15

Programming Unreliable Hardware

Michael Carbin
Massachusetts Institute of Technology
Sasa Misailovic
University of Illinois

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

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

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

15.1 Introduction

Reliability is a major concern in the design of computer systems. The current goal of delivering systems with negligible error rates restricts the available design space and imposes significant engineering costs. And as other goals such as energy efficiency, circuit scaling, and new features and functionality continue to grow in importance, maintaining even current error rates will become increasingly difficult.

[a] From *Foundations of Probabilistic Programming*, edited by Gilles Barthe, Joost-Pieter Katoen and Alexandra Silva published 2020 by Cambridge University Press.

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

In response to this situation, researchers have developed numerous techniques for detecting and masking errors in both hardware (Ernst et al., 2003) and software (Reis et al., 2005; Perry et al., 2007; de Kruijf et al., 2010). Because these techniques typically come at the price of increased execution time, increased energy consumption, or both, they can substantially hinder or even cripple overall system performance.

Many computations, however, can easily tolerate occasional errors. An *approximate computation* (including many multimedia, financial, machine learning, and big data analytics applications) can tolerate occasional errors in its execution and/or the data that it manipulates (Rinard, 2006; Misailovic et al., 2010; Carbin and Rinard, 2010). A *checkable computation* can be augmented with an efficient checker that verifies either the exact correctness (Blum and Kanna, 1989; Leveson et al., 1990) or the approximate acceptability of the results that the computation produces. If the checker does detect an error, it can reexecute the computation to obtain an acceptable result.

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

15.1.1 Background

Approximate computations have emerged as a major component of many computing environments. Motivated in part by the observation that approximate computations can often acceptably tolerate occasional computation and/or data errors (Rinard, 2006; Misailovic et al., 2010; Carbin and Rinard, 2010), researchers have developed a range of new mechanisms that forgo exact correctness to optimize other objectives. Typical goals include maximizing program performance subject to an accuracy constraint and altering program execution to recover from otherwise fatal errors (Rinard et al., 2004).

Software Techniques Most software techniques deploy *unsound transformations* – transformations that change the semantics of an original exact program. Proposed mechanisms include skipping tasks (Rinard, 2006), loop perforation (skipping iterations of time-consuming loops) (Misailovic et al., 2010; Sidiroglou et al., 2011), sampling reduction inputs (Zhu et al., 2012), multiple selectable implementations of a given component or components (Baek and Chilimbi, 2010; Ansel et al., 2011; Hoffman et al., 2011; Zhu et al., 2012), dynamic knobs (configuration parameters that can be changed as the program executes) (Hoffman et al., 2011) and synchronization elimination (forgoing synchronization not required to produce an acceptably accurate result) (Misailovic et al., 2013). The results show that aggressive techniques such

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

as loop perforation can deliver up to a four-fold performance improvement with acceptable changes in the quality of the results that the application delivers.

Hardware Techniques The computer architecture community has begun to investigate new designs that improve performance by breaking the traditional fully reliable digital abstraction that computer hardware has traditionally sought to provide. The goal is to reduce the cost of implementing a reliable abstraction on top of physical materials and manufacturing methods that are inherently unreliable. For example, researchers are investigating designs that incorporate aggressive device and voltage scaling techniques to provide low-power ALUs and memories. A key aspect of these components is that they forgo traditional correctness checks and instead expose timing errors and bitflips with some non-negligible probability (de Kruijf et al., 2010; Esmaeilzadeh et al., 2012; Leem et al., 2010; Liu et al., 2011; Narayanan et al., 2010; Palem, 2005; Sampson et al., 2011).

In this work, we focus on hardware techniques that manifest as *soft errors* – errors that occur in the system nondeterministically. They may affect the values computed by individual instruction executions or data stored in individual memory locations. We can associate the probability of soft-error occurrence with each (unprotected) instruction. However, soft errors do not last over multiple instruction executions or permanently damage the hardware.

15.1.2 Reasoning About Approximate Programs

Approximate computing violates the traditional contract that the programming system must preserve the standard semantics of the program. It therefore invalidates standard paradigms and motivates new, more general, approaches to reasoning about program correctness, and acceptability.

One key aspect of approximate applications is that they typically contain *critical* regions (which must execute without error) and *approximate* regions (which can execute acceptably even in the presence of occasional errors) (Rinard, 2006; Carbin and Rinard, 2010). Existing systems, tools, and type systems have focused on helping developers identify, separate, and reason about the binary distinction between critical and approximate regions (Rinard, 2006; Carbin and Rinard, 2010; Liu et al., 2011; Sampson et al., 2011; Esmaeilzadeh et al., 2012). However, in practice, no computation can tolerate an unbounded accumulation of errors – to execute acceptably, executions of even approximate regions must satisfy some minimal requirements.

Approximate computing therefore raises a number of fundamental new research questions. For example, what is the probability that an approximate program will produce the same result as a corresponding original exact program? How much do

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

the results differ from those produced by the original program? And is the resulting program safe and secure?

Because traditional correctness properties do not provide an appropriate conceptual framework for addressing these kinds of questions, we instead work with *acceptability properties* – the minimal requirements that a program must satisfy for acceptable use in its designated context. We identify three kinds of acceptability properties and use the following program (which computes the minimum element `min` in an `N`-element array) to illustrate these properties:

```
int min = INT_MAX;
for (int i = 0; i < N; ++i)
    if (a[i] < min) min = a[i];
```

Integrity Properties: Integrity properties are properties that the computation must satisfy to produce a successful result. Examples include both computation-independent properties (no out of bounds accesses, null dereferences, divide by zero errors, or other actions that would crash the computation) and computation-dependent properties (for example, the computation must return a result within a given range). One integrity property for our example is that accesses to the array `a` must always be within bounds.

Reliability Properties: Reliability properties characterize the probability that the produced result is correct. Reliability properties are often appropriate for approximate computations executing on unreliable hardware platforms that exhibit occasional nondeterministic errors. A potential reliability property for our example program is that `min` must be the minimum element in `a[0]–a[N-1]` with probability at least 95%.

Accuracy Properties: Accuracy properties characterize how accurate the produced result must be. For example, an accuracy property might state that the transformed program must produce a result that differs by at most a specified percentage from the result that a corresponding original program produces (Misailovic et al., 2011; Zhu et al., 2012). Alternatively, a potential accuracy property for our example program might require the `min` to be within the smallest `N/2` elements `a[0]–a[N-1]`. Such an accuracy property might be satisfied by, for example, a loop perforation transformation that skips `N/2-1` of the loop iterations.

In other research, we have developed techniques for reasoning about integrity properties (Carbin et al., 2012, 2013a) and both worst-case and probabilistic accuracy properties (Misailovic et al., 2011; Zhu et al., 2012; Carbin et al., 2012). We have extended the research presented in this chapter to include combinations of reliability and accuracy properties (Misailovic et al., 2014), and more recently to reason about message-passing parallel programs with approximate communication (Fernando et al., 2019).

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

15.1.3 Verifying Reliability (Contributions)

To meet the challenge of reasoning about reliability, we present a 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 parts of the computation execute on unreliable hardware. 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.

Rely supports and is specifically designed to enable partitioning a program into *critical* regions (which must execute without error) and *approximate* regions (which can execute acceptably even in the presence of occasional errors) (Rinard, 2006; Carbin and Rinard, 2010). In contrast to previous approaches, which support only a binary distinction between critical and approximate regions, quantitative reliability can provide precise static probabilistic acceptability guarantees for computations that execute on unreliable hardware platforms. This chapter describes the following contributions we initially presented in Carbin et al. (2013b).

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

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

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, the analysis computes a symbolic reliability precondition that characterizes the set of valid specifications for the function. The analysis then verifies that the developer-provided specifications are valid according to the reliability precondition.

15.2 Example

Figure 15.1 presents the syntax of the Rely language. Rely is an imperative language for computations over integers, floats (not presented), and multidimensional arrays.

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

$$
\begin{array}{rcl@{\qquad}rcl}
n & \in & \text{Int}_{\text{M}} & e \in \mathit{Exp} & \rightarrow & n \mid x \mid \texttt{(}\mathit{Exp}\texttt{)} \mid \mathit{Exp}\ \mathit{iop}\ \mathit{Exp} \\
r & \in & \mathbb{Q} & b \in \mathit{BExp} & \rightarrow & \texttt{true} \mid \texttt{false} \mid \mathit{Exp}\ \mathit{cmp}\ \mathit{Exp} \mid \texttt{(}\mathit{BExp}\texttt{)} \mid \\
x, \ell & \in & \text{Var} & & & \mathit{BExp}\ \mathit{lop}\ \mathit{BExp} \mid \texttt{!}\mathit{BExp} \mid \texttt{!.}\mathit{BExp} \\
a & \in & \text{ArrVar} & \mathit{CExp} & \rightarrow & e \mid a
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
m & \in & \text{MVar} & F & \rightarrow & (T \mid \texttt{void})\ \mathit{ID}\ \texttt{(}P^*\texttt{)}\ \texttt{\{}\ S\ \texttt{\}} \\
V & \rightarrow & x \mid a \mid V, x \mid V, a & P & \rightarrow & P_0\ \texttt{[in}\ m\texttt{]} \\
\mathit{RSpec} & \rightarrow & r \mid \texttt{R(}V\texttt{)} \mid r \star \texttt{R(}V\texttt{)} & P_0 & \rightarrow & \texttt{int}\ x \mid T\ a\texttt{(}n\texttt{)} \\
T & \rightarrow & \texttt{int} \mid \texttt{int<}\mathit{RSpec}\texttt{>} & S & \rightarrow & D^*\ S_s\ S_r^?
\end{array}
$$

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

Figure 15.1 Rely's Language Syntax

To illustrate how a developer can use Rely, Figure 15.2 presents a Rely-based implementation of a pixel block search algorithm derived from that in the x264 video encoder (x264, 2013).

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 `cblock`, potentially decreasing the video's peak signal-to-noise ratio or increasing its size. However, previous studies on soft error injection (de Kruijf et al., 2010) and more aggressive transformations like loop perforation (Misailovic et al., 2010; Sidiroglou et al., 2011) have demonstrated that the quality of x264's final result is only slightly affected by perturbations of this computation.

15.2.1 Reliability Specifications

The function declaration on Line 6 specifies the types and reliabilities of `search_ref`'s parameters and return value. The parameters of the function are `pblocks(3)`, a three-dimensional array of pixels, and `cblock(2)`, a two-dimensional array of pixels. In addition to the standard signature, the function declaration contains *reliability specifications* for each result that the function produces.

Rely's reliability specifications express the reliability of a function's results – when executed on an unreliable hardware platform – as a function of the reliabil-

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

```
1  #define nblocks 20
2  #define height  16
3  #define width   16
4
5  int<0.99*R(pblocks, cblock)>
6  search_ref (
7   int<R(pblocks)> pblocks(3) in urel,
8   int<R(cblock)> cblock(2) in urel)
9  {
10   int minssd = INT_MAX,
11       minblock = -1 in urel;
12   int ssd, t, t1, t2 in urel;
13   int i = 0, j, k;
14
15   repeat nblocks  {
16     ssd = 0;
17     j = 0;
18     repeat height {
19       k = 0;
20       repeat width {
21         t1 = pblocks[i,j,k];
22         t2 = cblock[j,k];
23         t = t1 -. t2;
24         ssd = ssd +. t *. t;
25         k = k + 1;
26       }
27       j = j + 1;
28     }
29
30     if (ssd <. minssd) {
31       minssd = ssd;
32       minblock = i;
33     }
34
35     i = i + 1;
36   }
37   return minblock;
38 }
```

Figure 15.2 Rely Code for Motion Estimation

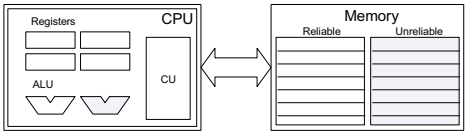

Figure 15.3 Machine Model.

```
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 15.4 Hardware Reliability Specification

```
(3)   {Q0 ∧ Aret ≤ r0^4 · R(i, ssd, minssd)
       ∧ Aret ≤ r0^4 · R( minblock, ssd, minssd)}
    if (ssd <. minssd) {
(2)   {Q0 ∧ Aret ≤ r0 · R(i, ℓ30)}
      minssd = ssd;
      {Q0 ∧ Aret ≤ r0 · R(i, ℓ30)}
      minblock = i;
      {Q0 ∧ Aret ≤ r0 · R(minblock, ℓ30)}
    } else {
(2)   {Q0 ∧ Aret ≤ r0 · R(minblock, ℓ30)}
      skip;
      {Q0 ∧ Aret ≤ r0 · R(minblock, ℓ30)}
    }
(1)   {Q0 ∧ Aret ≤ r0 · R(minblock, ℓ30)}
```

Figure 15.5 `if` Statement Analysis in Last Iteration

ities of its inputs. The specification for the reliability of `search_ref`'s result is `int<0.99*R(pblocks,cblock)>`. This states that the return value is an integer with a reliability that is at least 99% of the **joint reliability** of the parameters `pblocks` and `cblock` (denoted by `R(pblocks, cblock)`). The joint reliability of a set of parameters is the probability that they all have the correct value when passed in from the caller. The joint reliability is a key abstraction for our verification; we will formalize it in Section 15.4.2.

The reliability specification holds for all possible values of the joint reliability of `pblocks` and `cblock`. For instance, if the contents of the arrays `pblocks` and `cblock` are fully reliable (correct with probability one), then the return value is correct with probability 0.99.

In Rely, arrays are passed by reference and the execution of a function can, as

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

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.

15.2.2 Unreliable Computation

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

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

Lines 10-13 declare the local variables of the function. By default, variables in Rely are allocated in a default, fully reliable memory region. However, a developer can also optionally specify a memory region for each local variable. For example, the variables declared on Lines 10-12 reside in `urel`.

Unreliable Operations The operations on Lines 23, 24, and 30 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.

Using these operations, `search_ref`'s implementation *approximately* computes the index (`minblock`) of the most similar block, i.e. the block with the minimum distance from `cblock`. The `repeat` statement on line 15, iterates a constant `nblock` number of times, enumerating over all previously encoded blocks. For each encoded block, the `repeat` statements on lines 18 and 20 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 30 through 33 selects the block that is – approximately – the most similar to `cblock`.

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

15.2.3 Hardware Semantics

Figure 15.3 illustrates the conceptual machine model behind Rely's reliable and unreliable operations; the model consists of a CPU and a memory.

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

The arithmetic-logical unit can execute reliably or unreliably. We have represented this in Figure 15.3 by physically separate reliable and unreliable functional units, but this distinction can be achieved through other mechanisms, such as dual-voltage architectures (Esmaeilzadeh et al., 2012). Unreliable functional units may omit additional checking logic, enabling the unit to execute more efficiently but also allowing for soft errors that may occur due to, for example, power variations within the ALU's combinatorial circuits or particle strikes. As is provided by existing computer architecture proposals (Sampson et al., 2011; Esmaeilzadeh et al., 2012), 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 address.

Memory Rely supports machines with memories that consist of an arbitrary number of memory partitions (each potentially of different reliability), but for simplicity Figure 15.3 partitions memory into two regions: reliable and unreliable. Unreliable memories can, for example, use decreased DRAM refresh rates to reduce power consumption at the expense of increased soft error rates (Liu et al., 2011; Sampson et al., 2011).

Hardware Reliability Specification

We abstract the behavior of the unreliable hardware platforms through a reliability specification. It specifies the reliability of arithmetic/logical and memory operations. Figure 15.4 presents a hardware reliability specification that is inspired by the results from existing computer architecture literature (Ernst et al., 2003; Liu et al., 2011). Each entry specifies the reliability – the probability of a correct execution – of arithmetic operations (e.g., `+.`) and memory read/write operations.

For ALU operations, the presented reliability specification uses the reliability of an unreliable multiplication operation that we selected from Ernst et al. (2003, Figure 9). For memory operations, the specification uses the probability of a bit flip in a memory cell that we selected from Liu et al. (2011, Figure 4) with extrapolation to the probability of a bit flip within a 32-bit word. Note that a memory region specification includes two reliabilities: the reliability of a read (`rd`) and the reliability of a write (`wr`).

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

15.2.4 Reliability Analysis

Given a Rely program, Rely's reliability analysis verifies that each function in the program satisfies its reliability specification when executed on unreliable hardware. The analysis takes as input a Rely program and a *hardware reliability specification*.

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

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

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

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

Precondition Generator

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

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

reliability of the set X of variables (including the function parameters) on entry to the function.

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

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

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

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

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

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

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

Loops The statement that precedes the `return` statement is the `repeat` statement on Line 15. A key difficulty with reasoning about the reliability of variables modified within a loop is that if a variable is updated unreliably and has a loop-carried dependence then its reliability monotonically decreases as a function of the number of loop iterations. Because the reliability of such variables can, in principle, decrease arbitrarily in an unbounded loop, Rely provides both an unbounded loop statement (with an associated analysis) and an alternative *bounded loop* statement that lets a developer specify a compile-time bound on the maximum number of its iterations that therefore bounds the reliability degradation of modified variables. The loop on Line 15 iterates `nblocks` times and therefore decreases the reliability of any modified variables `nblocks` times. Because the reliability degradation is bounded, Rely's analysis uses unrolling to reason about the effects of a bounded loop.

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

Conditionals The analysis of the body of the loop on Line 15 encounters the `if` statement on Line 30.[1] This `if` statement uses an unreliable comparison operation on `ssd` and `minssd`, both of which reside in unreliable memory. The reliability of `minblock` when modified on Line 32 therefore also depends on the reliability of this expression because faults may force the execution down a different path.

Figure 15.5 presents a Hoare logic style presentation of the analysis of the conditional statement. The analysis works in three steps; the preconditions generated by each step are numbered with the corresponding step.

Step 1 To capture the implicit dependence of a variable on an unreliable condition, Rely's analysis first uses latent *control flow variables* to make these dependencies explicit. A control flow variable is a unique program variable (one for each statement) that records whether the conditional evaluated to *true* or *false*. We denote the control flow variable for the `if` statement on Line 30 by ℓ_{30}.

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

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

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

The reliability of the `if` statement's expression is greater than or equal to the product of (1) the reliability of the `<.` operator (r_0), (2) the reliability of reading both `ssd` and `minssd` from unreliable memory (r_0^2), and (3) the reliability of the computation that produced `ssd` and `minssd` ($\mathcal{R}(\texttt{ssd},\texttt{minssd})$). The analysis therefore transforms each predicate that contains the variable ℓ_{30}, by multiplying the right-hand side of the inequality with r_0^3 and replacing the variable ℓ_{30} with `ssd` and `minssd`.

This produces the precondition Q_2:

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

[1] This happens after encountering the increment of `i` on Line 35, which does not modify the current precondition because it does not reference `i`.

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

Simplification After unrolling a single iteration of the loop that begins at Line 15, the analysis produces

$$Q_0 \wedge A_{ret} \leq r_0^{2564} \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock}, \texttt{i}, \texttt{ssd}, \texttt{minssd})$$

as the precondition for a single iteration of the loop's body. The constant 2564 represents the number of unreliable operations within a single loop iteration.

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

This simplification is how Rely's analysis achieves scalability when there are multiple paths in the program. Specifically, the simplified precondition is a lower bound of the reliability specification of all its program paths.

Final Precondition When the analysis reaches the beginning of the function after fully unrolling the loop on Line 15, it has a precondition that bounds the set of valid specifications as a function of the reliability of the parameters of the function. For `search_ref`, the analysis generates the precondition $A_{ret} \leq 0.994885 \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock}) \wedge A_{\texttt{pblocks}} \leq \mathcal{R}(\texttt{pblocks}) \wedge A_{\texttt{cblock}} \leq \mathcal{R}(\texttt{cblock})$.

Precondition Checker

The final precondition is a conjunction of predicates of the form $A_{out} \leq r \cdot \mathcal{R}(X)$, where A_{out} is a placeholder for the reliability specification of an output. Because reliability specifications are all of the form $r \cdot \mathcal{R}(X)$ (Figure 15.1), each predicate in the final precondition (where each A_{out} is replaced with its specification) is of the form form $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$, where $r_1 \cdot \mathcal{R}(X_1)$ is a reliability specification and $r_2 \cdot \mathcal{R}(X_2)$ is computed by the analysis. Similar to the analysis's simplifier (Section 15.2.4), the precondition checker verifies the validity of each predicate by checking that (1) r_1 is less than r_2 and (2) $X_2 \subseteq X_1$.

For `search_ref`, the analysis computes the predicates

$$0.99 \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock}) \leq 0.994885 \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock}),$$
$$\mathcal{R}(\texttt{pblocks}) \leq \mathcal{R}(\texttt{ pblocks}), \text{ and also } \mathcal{R}(\texttt{cblock}) \leq \mathcal{R}(\texttt{cblock}).$$

Because these predicates are valid according to the checking procedure, `search_ref` satisfies its reliability specification when executed.

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

15.3 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 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 15.3.4, we provide big-step definitions that specify the probability of an entire execution.

15.3.1 Preliminaries

Rely's semantics is given in the terms of an abstract machine that consists of a heap and a stack. The heap is an abstraction over the physical memory of the concrete machine, including its various reliable and unreliable memory regions. Each variable (both scalar and array) is allocated in the heap. The stack consists of frames – one for each function invocation – which contain references to the locations of each allocated variable. This conceptual model of local variables does not need to be concretized in the compilation model. For example, placing local variables in a reliable stack can achieve competitive performance (Misailovic et al., 2014).

Hardware Reliability Specification A hardware reliability specification $\psi \in \Psi = (iop + cmp + lop + M_{op}) \rightarrow \mathbb{Q}_{\geq 0}$ is a finite map from arithmetic/logical operations (iop, cmp, lop) and *memory region operations* (M_{op}) to reliabilities (i.e., the probability that the operation executes correctly).

Arithmetic/logical operations *iop*, *cmp*, and *lop* include both reliable and unreliable versions of each integer, comparison, and logical operation. The reliability of each reliable operation is 1 and the reliability of an unreliable operation is as provided by a specification (Section 15.2.3).

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

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

References Given a finite, contiguous address space Loc, a *reference* is a tuple $\langle n_b, \langle n_1, \ldots, n_k \rangle, m \rangle \in \text{Ref}$ consisting of a base address $n_b \in \text{Loc}$, a dimension descriptor $\langle n_1, \ldots, n_k \rangle$, and a memory region m. Base addresses and the components of dimension descriptors range over the finite set of bounded-width machine integers $n \in \text{Int}_\text{M}$.

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

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

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

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

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

The discrete probability distribution $P_f(n_f \mid op, n_1, ..., n_k)$ models the manifestation of a soft error during an incorrect execution of an operation. Specifically, it gives the probability that an incorrect execution of an operation op on operands $n_1, \ldots, n_k$ produces a value n_f that is different from the correct result of the operation. This distribution is inherently tied to the properties of the underlying hardware.

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

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

15.3.2 Semantics of Expressions

Figure 15.6 presents a selection of the rules for the dynamic semantics of integer expressions. The labeled probabilistic small-step evaluation relation $\langle e, \sigma, h \rangle \xrightarrow[\psi]{\theta, p} e'$ states that from a frame σ and a heap h, an expression e evaluates in one step with

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

$$\text{E-Var-C}\quad \frac{\langle n_b, \langle 1\rangle, m\rangle = \sigma(x)}{\langle x, \sigma, h\rangle \overset{\mathrm{C},\, \psi(rd(m))}{\longrightarrow_\psi} h(n_b)}$$

$$\text{E-Var-F}\quad \frac{\langle n_b, \langle 1\rangle, m\rangle = \sigma(x) \qquad p = (1 - \psi(rd(m))) \cdot P_f(n_f \mid rd(m), h(n_b))}{\langle x, \sigma, h\rangle \overset{\langle \mathrm{F}, n_f\rangle,\, p}{\longrightarrow_\psi} n_f}$$

$$\text{E-Iop-R1}\quad \frac{\langle e_1, \sigma, h\rangle \overset{\theta,\, p}{\longrightarrow_\psi} e_1'}{\langle e_1 \ iop \ e_2, \sigma, h\rangle \overset{\theta,\, p}{\longrightarrow_\psi} e_1' \ iop \ e_2}$$

$$\text{E-Iop-R2}\quad \frac{\langle e_2, \sigma, h\rangle \overset{\theta,\, p}{\longrightarrow_\psi} e_2'}{\langle n \ iop \ e_2, \sigma, h\rangle \overset{\theta,\, p}{\longrightarrow_\psi} n \ iop \ e_2'}$$

$$\text{E-Iop-C}\quad \frac{}{\langle n_1 \ iop \ n_2, \sigma, h\rangle \overset{\mathrm{C},\, \psi(iop)}{\longrightarrow_\psi} iop(n_1, n_2)}$$

$$\text{E-Iop-F}\quad \frac{p = (1 - \psi(iop)) \cdot P_f(n_f \mid iop, n_1, n_2)}{\langle n_1 \ iop \ n_2, \sigma, h\rangle \overset{\langle \mathrm{F}, n_f\rangle,\, p}{\longrightarrow_\psi} n_f}$$

Figure 15.6 Dynamic Semantics of Integer Expressions

probability p to an expression e' given a hardware reliability specification ψ. The label $\theta \in \{\mathrm{C}, \langle \mathrm{C}, n\rangle, \langle \mathrm{F}, n_f\rangle\}$ denotes whether the transition corresponds to a correct (C or $\langle \mathrm{C}, n\rangle$) or a faulty ($\langle \mathrm{F}, n_f\rangle$) evaluation of that step. For a correct transition $\langle \mathrm{C}, n\rangle$, $n \in \mathrm{Int_M}$ records a nondeterministic choice made for that step. For a faulty transition $\langle \mathrm{F}, n_f\rangle$, $n_f \in \mathrm{Int_M}$ represents the value that the fault introduced in the semantics of the operation.

To illustrate the meaning of the rules, consider the rules for variable reference expressions. A variable reference x reads the value stored in the memory address for x. A variable reference can be evaluated in two ways:

- Correct [E-Var-C]. The variable reference evaluates correctly and successfully returns the integer stored in x. This happens with probability $\psi(rd(m))$, where m is the memory region in which x allocated. This probability is the reliability of reading from x's memory region.
- Faulty [E-Var-F]. The variable reference experiences a fault and returns another integer n_f. The probability that the faulty execution returns a specific integer n_f is $(1 - \psi(rd(m))) \cdot P_f(n_f \mid rd(m), h(n_b))$. P_f is the distribution that gives the probability that a failed memory read operation returns a value n_f instead of the true stored value $h(n_b)$ (Section 15.3.1).

15.3.3 Semantics of Statements

Figure 15.7 presents the semantics of the scalar and control flow fragment of Rely. The labeled probabilistic small-step execution relation $\langle s, \varepsilon\rangle \overset{\theta,\, p}{\longrightarrow_\psi} \langle s', \varepsilon'\rangle$ 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

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

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

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

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

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

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

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

E-If-False

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

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-Bounded

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

Figure 15.7 Dynamic Semantics of Statements

whether the transition evaluated correctly (C or $\langle \mathrm{C}, n\rangle$) or experienced a fault ($\langle \mathrm{F}, n_f\rangle$). The semantics of the statements in the language is largely similar to that of traditional presentations except that the statements have the ability to encounter faults during execution.

The semantics we present here is designed to allow unreliable computation at all points in the application – subject to the constraint that the application is still memory safe and exhibits control flow integrity.

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

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 due to an unreliable array index computation, Rely requires that each array read and write include a bounds check. These bounds check computations execute reliably. We presented the semantics of memory accesses in Carbin et al. (n.d.).

Control Flow Integrity To prevent execution from taking control flow edges that do not exist in the program's static control flow graph, Rely assumes that (1) instructions are stored, fetched, and decoded reliably (as supported by existing unreliable processor architectures (Sampson et al., 2011; Esmaeilzadeh et al., 2012)) and (2) targets of control flow branches are reliably computed. These two properties allow for the control flow transfers in the rules [E-If-True], [E-If-False], and [E-Seq-R2] to execute reliably with probability 1.

Note that the semantics does not require a specific underlying mechanism to achieve reliable execution and, therefore, an implementation can use any applicable software or hardware technique (Reis et al., 2005; Perry et al., 2007; de Kruijf et al., 2010; Feng et al., 2010; Pattabiraman et al., 2008; Schlesinger et al., 2011; Hiller et al., 2002; Thomas and Pattabiraman, 2013).

15.3.4 Big-step Notations

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

Definition 15.1 (Big-step Trace Semantics)**.**

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

The big-step trace semantics is, conceptually, a reflexive transitive closure of the small-step execution relation that records a trace of the execution. We define a *trace*, $\tau \in \mathrm{T} ::= \cdot \mid \theta :: \mathrm{T}$, as the sequence of small-step transition labels, $\tau = \theta_1 :: \ldots :: \theta_n :: \cdot$. The *probability of a trace*, p, is the product of the probabilities of each transition, $p = \prod_{i=1}^{n} p_i$.

Definition 15.2 (Big-step Aggregate Semantics)**.**

$$\langle s, \varepsilon\rangle \overset{p}{\Longrightarrow}_{\psi} \varepsilon' \text{ where } p = \sum \{\!\{ p_\tau \mid \exists \tau \in \mathrm{T}. \langle s, \varepsilon\rangle \overset{\tau, p_\tau}{\Longrightarrow}_{\psi} \varepsilon' \}\!\}$$

The big-step aggregate semantics computes the aggregate probability (over all finite length traces) that a statement s evaluates to an environment ε' from an

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

environment ε given a hardware reliability specification ψ. The big-step aggregate semantics therefore gives the total probability that a statement s starts from an environment ε and terminates in an environment ε'.[2]

Termination and Errors An unreliable execution of a statement may experience a run-time error (due to an out-of-bounds array access) or not terminate at all. The big-step aggregate semantics does not collect such executions. Therefore, the sum of the probabilities of the big-step transitions from an environment ε may not equal to 1. Specifically, let $p \in \mathrm{E} \to \mathbb{R}_{\geq 0}$ be a measure for the set of environments reachable from ε, i.e., $\forall \varepsilon'. \langle s, \varepsilon \rangle \overset{p(\varepsilon')}{\Longrightarrow}_{\psi} \varepsilon'$. Then p is *subprobability* measure, i.e., $0 \leq \sum_{\varepsilon' \in \mathrm{E}} p(\varepsilon') \leq 1$ (Kozen, 1981).

15.4 Semantics of Quantitative Reliability

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

15.4.1 Paired Execution

The *paired execution* semantics is the primary execution relation that enables one 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 15.3 (Paired Execution)**.** $\varphi \in \Phi = \mathrm{E} \to \mathbb{R}_{\geq 0}$

$$\langle s, \langle \varepsilon, \varphi \rangle \rangle \Downarrow_{\psi} \langle \varepsilon', \varphi' \rangle \text{ such that } \langle s, \varepsilon \rangle \overset{1}{\Longrightarrow}_{1_{\psi}} \varepsilon' \text{ and}$$
$$\varphi'(\varepsilon'_u) = \sum \{\!\{ \varphi(\varepsilon_u) \cdot p_u \mid \varepsilon_u \in \mathrm{E}, \langle s, \varepsilon_u \rangle \overset{p_u}{\Longrightarrow}_{\psi} \varepsilon'_u \}\!\}$$

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. In particular, φ is a distribution on environments before the unreliable execution of s whereas φ' is the distribution on environments

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

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

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

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

$$[\![R]\!] \in \mathrm{E} \times \Phi \to \mathbb{R}_{\geq 0} \qquad [\![r]\!](\varepsilon, \varphi) = r \qquad [\![R_1 \cdot R_2]\!](\varepsilon, \varphi) = [\![R_1]\!](\varepsilon, \varphi) \cdot [\![R_2]\!](\varepsilon, \varphi)$$

$$[\![\mathcal{R}(X)]\!](\varepsilon, \varphi) = \sum_{\varepsilon_u \in \mathcal{E}(X, \varepsilon)} \varphi(\varepsilon_u) \qquad \mathcal{E} \in \mathcal{P}(\mathrm{Var} + \mathrm{ArrVar}) \times \mathrm{E} \to \mathcal{P}(\mathrm{E})$$

$$\mathcal{E}(X, \varepsilon) = \{\varepsilon' \mid \varepsilon' \in \mathrm{E} \wedge \forall v.\, v \in X \Rightarrow \mathrm{equiv}(\varepsilon', \varepsilon, v)\})$$

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

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

Figure 15.8 Predicate Semantics

after executing s. These distributions specify the probability of reaching a specific environment as a result of faults during the execution.

The unreliable environment distributions are discrete because E is a countable set (Lemma 15.4). Therefore, φ' can be defined pointwise: for any environment $\varepsilon'_u \in \mathrm{E}$, the value of $\varphi'(\varepsilon'_u)$ is the probability that the unreliable execution of the statement s results in the environment ε'_u given the distribution on possible starting environments, φ, and the aggregate probability p_u of reaching ε'_u from any starting environment $\varepsilon_u \in \mathrm{E}$ according to the big-step aggregate semantics. In general, φ' is a subprobability measure because it is defined using the big-step aggregate semantics, which is also a subprobability measure (Section 15.3.4).

Lemma 15.4 (Discrete Distribution)**.** *The probability space of unreliable environments* (E, φ) *is discrete.*

Sketch A probability distribution is discrete if it is defined on a countable sample space. Therefore, we need to prove that the set E is countable. We can accomplish it by proving that both the stack and the heap are countable. We demonstrate the former by observing that the number of variables in each stack frame is finite, and the number of frames is countable. We demonstrate the latter by noting that the number of locations is finite, and each is of the finite size. Full proof is available in Carbin et al. (n.d.). □

15.4.2 Reliability Predicates and Transformers

The paired execution semantics enables a definition of 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:

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

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

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

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

Semantics of Reliability Predicates.

Figure 15.8 presents the denotational semantics of reliability predicates via the semantic function $[\![P]\!]$. The denotation of a reliability predicate is the set of configurations that satisfy the predicate. A key new element in the semantics of this predicate language is the semantics of joint reliability factors.

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

Auxiliary Definitions We define predicate satisfaction and validity as:

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

Reliability Transformer

Given a semantics for predicates, it is now possible to 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 (Dijkstra, 1975).

Definition 15.5 (Reliability Transformer)**.**

$\psi \models \{P\}\ s\ \{Q\} \equiv$

$\forall \varepsilon. \forall \varphi. \forall \varepsilon'. \forall \varphi'. (\langle \varepsilon, \varphi \rangle \models P \wedge \langle s, \langle \varepsilon, \varphi \rangle \rangle \Downarrow_{\psi} \langle \varepsilon', \varphi' \rangle) \Rightarrow \langle \varepsilon', \varphi' \rangle \models Q$

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

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

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

15.5 Reliability Analysis

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

15.5.1 Preliminaries

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

- **Conditional Flattening:** Each conditional has a unique *control flow variable* ℓ associated with it that $\mathcal{T}$ uses to flatten a conditional of the form $\texttt{if}_\ell\ (b)\ \{s_1\}\ \{s_2\}$ to the sequence $\ell = b\ ;\ \texttt{if}_\ell\ (\ell)\ \{s_1\}\ \{s_2\}$. This transformation reifies the control flow variable as an explicit program variable that records the value of the conditional.
- **SSA:** The transformation function also transforms a Rely program to a SSA renamed version of the program. The ϕ-nodes for a conditional include a reference to the control flow variable for the conditional. For example, $\mathcal{T}$ transforms a sequence of statements of the form

  $$\ell = b\ ;\ \texttt{if}_\ell\ (\ell)\ \{x = 1\}\ \{x = 2\}$$

  to the sequence of statements

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

$$\ell = b \;;\; \texttt{if}_\ell\ (\ell)\ \{x_1 = 1\}\ \{x_2 = 2\} \;;\; x = \phi(\ell, x_1, x_2).$$

We rely on standard treatments for the semantics of ϕ-nodes (Barthe et al., 2012) and arrays (Knobe and Sarkar, 1998). We also note that $\mathcal{T}$ applies the SSA transformation such that a reference of a parameter at any point in the body of the function refers to its initial value on entry to the function. This property naturally gives a function's reliability specifications a semantics that refers to the reliability of variables on entry to the function.

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

Auxiliary Maps The map $\Lambda \in \text{Var} \to M$ is a map from program variables to their declared memory regions. We compute this map by inspecting the parameter and variable declarations in the function. The map $\Gamma \in \text{Var} \to R$ is the unique map from the outputs of a function – namely, the return value and arrays passed as parameters – to the reliability factors (Section 15.4.2) for the developer-provided specification of each output. We allocate a fresh variable named ret that represents the return value of the program.

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

15.5.2 Precondition Generation

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

Figure 15.9 presents a selection of Rely's reliability precondition generation rules. The generator takes as input a statement s, a postcondition Q, and (implicitly) the maps Λ and Γ. The generator produces as output a precondition P, such that if P holds before the paired execution of s, then Q holds after.

We have designed 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

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

$$\rho \in (Exp + BExp) \to \mathbb{Q}_{\geq 0} \times \mathcal{P}(\text{Var}) \qquad \rho(n) = (1, \varnothing) \qquad \rho(x) = (\psi(rd(\Lambda(x))), \{x\})$$

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

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

Figure 15.9 Reliability Precondition Generation

that must hold at the end of execution of a function is that each array must be at least as reliable as implied by its specification. The analysis captures this property by setting the initial Q for the body of a function to

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

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

Reasoning about Expressions

The topmost part of Figure 15.9 first presents the 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 $\rho \in (Exp + BExp) \to \mathbb{Q}_{\geq 0} \times \mathcal{P}(\text{Var})$ 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

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

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. The rules for boolean and relational operations are defined analogously.

Generation Rules for Statements

As in a precondition generator, the analysis works backwards from the end of the program to the beginning. We have therefore structured the discussion of the statements starting with function returns.

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

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

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

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

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

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

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

Phi-nodes The rule for a ϕ-node $x = \phi(\ell, x_1, x_2)$ captures the implicit dependence of the effects of control flow on the value of a variable x. For the merged value x, the rule establishes Q by generating a precondition that ensures that Q holds independently for both x_1 and x_2, given an appropriate substitution. Note that the rule also includes ℓ in the substitution; this explicitly captures x's dependence on ℓ. The flattening statement inserted before a conditional (Section 15.5.1), later replaces the reliability of ℓ with that of its dependencies.

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

Unbounded `while` We present the analysis for unbounded `while` loops in the section that follows.

Function Calls The analysis for functions is modular and takes the reliability specification from the function declaration and substitutes the reliabilities of the function's formal arguments with the reliabilities of the expressions that represent the function's actual arguments. We presented the rule for function calls in Carbin et al. (n.d.).

Unbounded `while` *Loops*

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

Dependence Graph The 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. The analysis additionally classifies each edge as reliable or unreliable meaning that a reliable or unreliable operation creates the dependence.

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

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

- **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. The analysis classifies such an edge as reliable if every operation in the assignment (i.e., the operations in the expression and the write to memory) are reliable. Otherwise, the analysis marks the edge as unreliable. The rules for array load and store statements are similar, and include dependencies induced by the computation of array indices.
- **Control Flow Side Effects:** y is assigned within an `if` statement and the `if` statement's control flow variable is named x; this condition captures control dependencies. The analysis classifies each such edge as reliable.

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

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

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

Lemma 15.6 (Termination)**.** *Iteration of $F(A)$ terminates.*

This follows from the monotonicity of RP and the fact that the range of $F(A)$ is finite (given a simplifier that removes redundant predicates and produces a canonical, symbolic predicate representation – which we present a subsumption-based variant in Section 15.5.3) – together, forming finite descending chains. The key intuition is that the set of rational constants in the precondition before and after an iteration does not change (because all variables are reliably updated) and the set of variables that can occur in a joint reliability factor is finite. Therefore, there are a finite number of unique preconditions in the range of $F(A)$.

Final Precondition In the last step, the analysis produces a final precondition that preserves the reliability of variables that are reliably updated by conjoining Q'_r

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

with the predicate $Q_u[(R_i \leq 0)/(R_i \leq R_j)]$, where R_i and R_j are joint reliability factors. The substitution on Q_u sets the joint reliability factors that contain unreliably updated variables to zero.

Properties

Rely's analysis is sound with respect to the transformer semantics presented in Section 15.4.

Theorem 15.7 (Soundness)**.** $\psi \models \{RP_\psi(s, Q)\}\ s\ \{Q\}$

This theorem states that if a configuration $\langle \varepsilon, \varphi \rangle$ satisfies a generated precondition and the paired execution of s yields a configuration $\langle \varepsilon', \varphi' \rangle$, then $\langle \varepsilon', \varphi' \rangle$ satisfies Q. Alternatively, s transforms the precondition generated by the analysis to Q.

We demonstrate the basic constructions for reasoning about soundness of the analysis via a detailed presentation of the soundness of the rule for assignment.

Lemma 15.8 (Soundness of Assignment)**.**

$$\begin{gathered} \psi \models \{A \leq \rho_{\mathrm{p}}(e) \cdot \psi(wr(x)) \cdot \mathcal{R}(X/\{x\} \cup \rho_{\mathrm{var}}(e))\} \\ x = e \\ \{A \leq \mathcal{R}(X)\} \end{gathered}$$

Outline By the definition of the reliability transformer, this judgment is equivalent to proving that $A \leq [\![\mathcal{R}(X)]\!](\varepsilon', \varphi')$ given the two premises:
(1) $A \leq [\![\rho_{\mathrm{p}}(e) \cdot \psi(wr(x)) \cdot \mathcal{R}(Y)]\!](\varepsilon, \varphi)$ and
(2) $\langle s, \varepsilon, \varphi \rangle \Downarrow_\psi \langle \varepsilon', \varphi' \rangle$ where $Y = (X - \{x\}) \cup \rho_{\mathrm{var}}(e)$.

We establish this theorem by proving that

$$[\![\rho_{\mathrm{p}}(e) \cdot \psi(wr(x)) \cdot \mathcal{R}(Y)]\!](\varepsilon, \varphi) \leq [\![\mathcal{R}(X)]\!](\varepsilon', \varphi')$$

and then using the transitivity of $\leq$, namely, that $A \leq [\![\rho_{\mathrm{p}}(e) \cdot \psi(wr(x)) \cdot \mathcal{R}(Y)]\!](\varepsilon, \varphi) \leq [\![\mathcal{R}(X)]\!](\varepsilon', \varphi')$ This follows from the following definitions and lemmas.

Lemma 15.9 (Initial Reliability)**.**
If $ins = \{\varepsilon_u \mid \mathrm{equiv}(\varepsilon, \varepsilon_u, Y)\}$ *then* $[\![\mathcal{R}(Y)]\!](\varepsilon, \varphi) = \sum_{\varepsilon_u \in ins} \varphi(\varepsilon_u)$.

This lemma is a restatement of the semantics of joint reliability factors as laid out in Section 15.4.

Lemma 15.10 (Final Reliability)**.**
If $outs \subseteq \{\varepsilon'_u \mid \mathrm{equiv}(\varepsilon', \varepsilon'_u, X)\}$ *then* $\sum_{\varepsilon_u \in outs} \varphi'(\varepsilon_u) \leq [\![\mathcal{R}(X)]\!](\varepsilon', \varphi')$.

This lemma is also a restatement of the semantics of joint reliability factors.

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

Definition 15.11 (Unreliable Execution Summary)**.** An *unreliable execution summary* is a tuple $(\varepsilon_u, \tau, p, \varepsilon'_u) \in U = \{(\varepsilon_u, \tau, p, \varepsilon'_u) \mid \langle x = e, \varepsilon_u \rangle \overset{\tau,p}{\Longrightarrow}_\psi \varepsilon'_u\}$ such that from an environment ε_u, execution of the statement $x = e$ under the reliability model ψ proceeds following a trace τ with probability p and yields an environment ε'_u.

Unreliable execution summaries enable us to construct a conservative approximation of the paired execution semantics:

Lemma 15.12 (Paired Execution Approximation)**.**
If $execs \subseteq U$ *and* $\langle s, \varepsilon, \varphi \rangle \Downarrow_\psi \langle \varepsilon', \varphi' \rangle$ *then*

$$\Big(\sum_{ex \in execs} \varphi(\pi_{\varepsilon_u}(ex)) \cdot \pi_p(ex) \Big) \leq \sum_{ex \in (execs)} \varphi'(\pi_{\varepsilon'_u}(ex)).$$

This lemma states that for any set of execution summaries *execs* the sum – over all summaries – of the product of the probability of each summary's initial state (according to φ) and the probability of the execution's trace p is less than or equal to the sum of the probability of each summary's final state according to φ'.

This lemma follows from the definition of the paired execution semantics provided in Section 15.4. According to the paired execution semantics, the probability of any final state ε'_u – i.e., $\varphi'(\varepsilon'_u)$ – is the sum over all states ε_u of the aggregate probability (as defined by the aggregate big-step semantics) that the unreliable program reaches ε'_u. Additionally, the aggregatate probability is the sum over all unreliable traces that reach ε'_u from ε_u. This sum is therefore bounded from below by the sum over any subset of all states and traces.

Definition 15.13 (Fully Reliable Execution Summaries)**.** Let the set of fully reliable execution summaries be

$$execs_c = \{(\varepsilon_u, \tau, p, \varepsilon'_u) \mid (\varepsilon_u, \tau, p, \varepsilon'_u) \in U \wedge \ \varepsilon_u \in \{\varepsilon_u \mid \text{equiv}(\varepsilon, \varepsilon_u, Y)\} \wedge \text{correct}(\tau)\},$$

where $\text{correct}(\tau)$ is a predicate that is true only if the trace τ is a list of the form C^+ (C transition label indicates that a transition executed reliably).

The set of fully reliable execution summaries characterizes the set of pairs of environments ε_u and ε'_u where ε_u has the correct values for the set of variables Y and execution from ε_u proceeds fully reliably to ε'_u.

Proof Using the set of fully reliable executions, we can build our proof of the main

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

theorem. Via Lemma 15.9, we know that

$$\begin{aligned}[\![\rho_{\mathrm{p}}(e) \cdot \psi(wr(x)) \cdot \mathcal{R}(Y)]\!](\varepsilon, \varphi) &= \rho_{\mathrm{p}}(e) \cdot \psi(wr(x)) \cdot \sum_{\varepsilon_u \in ins} \varphi(\varepsilon_u) \\ &= \sum_{\varepsilon_u \in ins} \varphi(\varepsilon_u) \cdot \rho_{\mathrm{p}}(e) \cdot \psi(wr(x)) \\ &= \sum_{ex \in execs_c} \varphi(\pi_{\varepsilon_u}(ex)) \cdot \pi_p(ex)\end{aligned}$$

This fact is true because all initial environments ε_u have the same values for all variables in Y and $\rho_{\mathrm{p}}(e) \cdot \psi(wr(x))$ is the probability that the statement executes correctly.

Continuing on from this right-hand side, we use the paired execution approximation and Lemma 15.10 to complete our proof:

$$\begin{aligned}\sum_{ex \in execs_c} \varphi(\pi_{\varepsilon_u}(ex)) \cdot \pi_p(ex) &\leq \sum_{ex \in execs_c} \varphi'(\pi_{\varepsilon'_u}(ex)). \\ &\leq [\![\mathcal{R}(X)]\!](\varepsilon', \varphi')\end{aligned}$$

The first step follows from the fact that $execs_c \subseteq U$. The second step fact follows from the fact that $\pi_{\varepsilon'_u}(execs_c) \subseteq \{\varepsilon'_u \mid \mathrm{equiv}(\varepsilon', \varepsilon'_u, X)\}$. We know that $\pi_{\varepsilon'_u}(execs_c) \subseteq \{\varepsilon'_u \mid \mathrm{equiv}(\varepsilon', \varepsilon'_u, X)\}$. because if (1) all values referenced in e have the correct value and (2) both e and the assignment to x execute reliably, then x has the correct value (and the remaining variables in X have the same correct values as they have not been modified). We can therefore conclude that $A \leq [\![\rho_{\mathrm{p}}(e) \cdot \psi(wr(x)) \cdot \mathcal{R}(Y)]\!](\varepsilon, \varphi) \leq [\![\mathcal{R}(X)]\!](\varepsilon', \varphi')$ □

15.5.3 Specification Checking

As the last step of the analysis for a function, the analysis checks the developer-provided reliability specifications for the function's outputs as captured by the precondition generator's final precondition. Because each specification has the form $r \cdot \mathcal{R}(X)$ (Figure 15.1) the precondition is a conjunction of predicates of the form $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$. While these joint reliability factors represent arbitrary and potentially complex distributions of the values of X_1 and X_2, there is a simple and sound (though not complete) procedure to check the validity of each predicate in a precondition that follows from the *ordering* of joint reliability factors.

Proposition 15.14 (Ordering)**.** *For two sets of variables X and Y, if $X \subseteq Y$ then $\mathcal{R}(Y) \leq \mathcal{R}(X)$.*

The proposition states that the joint reliability of a set of variables Y is less than or equal to the joint reliability of any subset of the variables – regardless of the distribution of their values.

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

Proof (Sketch) First, we consider the case when all variables in X and Y are scalars. Let U_Y be the set passed as the argument at the base case of the recursion started by the call $\text{rel}(Y, \varepsilon, \varphi, \text{E})$ and U_X be the set passed as the argument at the base case of the recursion started by the call $\text{rel}(X, \varepsilon, \varphi, \text{E})$. Then, if $X \subseteq Y$, the set $U_Y \subseteq U_X$, since the variables in $Y \setminus X$ provide additional restrictions on the states that are contained in U_Y. The theorem statement follows from the inequality $\sum_{v \in U_X} \varphi(v) \geq \sum_{v \in U_Y} \varphi(v)$.

If a is an array variable, then the function rel adds a constraint for each element of a. Then, we can apply the same argument for each such obtained sets U_X and U_Y. This property holds for each array element, and is not affected by the minimum operator in the function rel. □

As a consequence of the ordering of joint reliability factors, there is a simple and sound method to check the validity of a predicate.

Corollary 15.15 (Predicate Validity)**.** *If* $r_1 \leq r_2$ *and* $X_2 \subseteq X_1$ *then* $\models r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$.

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

Checking Because the predicates in the precondition generator's output are mutually independent, it is possible to use Corollary 15.15 to check the validity of the full precondition by checking the validity of each predicate.

Our implementation performs simplification transformations after each precondition generator step to simplify numerical expressions and remove predicates that are trivially valid or *subsumed* by another predicate.

Proposition 15.16 (Predicate Subsumption)*A predicate* $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$ *subsumes (i.e., soundly replaces) another predicate* $r_1' \cdot \mathcal{R}(X_1') \leq r_2' \cdot \mathcal{R}(X_2')$ *if* $r_1' \cdot \mathcal{R}(X_1') \leq r_1 \cdot \mathcal{R}(X_1)$ *and* $r_2 \cdot \mathcal{R}(X_2) \leq r_2' \cdot \mathcal{R}(X_2')$.

This property follows directly from the ordering of joint reliability factors. We provide the proof in Carbin et al. (n.d., Section C.3).

15.6 Related Work

Integrity Almost all approximate computations have critical regions that must execute without error for the computation as a whole to execute acceptably. *Dynamic criticality analyses* automatically change different regions of the computation or internal data structures, and observe how the change affects the program's output, e.g. Rinard (2006); Carbin and Rinard (2010); Misailovic et al. (2010). In addition,

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

specification-based *static criticality analyses* let the developer identify and separate critical and approximate program regions, e.g., Liu et al. (2011); Sampson et al. (2011). Carbin et al. (2012) present a verification system for relaxed approximate programs based on a relational Hoare logic. The system enables rigorous reasoning about the integrity and worst-case accuracy properties of a program's approximate regions.

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

Accuracy In addition to reasoning about how often a computation may produce a correct result, it may also be desirable to reason about the accuracy of the result that the computation produces. Dynamic techniques observe the accuracy impact of program transformations, e.g., Rinard (2006), Misailovic et al. (2010), Ansel et al. (2011), Baek and Chilimbi (2010), Sidiroglou et al. (2011), or injected soft errors, e.g., de Kruijf et al. (2010), Liu et al. (2011), Sampson et al. (2011). Empirical techniques like Approxilyzer (Venkatagiri et al., 2015, 2019) present systematic exploration of the impact of soft errors on individual program instructions, including the accuracy of the result. Researchers have developed static techniques that use probabilistic reasoning to characterize the accuracy impact of various sources of uncertainty (Misailovic et al., 2011; Chaudhuri et al., 2011; Zhu et al., 2012). And of course, the accuracy impact of the floating point approximation to real arithmetic has been extensively studied in numerical analysis.

Fault Tolerance and Resilience Researchers have developed various software, hardware, or mixed approaches for detection and recovery from specific types of soft errors that guarantee a reliable program execution, e.g., Reis et al. (2005), Perry et al. (2007), de Kruijf et al. (2010). For example, Reis et al. (2005) present a compiler that replicates a computation to detect and recover from single event upsets. These techniques are complementary to Rely – each can provide implementations of operations that need to be reliable (as specified by the developer or required by Rely) to preserve memory safety and control flow integrity.

Follow up works Since publishing the original paper (Carbin et al., 2013b), we and other researchers have extended this research in various directions. We developed the Chisel optimization system to automate the placement of approximate operations and data (Misailovic et al., 2014). Chisel extends the Rely reliability specifications

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

(that capture acceptable frequency of errors) with absolute error specifications (that also capture acceptable magnitude of errors). It formulates an integer optimization problem to automatically navigate the tradeoff space and generate an approximate computation that provides maximum energy savings while satisfying both the reliability and absolute error specifications.

Several static analyses studied the interactions between safety and reliability. Decaf (Boston et al., 2015) presents a type system that incorporates reliability specifications with EnerJ type annotations. FlexJava (Park et al., 2015) presents a static analysis for inferring annotations on approximate variables. Leto (Boston et al., 2018) provides a flexible interface for expressing custom hardware error models and a verification framework that can prove various properties about programs that execute on such hardware. Aloe (Joshi et al., 2020) adds support for analyzing recovery blocks to Rely.

More recently, researchers also extended the reliability analysis to other kinds of computations. Hung et al. (2019) extend the reliability analysis to quantum programs. Fernando et al. (2019) presented an approach for analyzing message-passing parallel programs with unreliable computation and/or communication.

15.7 Conclusion

The software and hardware communities have grown accustomed to the digital abstraction of computing: the computing substrate is designed to either faithfully execute an operation or detect and report that an error has occurred. This abstraction has enabled a process whereby increased performance capability in the substrate enables the development of increasingly larger and more complicated computing systems that are composed of less complicated, modularly-specified components.

Emerging trends in the scalability of existing hardware design techniques, however, jeopardize the hope that future gains in computing performance will still be accompanied by a digital abstraction. Instead, future high-performance computing platforms may produce uncertain results and, therefore, it may no longer be possible to use traditional techniques to modularly compose components to execute on these platforms.

While there is an immediate opportunity for our work to enable the reasoning needed to reliably achieve better performance in the face of uncertainty, the true motivation for this work is that the nature of computing itself has changed. Emerging applications, such as machine learning, multimedia, and data analytics are inherently uncertain computations that operate over uncertain inputs. Moreover, emerging uncertain computational substrates, such as intermittently powered devices, biological devices, and quantum computing, create new possibilities for where computation can take place and even what can be computed itself.

https://doi.org/10.1017/9781108770750.016 Published online by Cambridge University Press

Going forward, this work will enable the software and hardware communities to discard the notion that they must rely on the digital abstraction to build computing systems. Instead, emerging computing systems will use abstractions of acceptability that will enable these systems to exploit not only the performance benefits of uncertain substrates, but also the new possibilities that these platforms offer for computation.

Acknowledgments

We thank Martin Rinard, our advisor and the co-authors of the conference version of this chapter (Carbin et al., 2013b). We also thank Vimuth Fernando for proofreading the draft. 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

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

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

Barthe, G., Demange, D., and Pichardie, D. 2012. A formally verified SSA-Based middle-end: Static single assignment meets compcert. ESOP.

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

Boston, Brett, Sampson, Adrian, Grossman, Dan, and Ceze, Luis. 2015. Probability type inference for flexible approximate programming. In: *OOPSLA*.

Boston, Brett, Gong, Zoe, and Carbin, Michael. 2018. Leto: verifying application-specific hardware fault tolerance with programmable execution models. In: *OOPSLA*.

Carbin, M., and Rinard, M. 2010. Automatically Identifying Critical Input Regions and Code in Applications. ISSTA.

Carbin, M., Misailovic, S., and Rinard, M. *Verifying Quantitative Reliability of Programs that Execute on Unreliable Hardware (Appendix)*. http://groups.csail.mit.edu/pac/rely.

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

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

Carbin, M., Misailovic, S., and Rinard, M. 2013b. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. OOPSLA.

Chaudhuri, S., Gulwani, S., Lublinerman, R., and Navidpour, S. 2011. Proving Programs Robust. FSE.

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

Dijkstra, Edsger W. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, **18**(August), 453–457.

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

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

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

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

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

Hoffman, H., S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and Rinard, M. 2011. Dynamic Knobs for Responsive Power-Aware Computing. ASPLOS.

Hung, Shih-Han, Hietala, Kesha, Zhu, Shaopeng, Ying, Mingsheng, Hicks, Michael, and Wu, Xiaodi. 2019. Quantitative robustness analysis of quantum programs. *Proceedings of the ACM on Programming Languages*, **3**(POPL), 31.

Joshi, Keyur, Fernando, Vimuth, and Misailovic, Sasa. 2020. Aloe: Verifying Reliability of Approximate Programs in the Presence of Recovery Mechanisms. CGO. ACM.

Knobe, K., and Sarkar, V. 1998. Array SSA form and its use in parallelization. POPL.

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

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

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

Liu, S., Pattabiraman, K., Moscibroda, T., and Zorn, B. 2011. Flikker: Saving DRAM refresh-power through critical data partitioning. ASPLOS.

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

Misailovic, S., Roy, D., and Rinard, M. 2011. Probabilistically Accurate Program Transformations. SAS.

Misailovic, S., Kim, D., and Rinard, M. 2013. Parallelizing Sequential Programs With Statistical Accuracy Tests. *ACM TECS Special Issue on Probabilistic Embedded Computing*.

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

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

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

Park, Jongse, Esmaeilzadeh, Hadi, Zhang, Xin, Naik, Mayur, and Harris, William. 2015. Flexjava: Language support for safe and modular approximate programming. In: *FSE*.

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

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

Reis, G., Chang, J., Vachharajani, N., Rangan, R., and August, D. 2005. SWIFT: Software Implemented Fault Tolerance. CGO.

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

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

Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. 2011. EnerJ: approximate data types for safe and general low-power computation. PLDI.

Schlesinger, C., Pattabiraman, K., Swamy, N., Walker, D., and Zorn, B. 2011. YARRA: An Extension to C for Data Integrity and Partial Safety. CSF.

Sidiroglou, S., Misailovic, S., Hoffmann, H., and Rinard, M. 2011. Managing Performance vs. Accuracy Trade-offs With Loop Perforation. FSE.

Thomas, A., and Pattabiraman, K. 2013. Error Detector Placement for Soft Computation. DSN.

Venkatagiri, Radha, Mahmoud, Abdulrahman, Hari, Siva Kumar Sastry, and Adve, Sarita V. 2015. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In: *MICRO*.

Venkatagiri, Radha, Ahmed, Khalique, Mahmoud, Abdulrahman, Misailovic, Sasa, Marinov, Darko, Fletcher, Christopher W, and Adve, Sarita V. 2019. gem5-Approxilyzer: An Open-Source Tool for Application-Level Soft Error Analysis. In: *DSN*.

x264. 2013. http://www.videolan.org/x264.html.

Zhu, Z., Misailovic, S., Kelner, J., and Rinard, M. 2012. Randomized Accuracy-Aware Program Transformations for Efficient Approximate Computations. POPL.