DOI:10.1145/2958738

Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware

By Michael Carbin, Sasa Misailovic, and Martin C. Rinard

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.

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.

In response to this situation, researchers have developed numerous techniques for detecting and masking errors in both hardware[10] and software.[9,23,24] 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 often acceptably tolerate occasional errors in its execution and/or the data that it manipulates.[7, 20, 25] A *checkable computation* can be augmented with an efficient checker that verifies either the exact correctness[4, 14] or the approximate acceptability[1] of the results that the computation produces. If the checker does detect an error, it can re-execute 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) fast and energy efficient execution that (2) delivers acceptably accurate results often enough to satisfy the needs of their users.

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,[7, 20, 25] 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.[26]

Software Techniques: Most software techniques deploy *unsound transformations*—transformations that change the semantics of an original exact program. Proposed mechanisms include skipping tasks,[16, 25] loop perforation (skipping iterations of time-consuming loops),[20, 29] sampling reduction inputs,[30] multiple selectable implementations of a given component or components,[2, 3, 12, 30] dynamic knobs (configuration parameters that can be changed as the program executes)[12] and synchronization elimination (forgoing synchronization not required to produce an acceptably accurate result).[16, 18] The results show that aggressive techniques such as loop perforation can deliver up to a fourfold 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

The original version of this paper appeared in *Proceedings of the 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Indianapolis, IN, Oct. 2013).

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.[9, 11, 13, 15, 21, 22, 27]

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 behavior, 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).[7, 25] Existing systems, tools, and type systems have focused on helping developers identify, separate, and reason about the binary distinction between critical and approximate regions.[7, 11, 15, 25, 27] 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 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 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 (e.g., the computation must return a result within a given range). One integrity property for our example program 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.[19, 30] 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 this article we focus on reliability properties for approximate computations executing on unreliable hardware platforms. In other research, we have developed techniques for reasoning about integrity properties[5, 6] and both worst-case and probabilistic accuracy properties.[5, 19, 30] We have also extended the research presented in this article to include combinations of reliability and accuracy properties.[17]

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—that is, the probability with which the computation produces a correct result when parts of the computation execute on unreliable hardware with soft errors (independent errors that occur nondeterministically with some probability). 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).[7, 25] 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 article specifically presents the following contributions:

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

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

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

2. EXAMPLE

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

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.

[a] x264 (http://www.videolan.org/x264.html).

Figure 1. Rely code for motion estimation computation.

```
 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  }
```

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 the video's encoded size. However, previous studies on soft error injection[9] and more aggressive transformations like loop perforation[20, 29] have demonstrated that the quality of x264's final result is only slightly affected by perturbations of this computation.

2.1. Reliability specifications

The function declaration on Line 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 reliabilities of its inputs. A reliability specification has the form $r \cdot \mathcal{R}(X)$, where r is a real number between 0 and 1 and X is a set of variables. For example, the specification for the reliability of search_ref's result is int<0.99*R(pblocks,cblock)>. This specification states that the return value is an integer with a reliability that is at least 99% of the *joint reliability* of the parameters pblocks and cblock (denoted by R(pblocks, cblock)). The joint reliability of a set of parameters is the probability that they all have the correct value when passed in from the caller. This specification holds for all possible values of the joint reliability of pblocks and cblock. For instance, if the contents of the arrays pblocks and cblock are fully reliable (correct with probability one), then the return value is correct with probability 0.99.

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

Joint reliabilities serve as an abstraction of a function's input distribution, which enables Rely's analysis to be both modular and oblivious to the exact shape of the distributions. This is important because (1) such exact shapes can be difficult for developers to identify and specify and (2) known tractable classes of probability distributions are not closed under many operations found in standard programming languages, which can complicate attempts to develop compositional analyses that work with such exact shapes.[19, 28]

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 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 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, that is, 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`.

2.3. Hardware semantics

Figure 2 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. Figure 2 presents physically separate reliable and unreliable functional units, but this distinction can be achieved through other mechanisms, such as dual-voltage architectures.[11] 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.

To prevent the execution from taking control flow edges that are not in the program's static control flow graph, the control unit of the CPU reliably fetches, decodes, and schedules instructions (as is supported by existing unreliable processor architectures[11, 27]). In addition, 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 2 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.[15, 27]

2.4. Hardware reliability specification

Rely's analysis works with a hardware reliability specification that specifies the reliability of arithmetic/logical and memory operations. Figure 3 presents a hardware reliability specification that is inspired by results from the existing computer architecture literature.[10, 15] Each entry specifies the reliability—the probability of a correct execution—of arithmetic operations (e.g., `+.`) and memory read/write operations.

For ALU operations, the presented reliability specification uses the reliability of an unreliable multiplication operation from Ref.[10], Figure 9. For memory operations, the specification uses the probability of a bit flip in a memory cell from Ref.[15], 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`).

Figure 2. Machine model. Orange boxes represent unreliable components.

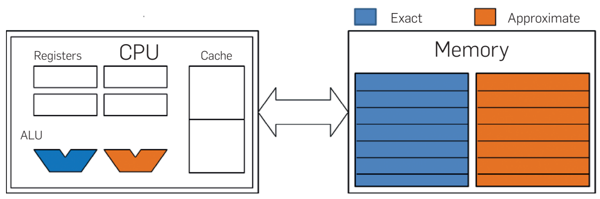
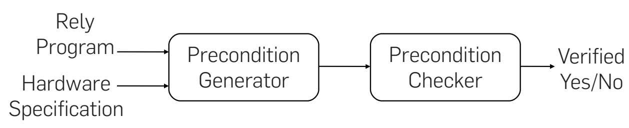


Figure 3. Hardware reliability specification.

```
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 4. Rely analysis overview.



2.5. Reliability analysis

Given a Rely program, Rely's reliability analysis verifies that the each function in the program satisfies its reliability specification when executed on unreliable hardware. Figure 4 presents an overview of Rely's analysis. The analysis consists of two components: the *precondition generator* and the *precondition checker*.

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

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

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

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

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

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. The analysis produces this precondition by starting at the end of the function from a postcondition that must be true when the function returns and then working 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`—$\mathcal{R}$(`pblocks`) and $\mathcal{R}$(`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 comes from the design principle for reliability approximation. Specifically, this reliability is the probability of correctly reading `minblock` from unreliable memory—which is $r_0 = 1 - 10^{-7}$ according to the hardware reliability specification—multiplied by $\mathcal{R}$(`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.

Conditionals: The analysis of the body of the loop on Line 15 encounters the `if` statement on Line 30.[b] 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 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 analyzes 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.

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

Figure 5. if statement analysis in the last loop iteration.

```
(3)  {$Q_0 \wedge A_{ret} \le r_0^4 \cdot \mathcal{R}(\texttt{i, ssd, minssd})$
        $\wedge A_{ret} \le r_0^4 \cdot \mathcal{R}(\texttt{minblock, ssd, minssd})$}
     if (ssd <.  minssd) {
(2)      {$Q_0 \wedge A_{ret} \le r_0 \cdot \mathcal{R}(\texttt{i}, \ell_{30})$}
         minssd = ssd;
         {$Q_0 \wedge A_{ret} \le r_0 \cdot \mathcal{R}(\texttt{i}, \ell_{30})$}
         minblock = i;
         {$Q_0 \wedge A_{ret} \le r_0 \cdot \mathcal{R}(\texttt{minblock}, \ell_{30})$}
     } else {
(2)      {$Q_0 \wedge A_{ret} \le r_0 \cdot \mathcal{R}(\texttt{minblock}, \ell_{30})$}
         skip;
         {$Q_0 \wedge A_{ret} \le r_0 \cdot \mathcal{R}(\texttt{minblock}, \ell_{30})$}
     }
(1)  {$Q_0 \wedge A_{ret} \le r_0 \cdot \mathcal{R}(\texttt{minblock}, \ell_{30})$}
```

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` ($\mathcal{R}$(`ssd, 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:

$$\begin{aligned} Q_2 \;=\; & Q_0 \wedge A_{ret} \le r_0^4 \cdot \mathcal{R}(\texttt{i}, \texttt{ssd}, \texttt{minssd}) \\ & \wedge A_{ret} \le r_0^4 \cdot \mathcal{R}(\texttt{minblock}, \texttt{ssd}, \texttt{minssd}). \end{aligned}$$

Simplification: After unrolling a single iteration of the loop that begins at Line 15, the analysis produces $Q_0 \wedge A_{ret} \le r_0^{2564} \cdot \mathcal{R}(\texttt{pblocks, cblock, i, ssd, minssd})$ 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 $A_{ret} \le r_1 \cdot \mathcal{R}(X_1)$ *subsumes* another predicate $A_{ret} \le r_2 \cdot \mathcal{R}(X_2)$. Specifically, the analysis identifies that $r_1 \le r_2$ and $X_2 \subseteq X_1$, which together mean that the second predicate is a weaker constraint on A_{ret} than the first and can therefore be removed. This follows from the fact that the joint reliability of a set of variables is less than or equal to the joint reliability of any subset of the variables—regardless of the distribution of their values.

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

Final Precondition: When the analysis reaches the beginning of the function after fully unrolling the loop on Line 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

$$\begin{aligned} & A_{ret} \le \; 0.994885 \cdot \mathcal{R}(\texttt{pblocks}, \texttt{cblock}) \;\wedge \\ & A_{\texttt{pblocks}} \le \; \mathcal{R}(\texttt{pblocks}) \;\wedge \\ & A_{\texttt{cblock}} \le \; \mathcal{R}(\texttt{cblock}). \end{aligned}$$

Precondition checker. The final precondition is a conjunction of predicates of the form $A_{out} \le r \cdot \mathcal{R}(X)$, where A_{out} is a placeholder for the reliability specification of an output. Because reliability specifications are all of the form $r \cdot \mathcal{R}(X)$, each predicate in the final precondition (where each A_{out} is replaced with its specification) is of the form form $r_1 \cdot \mathcal{R}(X_1) \le r_2 \cdot \mathcal{R}(X_2)$, 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 (see Precondition checker section), 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 following predicates:

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

Because these predicates are valid according to the checking procedure, search_ref satisfies its reliability specification when executed on the specified unreliable hardware.

3. CASE STUDIES

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

Benchmarks: We analyze the following six computations:

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

3.1. Deriving reliability specification

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

Checkable Computations: Checkable computations can be augmented with an efficient checker that dynamically verifies the correctness of the computation's result. If the checker detects an error, then it re-executes the computation or executes an alternative reliable implementation. For instance, a Newton's method computation searches for value of input x for which a function $f(x)$ is 0. Once this computation finds a zero of the function, x_0, it is typically much less expensive to compute $f(x_0)$ and check if it equals 0.

Quantitative reliability enables a developer to model the performance of this checked implementation of the computation. We will use T_{pass} to denote the expected time required to compute the correct value of the computation and perform the check and T_{fail} to denote the expected time required to compute an incorrect result, perform the check, and then rerun the reliable version of the computation (that produces the correct result).

If r denotes the reliability of the computation, then the expected execution time of the checked computation as a whole is $T' = r \cdot T_{pass} + (1 - r) \cdot T_{fail}$. This time can be compared with the time to always perform a reliable version of the computation. Therefore, this reasoning allows a developer to find the reliability r that meets the developer's performance improvement goal and can be analogously applied for alternative resource usage measures, such as energy consumption and throughput.

Approximate Computations: For computations that are inherently approximate, we can perform reliability profiling to relate the errors in the approximate computational kernels to the full application's errors.

To estimate the error of the computation, a developer can provide a *sensitivity testing procedure*, that specifies how the noise can be injected in the application. For instance, to estimate the error of the function search_ref from Figure 1, a profiler can modify the program to produce the correct minimum distance block with probability r and produce the maximum distance block with probability $1 - r$. This modification provides a conservative estimate of the bound on search_ref's accuracy loss given the reliability r (when the computation's inputs are reliable) and the assumption that a fault causes search_ref to return the worst-case result.

The profiler can then run the application on representative inputs, for different values of r. The profiler then compares the outputs of the original and modified program by computing a developer-provided application level quality-loss-metric. The profiler estimates the relationship between computation-level error, controlled by r, and application-level quality loss. Based on this estimate, the developer can select an appropriate value of r. In our example, if the developer is willing to accept 1% loss in the video's peak-signal-to-noise ratio (the quality-loss metric for the video encoder), then this procedure can help the developer select r to be 0.98.

Benchmark analysis summary.

				Predicates	
Benchmark	**Type**	**LOC**	**Time (ms)**	**N**	**S**
newton	Checkable	21	8	82	1
bisect	Checkable	30	7	16,356	2
coord	Checkable	36	19	20	1
search_ref	Approximate	37	348	36,205	3
matvec	Approximate	32	110	1061	4
hadamard	Approximate	87	18	3	3

3.2. Analysis summary

The table here presents Rely's analysis results on the benchmark computations. For each benchmark, the table presents the type of the computation (checkable or approximate), its length in lines of code (LOC), the execution time of the analysis, and the number of inequality predicates in the final precondition produced by the precondition generator both without and with our simplification strategy.

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

Number of Predicates: We used Rely with the hardware reliability specification from Figure 3 to generate a reliability precondition for each benchmark. The second to last column (labeled N) presents the number of predicates in the precondition when using a naïve strategy that does not include our simplification procedure. The rightmost column (labeled S) presents the number of predicates in each precondition when Rely employs our simplification procedure.

When Rely uses simplification, the size of each precondition is small (all consisting of less than five predicates). The difference in size between the naïvely generated preconditions and those generated via simplification demonstrates that simplification reduces the size of preconditions by multiple orders of magnitude. Simplification achieves these results by identifying that many of the additional predicates introduced by the reasoning required for conditionals can be removed. These additional predicates are often subsumed by another predicate.

4. RELATED WORK

In this section, we present an overview of the other work that intersects with Rely and its contributions to modeling and analysis of approximate computations, and computation fault tolerance.

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, for example, Refs.[7, 20, 25] In addition, specification-based *static criticality analyses* let the developer identify and separate critical and approximate program regions, for example, Refs.[15, 27] Carbin et al.[5] 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, for example, Refs.,[2, 3, 16, 20, 25, 29] or injected soft errors, for example, Refs.[9, 15, 27] Researchers have developed static techniques that use probabilistic reasoning to characterize the accuracy impact of various sources of uncertainty.[8, 19, 30] And of course, the accuracy impact of the floating point approximation to real arithmetic has been extensively studied in numerical analysis.

More recently, we developed the Chisel optimization system to automate the placement of approximate operations and data.[17] Chisel extends the Rely reliability specifications (that capture acceptable frequency of errors) with absolute error specifications (that also capture acceptable magnitude of errors). Chisel formulates an integer optimization problem to automatically navigate the tradeoff space and generate an approximate computation that provides maximum energy savings (for the given model of approximate hardware) while satisfying the developer's reliability and absolute error specifications.

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, for example, Refs.[9, 23, 24] For example, Reis et al.[24] present a compiler that replicates a computation to detect and recover from single event upsets. These techniques are complementary to Rely in that each can provide implementations of operations that need to be reliable, as either specified by the developer or as required by Rely, to preserve memory safety and control flow integrity.

5. CONCLUSION

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

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

Acknowledgments

We thank Deokhwan Kim, Hank Hoffmann, Vladimir Kiriansky, Stelios Sidiroglou, and Rishabh Singh for their insightful comments.

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

References

1. Achour, S., Rinard, M. Approximate checkers for approximate computations in topaz. In *OOPSLA* (2015).
2. Ansel, J., Wong, Y., Chan, C., Olszewski, M., Edelman, A., Amarasinghe, S. Language and compiler support for auto-tuning variable-accuracy algorithms. In *CGO* (2011).
3. Baek, W., Chilimbi, T.M. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI* (2010).
4. Blum, M., Kanna, S. Designing programs that check their work. In *STOC* (1989).
5. Carbin, M., Kim, D., Misailovic, S., Rinard, M. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI* (2012).
6. Carbin, M., Kim, D., Misailovic, S., Rinard, M. Verified integrity properties for safe approximate program transformations. In *PEPM* (2013).
7. Carbin, M., Rinard, M. Automatically identifying critical input regions and code in applications. In *ISSTA* (2010).
8. Chaudhuri, S., Gulwani, S., Lublinerman, R., Navidpour, S. Proving programs robust. In *FSE* (2011).
9. de Kruijf, M., Nomura, S., Sankaralingam, K. Relax: An architectural framework for software recovery of hardware faults. In *ISCA* (2010).
10. Ernst, D., Kim, N.S., Das, S., Pant, S., Rao, R., Pham, T., Ziesler, C., Blaauw, D., Austin, T., Flautner, K., Mudge, T. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO* (2003).
11. Esmaeilzadeh, H., Sampson, A., Ceze, L., Burger, D. Architecture support for disciplined approximate programming. In *ASPLOS* (2012).
12. Hoffman, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., Rinard, M. Dynamic knobs for responsive power-aware computing. In *ASPLOS* (2011).
13. Leem, L., Cho, H., Bau, J., Jacobson, Q., Mitra, S. Ersa: Error resilient system architecture for probabilistic applications. In *DATE* (2010).
14. Leveson, N., Cha, S., Knight, J.C., Shimeall, T. The use of self checks and voting in software error detection: An empirical study. In *IEEE TSE* (1990).
15. Liu, S., Pattabiraman, K., Moscibroda, T., Zorn, B. Flikker: Saving dram refresh-power through critical data partitioning. In *ASPLOS* (2011).
16. Meng, J., Chakradhar, S., Raghunathan, A. Best-effort parallel execution framework for recognition and mining applications. In *IPDPS* (2009).
17. Misailovic, S., Carbin, M., Achour, S., Qi, Z., Rinard, M. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *OOPSLA* (2014).
18. Misailovic, S., Kim, D., Rinard, M. Parallelizing sequential programs with statistical accuracy tests. *ACM TECS Special Iss. Prob. Embedded Comput*. (2013).
19. Misailovic, S., Roy, D., Rinard, M. Probabilistically accurate program transformations. In *SAS* (2011).
20. Misailovic, S., Sidiroglou, S., Hoffmann, H., Rinard, M. Quality of service profiling. In *ICSE* (2010).
21. Narayanan, S., Sartori, J., Kumar, R., Jones, D. Scalable stochastic processors. In *DATE* (2010).
22. Palem, K. Energy aware computing through probabilistic switching: A study of limits. *IEEE Trans. Comput.* (2005).
23. Perry, F., Mackey, L., Reis, G., Ligatti, J., August, D., Walker, D. Fault-tolerant typed assembly language. In *PLDI* (2007).
24. Reis, G., Chang, J., Vachharajani, N., Rangan, R., August, D. Swift: Software implemented fault tolerance. In *CGO* (2005).
25. Rinard, M. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS* (2006).
26. Rinard, M., Cadar, C., Dumitran, D., Roy, D., Leu, T., Beebee, W. Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI* (2004).
27. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI* (2011).
28. Sankaranarayanan, S., Chakarov, A., Gulwani, S. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *PLDI* (2013).
29. Sidiroglou, S., Misailovic, S., Hoffmann, H., Rinard, M. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE* (2011).
30. Zhu, Z., Misailovic, S., Kelner, J., Rinard, M. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL* (2012).

Michael Carbin, Sasa Misailovic, and Martin C. Rinard, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.