factors. A reliability factor is real-valued quantity of one of the following forms:

- **Constant**. A real-valued constant r in the range $[0, 1]$.
- **Reliability Variable**. A *reliability variable* $A \in \mathrm{RVar}$ is a real-valued variable with a value in the range $[0, 1]$.
- **Joint Reliability**. A *joint reliability* $\mathcal{R}(X)$ is 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.
- **Multiplication**. A product of reliability factors.

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 7 presents the denotational semantics of reliability predicates via the semantic function $[\![P]\!]$. The denotation of a reliability predicate is the set of tuples – which each consist of a *reliability state*, an environment, and unreliable environment distribution – that satisfy the predicate. A reliability state $\nu \in \mathrm{N} = \mathrm{RVar} \rightarrow [0, 1]$ is a finite map from reliability variables to their values.

We elide a discussion of the semantics of reliability predicates themselves because they are standard and instead focus on the semantics of reliability factors.

Constants and Reliability Variables. The denotation of a reliability constant r is the constant itself. The denotation of a reliability variable A is its value within the reliability state ν – i.e, $\nu(A)$.

Joint Reliability. The joint reliability $\mathcal{R}(X)$ is the probability that the values of the variables in X in the reliable environment ε are the same as the values of these variables in an unreliable environment ε_u sampled from the unreliable environment distribution φ. We use the helper function $\mathrm{rel}(X, \varepsilon, \varphi, U)$ to define this probability.

The function $\mathrm{rel}(X, \varepsilon, \varphi, U)$ gives a lower bound on the probability that the values of the variables in X in the reliable environment ε are the same as those in an unreliable environment ε_u sampled from the distribution φ, subject to the restriction that $\varepsilon_u \in U$ (where $U \subseteq \mathrm{E}$). The function $\mathrm{rel}(\cdot)$ is defined inductively on the set of variables X. In the base case when X is the empty set, $\mathrm{rel}(\varnothing, \varepsilon, \varphi, U)$ computes the probability of observing the set of unreliable environments U given the distribution φ. Because φ is a probability mass function, this probability is equal to $\sum_{\varepsilon_u \in U} \varphi(\varepsilon_u)$.

If X is non-empty, then we define $\mathrm{rel}(\cdot)$ by selecting a variable from X, restricting the set of environments U such that the remaining environments have the same value for that variable, and then recursing on the remaining variables in X with the new restricted set of environments. There are two cases we must consider when we select a variable from X:

- **Scalar Variable.** We define the predicate $\mathrm{equiv}(\varepsilon', \varepsilon, x, i)$ which is true if and only if ε' and ε have the same value at the location obtained by adding i to the base address of x. We define $\mathrm{rel}(\{x\} \cup X, \varepsilon, \varphi, U)$ for scalar variables recursively by using the predicate equiv to construct a new set of environments U' such that $U' \subseteq U$ and all environments in U' have the same value for x as in the reliable environment ε.
- **Array Variable.** We first define the function $\mathrm{len}(a, \varepsilon)$ which computes the length of the array a given its dimension descriptor in ε. For an array variable a, we define $\mathrm{rel}(\{a\} \cup X, \varepsilon, \varphi, U)$ recursively by taking the minimum over all indices of a of the probability that the value at that index (along with the values of the remaining variables in X) have the same value. The definition accomplishes this by using a set construction that is similar to the scalar case.

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

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

$$[\![R^*]\!] \in \mathrm{N} \times \mathrm{E} \times \Phi \to \mathbb{R} \qquad [\![r]\!](\nu, \varepsilon, \varphi) = r \qquad [\![A]\!](\nu, \varepsilon, \varphi) = \nu(A) \qquad [\![R_1^* \cdot R_2^*]\!](\nu, \varepsilon, \varphi) = [\![R_1^*]\!](\nu, \varepsilon, \varphi) \cdot [\![R_2^*]\!](\nu, \varepsilon, \varphi)$$

$$[\![\mathcal{R}(X)]\!](\nu, \varepsilon, \varphi) = \mathrm{rel}(X, \varepsilon, \varphi, U) \qquad \mathrm{rel} \ \in \ \mathcal{P}(\mathrm{Var} + \mathrm{ArrVar}) \times \mathrm{E} \times \Phi \times \mathcal{P}(\mathrm{E}) \to [0, 1]$$

$$\mathrm{rel}(\varnothing, \varepsilon, \varphi, U) = \sum_{\varepsilon_u \in U} \varphi(\varepsilon_u) \qquad \mathrm{rel}(\{x\} \cup X, \varepsilon, \varphi, U) = \mathrm{rel}(X, \varepsilon, \varphi, \{\varepsilon' \mid \varepsilon' \in U \wedge \mathrm{equiv}(\varepsilon', \varepsilon, x, 0)\})$$

$$\mathrm{rel}(\{a\} \cup X, \varepsilon, \varphi, U) = \min_{0 \leq i < \mathrm{len}(a, \varepsilon)} \mathrm{rel}(X, \varepsilon, \varphi, \{\varepsilon' \mid \varepsilon' \in U \wedge \mathrm{equiv}(\varepsilon', \varepsilon, a, i)\})$$

$$\mathrm{equiv}(\varepsilon', \varepsilon, x, i) = \pi_{\mathrm{heap}}(\varepsilon')(\pi_{\mathrm{base}}(\pi_{\mathrm{frame}}(\varepsilon')(x)) + i) = \pi_{\mathrm{heap}}(\varepsilon)(\pi_{\mathrm{base}}(\pi_{\mathrm{frame}}(\varepsilon)(x)) + i)$$

Figure 7: Predicate Semantics

This definition of the reliability of an array variable is suitable for representing results of reading to and writing from a single element of an array while abstracting away the specific index of an element. Any array element $a[i]$ that a computation accesses has a reliability that is greater than or equal to $\mathcal{R}(a)$.

We can also use this definition of the reliability of an array variable to recover the reliability of the full contents of the array – i.e., the probability that all array elements have the same value in the reliable and the unreliable execution. Given $\mathcal{R}(a)$ and the length of the array $n = \mathrm{len}(a, \varepsilon)$, a lower bound on this probability is $\max\{1 - n(1 - \mathcal{R}(a)), 0\}$. This result follows by applying the probabilistic union bound – since the probability that a single array element is unreliable is $1 - \mathcal{R}(a)$, the probability that any of n array elements are unreliable is bounded from above by $\min\{n(1 - \mathcal{R}(a)), 1\}$. Then the probability that all array elements are reliable is bounded from below by the complement of the previous bound.

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 [17].

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

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

Reliability predicates and the semantics of the reliability transformer 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 semantic 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 distribution after execution of the function.

5. Reliability Analysis

For each function in a program, Rely's reliability analysis generates a symbolic *reliability constraint* with a weakest-preconditions style analysis. The reliability constraint is a reliability predicate that constrains the set of specifications that are valid for the function. Specifically, the reliability constraint is a predicate of the form $\bigwedge_{i,j} A_i \leq R_j^*$ where the reliability variable A_i is a placeholder 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 constraint is valid after an appropriate substitution of the developer-provided specifications for each A_i, then the specifications are valid for the function.

5.1 Design

The following design points reflect some of the core challenges in developing Rely's reliability analysis.

Reliability Approximation. The analysis generates constraints 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.

Control Flow. Faults during the unreliable execution of a program can also affect the path that a control flow branch takes relative to a reliable execution of a program. In particular, if the condition of an `if` statement is incorrect, then an unreliable execution may take a different branch from the reliable execution. The analysis incorporates the effect of unreliable control flow on the reliability of a variable modified within an `if` statement by including in the variable's reliability analysis a dependence on the reliability of the statement's control flow variable (Section 3.4). The reliability of an `if` statement's control flow variable is equal to the reliability of the statement's condition expression.

Loops. The reliability of a variable that has a loop-carried dependence and is unreliably modified within a loop is a monotonically decreasing function of the number of loop iterations. Because variables unreliably modified within an unbounded `while` can be – dynamically – modified an arbitrary number of times, our analysis first checks if the loop condition and the variables modified within the loop are computed reliably, and if not it sets the reliability of the values modified within an unbounded `while` loop to zero. In contrast, because bounded `while` loops have a compile-time bound on the maximum number of iterations, our analysis uses this bound to unroll the loop and compute a more precise reliability.

Arrays. We have defined the semantics of the reliability of an array (Section 4.2.1) to be similar to that of a set: the reliability of the array is the reliability of the least reliable element in the array. Our analysis of array operations therefore conservatively assumes that each array read accesses the array element with the minimum reliability and conversely, each array write decreases the reliability of the element that previously had the minimum reliability.

5.2 Constraint Generation

Figure 8 presents a selection of Rely's reliability constraint generation rules in a weakest precondition style. The generator takes as input a statement s, a predicate Q, a set of control flow variables C, and (implicitly) the maps Λ and Γ. The generator produces as output a predicate 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 that implied by its specification. Our analysis captures this property by setting the initial Q for the body of a function to

$$\bigwedge_{a_i \in \mathit{params}(f)} A_i \leq \mathcal{R}(a_i)$$

where A_i is the reliability variable that holds the place of the developer-provided specification for the array a_i. 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 constraint generator works backwards through the program in a weakest-preconditions style, it generates a new constraint that – if valid at the beginning of the program – ensures that this initial Q true.

5.2.1 Preliminaries

Auxiliary Maps. The map Λ 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 Γ is a unique map from the observable outputs of a function – namely, the return value and arrays passed as parameters – to reliability variables (Section 4.2). Specifically, for each observable output of a function we allocate a unique reliability variable and store that mapping within Γ.

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 $c \cdot \mathcal{R}(\{x, z\})[d \cdot \mathcal{R}(\{y\} \cup X)/\mathcal{R}(\{x\} \cup X)]$ is the expression $c \cdot d \cdot \mathcal{R}(\{y, z\})$.

A parameterized substitution $e_0[e_2/e_1 : x \in X]$ takes a set of literals X. The parameter x is a name that e_1 and e_2 can reference. For a single literal $x' \in X$ the substitution binds the name x to x' in e_1 and e_2, producing expressions $e_1' = e_1[x'/x]$ and $e_2' = e_2[x'/x]$, and then performs the basic substitution $e_0[e_2'/e_1']$. The parameterized substitution chains $|X|$ basic substitutions, one for each element of X. The order of basic substitutions is non-deterministic.

5.2.2 Reasoning about Expressions

The topmost part of Figure 8 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 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.[2]

[2] The rules for boolean and relational operations (which we elide) are defined analogously.

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

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

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

Figure 8: Reliability Constraint Generation

5.2.3 Generation Rules for Statements

We next present the constraint generation rules for Rely statements. As in a weakest-precondition generator, the generator 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 that specified by the developer. To verify this, the analysis rule generates the addition constraint $\Gamma(f) \leq \rho_1(e) \cdot \mathcal{R}(\rho_2(e))$. This constrains the reliability of the return value, where $\Gamma(f)$ is the reliability variable allocated for the return value of the function.

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 $Q[(\psi(wr(m)) \cdot \rho_1(e) \cdot \mathcal{R}(\rho_2(e) \cup X \cup C))/\mathcal{R}(\{x\} \cup X)]$ binds each reliability factor in which x occurs – $\mathcal{R}(\{x\} \cup X)$ – and replaces the factor with a new reliability factor $\mathcal{R}(\rho_2(e) \cup X \cup C)$ where $\rho_2(e)$ is the set of variables referenced by e and C is the set of current control flow variables. Note that x may have the wrong value if a fault forced the program to take a different control path. The analysis accounts for this by incorporating the reliability of the current control flow variables into the substituted reliability factor.

The substitution also multiplies the reliability factor by $\rho_1(e) \cdot \psi(wr(m))$, which is the probability that e evaluates fully reliably and its value is reliably stored into the memory location for x.

Array loads and stores. The reliability of a load statement $x = a\texttt{[}e_1, \ldots, e_n\texttt{]}$ 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 $a\texttt{[}e_1, \ldots, e_n\texttt{]} = e$ depends on the reliability of evaluating the indices $e_1, \ldots, e_n$ and the reliability of evaluating the source expression e.

Note that the rule for array stores duplicates the predicate Q and only performs a substitution on one copy of Q. This duplication corresponds to the two cases that the analysis must reason about: the minimum reliability of the array is either the reliability of e – as it is now stored into the array – or it is the reliability of a previously written element. This duplication ensures that Q must also hold for all previously written elements and, specifically, ensures that Q must hold individually for every write to the array over its lifetime.

Skip and Sequence. The analysis for sequences and skip statements operates in a standard manner.

Conditional. For an execution of a conditional statement $\texttt{if}_\ell\ b\ s_1\ s_2$ to satisfy a predicate Q, then execution of each branch must satisfy Q. The rule implements this by forming a conjunction of the results of recursive invocations of the analysis on each branch.

The unreliable execution of a conditional can, in general, cause the execution to take a wrong branch and affect the reliability of the variables modified in the loop. The analysis uses control flow variables ℓ to express the dependence between the reliability of variables modified in the loop and the reliability of the condition b. The analysis performs three steps to represent the effects of unreliable control flow.

Before it analyzes the branches of the conditional, the analysis adds the conditional's control flow variable ℓ to the set of active control flow variables C. The variables in C are added to the joint reliability factors in any assignment within the conditional, reflecting the fact that the reliability of the assignment depends on the reliability of the conditional taking the correct branch.

After it computes the reliability predicates for the branches s_1 and s_2, the analysis additionally marks the reliability factors of variables that are modified only in a single branch. The function $modset(s)$ produces the set of variables that can be modified by the statement s (including modifications by any nested conditionals, loops, or function calls). The substitution appends the label ℓ to reliability factors that contain any variable from the set of variables modified within any of the branches.

As a final step, the analysis substitutes the reliability of the conditional's boolean expression for the reliability of its control flow variable.

Bounded `while` and `repeat`. Bounded `while` loops, $\texttt{while}_\ell\ b : n\ s$, and `repeat` loops, $\texttt{repeat}\ \ n\ \ s$, execute their bodies at most n times. Execution of such a loop therefore satisfies Q only 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 itself.

Unbounded `while`. For clarity of presentation, we present our analysis for unbounded `while` loops in Section 5.2.4.

Declarations. The analysis handles declarations in a similar way to assignments. The primary departure is that declarations do not occur under control flow constructs and, therefore, the current control flow variables need not be incorporated in to the analysis of these statements.

Function calls. We present the rule for function calls in the Appendix for clarity of presentation. In short, the analysis 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.

5.2.4 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. A 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 our 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 an 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.
- **Control Flow:** 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.
- **Conditional:** an `if` statement's control flow variable is named y and x occurs within the statement's conditional expression; this condition captures the dependence of the control flow variable on the conditional's expression. We classify such an edge as reliable if every operation in the conditional's expression is reliable. Otherwise, we mark the edge as unreliable.

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 paths to x in the dependence graph contain only reliable edges.

Fixpoint Analysis. Given a set of reliable variables X_r, the analysis next splits the current constraint Q into two parts. For each predicate $A \leq r \cdot \mathcal{R}(X)$ in Q, the analysis checks if the property $\forall x \in X.\, x \notin \text{modset}(s) \Rightarrow x \in X_r$ holds. 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 constraint Q_r and conjoins the remaining predicates to create the constraint Q_u.

The analysis next takes the constraint Q_r and iterates the function $RC_\psi(\ell = b\ ;\ s, Q_r, C \cup \{\ell\})$ until it reaches a fixpoint. The predicate Q_r' that results from this translates Q_r to constrain the set of valid specifications using the dependencies of each variable that occurs in Q_r.

Lemma 1 (Termination)**.** *Iteration of the function $RC_\psi(\ell = b\ ;\ s, Q_r, C \cup \{\ell\})$ terminates if each variable in each predicate of Q_r is updated reliably or not updated at all.*

Final Constraint. In the last step, the analysis produces its final constraint by conjoining Q_r' with the predicate $Q_u[0/\mathcal{R}(\{x\} \cup X) \ : \ x \in \text{modset}(s)]$, where the second predicate sets the reliability of variables that are updated unreliably to zero.

Using the dependence graph, our analysis sets the reliability to zero of only those variables that are updated unreliably within a loop. For reliably updated variables, the analysis computes their reliability to be at least as reliable as their dependencies on entry to the loop.

5.2.5 Properties

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

Theorem 1 (Soundness)**.** $\nu, \psi \models \{RC_\psi(s, Q, \emptyset)\}\ s\ \{Q\}$

This theorem property states that if an environment ε and unreliable environment distribution φ satisfy a generated constraint and the paired execution of s yields an environment ε' and an unreliable environment distribution φ', then ε' and φ' satisfy Q. Alternatively, s transforms the constraint generated by our analysis to Q. We present a proof sketch for this theorem in the Appendix.

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 against the constraint produced by the constraint generator. The analysis substitutes each reliability variable $\Gamma(\cdot)$ with the corresponding reliability specification. Each specification has the form $r \cdot R(X)$ (Figure 1). After substitution, the constraint is therefore a conjunction of predicates of the form

$$r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$$

A set of developer-provided specifications is therefore valid if the predicate is valid after substituting each reliability variable with the reliability specification the developer provided for that variable's associated function output.

A useful property for solving these inequalities is the ordering of reliability factors.

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

As a consequence of ordering of reliability factors, we can directly check the validity of each predicate in a constraint.

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

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 can be checked efficiently. In addition, because the predicates are mutually independent, we can check the validity of the constraint as a whole by checking the validity of each predicate in turn.

5.4 Implementation

We implemented the parser for the Rely language, the constraint generator, and the constraint checker in OCaml. The implementation consists of 2500 lines of code. The analysis can operate on numerical or symbolic hardware reliability specifications. Our implemented analysis performs simplification transformations after every constraint generator step to identify and remove duplicates and trivially satisfied constraints.

Simplification. Simplification removes *redundant constraints* and simplifies numerical expressions for individual constraints. It helps minimize the overall size of the reliability predicates.

Proposition 2 (Redundant Constraints)**.** *Let a predicate P be a conjunction of reliability constraints. A reliability constraint $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$ in P is redundant if 1) P contains another constraint $r_1 \cdot \mathcal{R}(X_1) \leq r_2' \cdot \mathcal{R}(X_2')$ such that $r_2' \cdot \mathcal{R}(X_2') \leq r_2 \cdot \mathcal{R}(X_2)$ or 2) P contains another constraint $r_1' \cdot \mathcal{R}(X_1') \leq r_2 \cdot \mathcal{R}(X_2)$ such that $r_1 \cdot \mathcal{R}(X_1) \leq r_1' \cdot \mathcal{R}(X_1')$.*

6. Case Studies

We next discuss six computations (three checkable, three approximate) that we implemented in Rely and analyzed using Rely's analysis. For the verification of the benchmarks we use the hardware reliability specification from Figure 3. We describe the summary of the analysis for the implementation and present two additional case studies on how a developer can write the reliability specifications.

6.1 Analysis Summary

Table 9 presents the benchmarks and the analysis results. For each benchmark we present the type of the computation (checkable or approximate), the number of lines of code, the time of the analysis and the number of inequality predicates in the function-level constraint before it is passed to the checker.

We analyze the following six computations:

- **newton:** This computation searches for a root of a univariate function using Newton's Method.

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 9: Benchmark Analysis Summary

- **secant:** This computation also searches for a root of a univariate function, but 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 in the Appendix. For each computation we provide the source code and the corresponding reliability specification – a bound on the reliability of the output given the reliability of the inputs of the computation.

Running on an Intel Xeon E5520 machine with 16 GB of main memory, the analysis times are under one second for all benchmarks. For each computation the constraint checker verifies that the specification of the output reliability of the computation is satisfied given the hardware reliability specification in Figure 3. We note that the use of the simplification procedure that removes redundant predicates during the evaluation significantly reduces the number of final constraints. In particular, it identifies that most of the constraints generated for conditional statements (as in the secant benchmark) and array update statements (as in the matvec benchmark) are redundant, and therefore can be suppressed immediately after analyzing the statement.

6.2 Reliability and Accuracy

The developer writes the reliability specifications of computations. His or her choice of a tolerable reliability bound is typically influenced by the perceived effect that the unreliable execution of the computation may have on the accuracy of the result and the execution time and energy consumption of the computation.

We present two case studies that relate the tolerable reliability of computations to the accuracy of the results that the computations produce.

6.2.1 Checkable Computations

Newton's Method. This computation 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 $f(x_0) = 0$. The reliability of the output of the computation is therefore $c \cdot \mathcal{R}(x_s)$, where c a tolerable reliability degradation that the developer sets.

Figure 10 presents the implementation of this computation. This is an example of a fixed point computation. The computation within each 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 correct 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)`.

```
1  #define tolerance 0.000001
2  #define maxsteps 40
3
4  float<0.9999*R(x)> F(float x in urel);
5
6  float<0.9999*R(x)> dF(float x in urel);
7
8  float <0.99*R(xs)> newton(float xs in urel){
9     float x in urel;
10    float xprim in urel;
11    float t1 in urel;
12    float t2 in urel;
13
14    x = xs;
15    xprim = xs +. 2*.tolerance;
16
17    while ((x -. xprim >=. tolerance)
18           ||. (x -. xprim <=. -tolerance)
19          ) : maxsteps {
20       xprim = x;
21       t1 = F(x);
22       t2 = dF(x);
23       x = x -. t1 /. t2;
24    }
25
26    if ((x -. xprim <=. tolerance)
27         &&. (x -. xprim >=. -tolerance))
28    {
29       return x;
30    } else {
31       return INFTY;
32    }
33 }
```

Figure 10: Newton's Method Implementation

Checker. To check the reliability of the root x_0 that the computation produces, it is enough to evaluate the function $f(x_0)$. The expected result of this evaluation should be equal to zero. In our analysis, the time to execute the checker is equal to a half of the time of a single iteration of Newton's method.

Reexecution. If the evaluation of the checker does not produce the expected value, the computation executes the fully reliable implementation of Newton's Method. The implementation of this *checked computation* is:

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

The function `newton_u` is the potentially unreliable implementation of the computation. If the checker detects an error, the computation calls `newton_r`, the fully reliable implementation of the computation. As we discussed earlier, the reliability of a computation can be ensured using both software (replication) or hardware (changing the operational mode) techniques.

Reliability/Execution Time Tradeoff. Let τ_r be the expected execution time of a Newton's method computation executing on reliable hardware, τ_u the expected execution time of the computation executing on unreliable hardware, τ_{um} the expected execution time of the computation that executes a maximum number of iterations (for this computation 40) on unreliable hardware, and τ_c the expected execution time of the checker.[3] Furthermore, let use denote the projected speedup as s and the target reliability of the function `newton` as r.

The expected execution time of the computation when it produces a correct result is $T_1 = \tau_u + \tau_c$, since it does not execute the `if` branch. The expected execution time of the computation when it executes on unreliable hardware is $T_2 \leq \tau_{um} + \tau_c + \tau_r$, since it executes the `if` branch. This analysis conservatively assumes that the soft errors always cause the computation to execute the maximum number of iterations, hence the upper bound τ_{um} on the execution time. The total expected execution time of the previous code block executed on unreliable hardware is then

$$T' = r \cdot T_1 + (1 - r) \cdot T_2$$

Finally, the expected execution time of the reliable computation τ_r and T' are related via $s = \tau_r / T'$. These analytic expressions allow the developer to estimate the expected resource usage improvement given a particular value of the target reliability r or vice versa.

Numerical Example. As an illustration, we now provide a numerical example of calculating the expected execution time. Let $\tau_r = 1.4\tau_u$, assuming that the computation executes on unreliable hardware and the reliable computation uses the software level replication. We take the average overhead of replication reported in [38]. Furthermore, let the reliable Newton's method computation converge on average in a half of the maximum number of steps (i.e., $\tau_{um} = 2\tau_u$) and the maximum number of iterations is 40.

[3] The analysis can be analogously applied for an alternative resource usage measure such as energy consumption

If the developer's projected speedup is 1.32, he or she can compute the corresponding reliability $r = 0.99$ from the previous formulas. Rely can verify that the computation satisfies this target reliability given the hardware reliability specification from Figure 3.

6.2.2 Approximate Computations

Sum of transformed differences. The motion estimation benchmark presented in Section 2 consists of a fitness function that determines the similarity of each block of pixels in the array `pblocks` to the block `cblock` and a comparison function that finds the element with the minimum fitness function. The fitness function in this computation is the sum of squared pixel distances (Figure 2, Lines 17-32). Another commonly used fitness function is the sum of pixel differences in the frequency domain. Hadamard benchmark is the main building block of this fitness function [54].

The function has the following signature:

```
int<0.99995 * R(bA, bB, satdstart)>
    hadamard (int<R(bA)> bA(2) in urel,
              int<R(bB)> bB(2) in urel,
              int satdstart in urel)
```

This computation takes as input two two-dimensional arrays `bA` and `bB` that represent the pixel blocks and the variable `satdstart`, which is the offset on the return value. The computation is a sequence of arithmetic operations on the array elements. In this case the analysis verifies the overall reliability degradation of `0.99995` relative to the reliability of the input parameters. We present the full source code of the computation in the Appendix.

The errors in this computation may, in general, affect properties of the computation related to 1) integrity and 2) accuracy of the computation. The integrity properties of a computation [10] encompass internal properties of a computation, that ensure that the execution of a computation does not lead to unexpected execution termination or latent memory errors, and external properties, that specify the form and legal results that the computation can produce.

If the computation satisfies the integrity properties, any output returned by the unreliable computation affects only accuracy of the final output of a program that uses this computation.

Integrity. For the sum of absolute transformed differences the integrity property requires that the result of the function is a non-negative number [10]. If the integrity check fails, the Hadamard computation needs to be reexecuted reliably (as for the checkable computations). However, the developer may consider that the probability of this event is negligible and would not impact the expected execution time. In this case, the developer may approximate the execution time of the checked computation on the unreliable hardware as the time without overhead for checking and re-execution.

Accuracy. As with the approximate computation from Section 2, previous research [15, 32] demonstrates that the result of the computation is only slightly affected by modifications of the computation. Loop perforation experiments demonstrate that video encoders produce an acceptable result even if the fitness function skips half of the pixels [32]. In contrast, a developer's specification that the reliability of the computation degrades by a factor c from the reliability of the inputs indicates the developer's expectation that the computation may fail to produce the exact result only in a small fraction of executions.

To determine the tolerable degradation factor c, the developer may perform end-to-end accuracy experiments (such as [15] or [32]), or measure the effect of unreliable execution on a local accuracy metric of the motion estimation. One local accuracy metric is the probability that an index of the best element that the unreliable execution produce is not among the top k elements that the reliable execution would produce. This accuracy metric relies on the fact that even if the computation does not return the best matching block, the next $k - 1$ blocks will produce similar results in the remaining computation.

We finally note that many faults that emerge in the computational pattern that computes a minimum or a maximum element in a collection (motion estimation is an instance of this pattern) may not propagate to the rest of the program. This includes errors that do not affect the ordering of the blocks and errors that may change the ordering between blocks in the `pblocks` array, but do not affect the position of the block with the best score. Note that the Rely's analysis is conservative in the sense that it assumes that any soft error will be visible to the outside computation.

7. Related Work

In this section we provide an overview of the previous research related to Rely.

7.1 Critical and Approximate Data

Static Criticality Analyses. Researchers have previously developed several specification-based static analysis based approaches that let developer identify and separate critical and approximate parts of computations. Sampson et al. [47] present EnerJ, a programming language and an information-flow type system that allows a developer to partition data into approximate and critical data and ensures that operations on approximate data do not affect critical data or memory safety of programs. Carbin et al. [9] present a verification system for nondeterministic relaxed programs based on relational Hoare logic. The system enables rigorous reasoning about relationships between the original and relaxed programs and captures the worst-case difference and non-interference properties of relaxed computations. Flikker [29] is a set of language extensions with 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 reliable or unreliable DRAM memory.

All of this prior research is designed to work with approximate computations that (to be acceptable) must produce correct results most of the time. But it focuses only on the binary distinction between reliable and approximate computations and does not address how often the approximate computations produce the correct result. In contrast, the research presented in this paper provides static probabilistic guarantees about the reliability of computations that execute on unreliable hardware. Because even approximate computations must produce the correct result most of the time, these probabilities are critical to the overall acceptability of these computations.

Criticality Testing. Researchers have also explored techniques that identify critical and/or approximate parts of the computation in existing imperative programs. All these approaches are dynamic in nature and use representative inputs to guide the search for critical subcomputations. These techniques are based on program transformations [32, 43], directed input fuzzing [3, 11, 52], and fault injection [28, 46, 53]. Snap [11] fuzzes values of the input fields of multimedia formats to determine critical subcomputations and critical input fields (the fields used by critical computations). Quality of service profiler [32] transforms programs using loop perforation to identify approximate parts of a computation.

Accuracy Analysis. As a related problem, researchers have analyzed how the uncertainty in the computation introduced by randomized program transformations or soft errors affects the accuracy of the final result of the computation. The existing techniques use static probabilistic or worst-case reasoning [5, 9, 13, 31, 41] or empirical evaluation [1, 2, 15, 23, 28, 29, 32, 43, 44, 50, 53].

7.2 Probabilistic Program Analysis

Kozen's work [25] 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 [4, 5, 34] and abstract interpretation [14, 16, 33, 51].

The language features that introduce probabilistic nondeterminism in programs studied in this previous research include probabilistic sampling, `x = random()` [4, 5, 25, 33], probabilistic choice between statements, `s1` $\oplus_p$ `s2` [14, 16, 34], and specifications of distributions of inputs of the computation [51]. Rely refines the probabilistic operators by defining a set of unreliable executions and memory accesses (each of which is a specific probabilistic assignment) that model faults in the underlying hardware model.

Morgan et al. [34] propose a weakest-precondition style analysis for probabilistic programs that treats the programs as *expectation transformers*. Preconditions and postconditions are defined as the bounds on probabilities that particular logical predicates hold at a specified location in the program. Rely' analysis, like [34], constructs weakest-precondition predicates for program statements. In contrast, Rely's predicates are relational, over the states of the reliable and unreliable executions of the program.

Barthe et al. [4, 5] define a probabilistic relational Hoare logic for a simple probabilistic imperative language using a denotational, monadic-style semantics. 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. These techniques require manual proofs or an SMT solver to verify the validity of predicates. In comparison, Rely presents an operational semantics for an imperative language and a semantics of reliability predicates based on the notion of a paired program execution as a foundation for the quantitative reliability analysis. Rely defines special joint reliability factors to simplify the construction and checking of the reliability predicates.

7.3 Fault Tolerance and Resilience

Researchers have developed various software-based, hardware based, or mixed-mode approaches for detection and recovery from soft errors.

Reis et al. [42] present a compiler based approach that replicates parts of the computation to detect and recover from single event upset errors. Perry et al. [38] present a fault tolerance-aware assembly language and type system that reasons about placement of replicated instructions in a way that they do not interfere with the original data or control-flow. They later extended this research to handle control flow errors [39]. These approaches typically apply replication transformations to the entire computation. Our technique allows for reasoning about selective placement of software fault tolerant mechanisms for data-based soft errors via reliability specifications and critical execution blocks.

De Kruijf et al. [15] present a language, compiler and a hardware architecture that can detect certain classes of soft errors and execute developer specified recovery routines if errors happen within a computation. Feng et al. [21] present a compiler that extends with fault tolerant code only some of the subcomputations, selected automatically by the compiler. Samurai [36] uses replication to protect critical data (selected by a developer) that resides in unreliable memory. Yarra [48] provides a set of annotations for specifying critical data and combines static analysis and dynamic checks to protect critical data from arbitrary (potentially malicious) reads or writes. Researchers have also developed techniques for determining a set of locations in a program where detectors for numerical and memory errors should be placed [22, 53]. While these techniques provide a means to specify critical code or memory blocks, they do not specify or reason about the quantitative reliability of the values that the computation operates on.

Researchers have also proposed frameworks and systems that recover a program's execution that experienced fault without the help of prior developer's annotations [24, 37, 45]. These recovery mechanisms demonstrate the potential to help the application produce the acceptable output despite the experienced fault.

7.4 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 [12], stochastic processors [35], error resilient architecture [26], and the Truffle architecture [19]. These techniques use voltage scaling, at different granularity, to save energy of the system at the expense of the accuracy of the results. The researchers demonstrate that for specific classes of applications, such as multimedia processing and machine learning, the obtained tradeoffs may be profitable to the user. Our approach aims to help developers better understand and control the behavior of their applications on such architectures.

8. 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 Hank Hoffmann, Deokhwan Kim, Vladimir Kiriansky, Stelios Sidiroglou, Rishabh Singh, and the anonymous referees for the useful comments on the previous versions of this work.

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, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and AmarasingheS. PetaBricks: A language and compiler for algorithmic choice. PLDI, 2009.

[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. In *OOPSLA*, 2012.

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

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

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

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

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

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

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

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

[12] L. Chakrapani, B. Akgul, S. Cheemalavagu, P. Korkmaz, K. Palem, and B. Seshasayee. Ultra-efficient (embedded) soc architectures based on probabilistic cmos (pcmos) technology. DATE, 2006.

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

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

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

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

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

[18] 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.

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

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

[21] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. ASPLOS'10.

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

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

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

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

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

[27] 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 Transactions on Software Engineering*, 1990.

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

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

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

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

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

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

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

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

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

[37] 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.

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

[39] F. Perry and D. Walker. Reasoning about control flow in the presence of transient faults. SAS, 2008.

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

[41] Jason R. and B. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. ICFP, 2010.

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

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

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

[45] 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.

[46] M. Rinard, C. Cadar, and H. Nguyen. Exploring the acceptability envelope. OOPSLA, 2005.

[47] 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.

[48] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. Zorn. Yarra: An extension to c for data integrity and partial safety. CSF, 2011.

[49] 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.

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

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

[52] W. N. Sumner, T. Bao, X. Zhang, and S. Prabhakar. Coalescing executions for fast uncertainty analysis. In *ICSE*, 2011.

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

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

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