Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2014-001 January 9, 2014

Reliability-Aware Optimization of Approximate Computational Kernels with Rely

Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin Rinard

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

Reliability-Aware Optimization of Approximate Computational Kernels with Rely

Sasa Misailovic Michael Carbin Sara Achour Zichao Qi Martin Rinard
MIT CSAIL
{misailo,mcarbin,sachour,zichaoqi,rinard}@csail.mit.edu

Abstract

Emerging high-performance architectures are anticipated to contain unreliable components (e.g., ALUs) that offer low power consumption at the expense of soft errors. Some applications (such as multimedia processing, machine learning, and big data analytics) can often naturally tolerate soft errors and can therefore trade accuracy of their results for reduced energy consumption by utilizing these unreliable hardware components.

We present and evaluate a technique for reliability-aware optimization of approximate computational kernel implementations. Our technique takes a standard implementation of a computation and automatically replaces some of its arithmetic operations with unreliable versions that consume less power, but may produce incorrect results with some probability.

Our technique works with a developer-provided specification of the required reliability of a computation – the probability that it returns the correct result – and produces an unreliable implementation that satisfies that specification.

We evaluate our approach on five applications from the image processing, numerical analysis, and financial analysis domains and demonstrate how our technique enables automatic exploration of the trade-off between the reliability of a computation and its performance.

1. Introduction

Researchers anticipate that the aggressive scaling of hardware feature sizes in the search for better performance will produce systems that are increasingly susceptible to *soft errors* [2, 25]. While many researchers have developed techniques for detecting and masking soft errors [5, 6, 19–21] (which typically incur time or energy overhead), some researchers have seen this development as an opportunity to develop tools and techniques that enable applications that are naturally tolerant to soft errors to execute on unreliable hardware with increased performance and reduced power-consumption [3, 4, 12, 22, 24].

Carbin et al. have proposed a language and analysis system, Rely, that enables developers to manually adapt their application to use unreliable hardware operations, write a specification for the desired reliability of their application (the probability that it produces the same result as if executed on reliable hardware), and then verify that the unreliable implementation of their application provably satisfies its reliability specification using the system's static analysis [3].

Rely is designed to work with a hardware platform that provides reliable and unreliable versions of standard arithmetic and logical operations as well as reliable and unreliable memories. In the previous Rely model of computation, the developer is responsible for identifying which operations must execute reliably and which may execute unreliably. The developer is also responsible for identifying which variables must be allocated in reliable memory and which operations may be allocated in unreliable memory. The Rely analysis system then verifies that the program satisfies the desired reliability specification.

While Rely provides a sound analysis for reasoning about the reliability of an unreliable implementation of a program, it presents a burden on the developer in that he or she needs to manually identify the reliable and unreliable operations in the program.

We present a new system that removes this burden from the developer. Starting with a computational kernel of an application and a specification of the desired reliability of this kernel, this new system automatically selects which operations in this kernel can execute unreliably. The resulting program maximizes energy savings or performance while still satisfying the desired reliability specification:

Optimization Algorithm. We present an optimization algorithm that casts the unreliable operation placement problem as an integer linear program. The integer linear program uses 1) a unique zero-one valued variable for each operation in a computational kernel that indicates whether the operation is reliable (zero) or unreliable (one) and 2) an objective function that minimizes the energy consumption of the program over a set of profiled execution traces given a setting of the variables then determine if an operation is unreliable.

Machine Model. We present an abstract machine model for programs that execute on unreliable hardware. The model uses a specification of the reliability and power consumption of each operation to give a precise definition of both the semantics and power consumption of an execution of an unreliable machine.

The machine model is also *failure-oblivious* [23] in that programs cannot halt due to error (e.g., invalid memory accesses). This semantics enables our optimization algorithm to make operations such as array index calculations unreliable without fear of halting the execution of the program.

Power Model for Optimization. We formalize a power model for our optimization algorithm that takes as input specifications of the reliability of each instruction class (integer, floating-point, and other non-arithmetic instructions), along with specifications of the relative power consumption of each instruction class and the savings associated when an instruction is selected to run in an unreliable mode. This power model provides a precise description of the architectural parameters that our optimization algorithm needs to operate.

Experimental Results. We have implemented our optimization algorithm within the Rely compiler [3]. We use this modified Rely compiler to automatically optimize versions of a set of benchmark computational kernels to run on unreliable hardware. The Rely compiler can work with two different scenarios for reasoning about soft errors: 1) selective fault tolerance in systems that use dual-modular redundancy [20] to achieve high-reliability and 2) approximate hardware designs that provide unreliable but more

power efficient versions of standard arithmetic operations and memories.

Our experimental results show that our implemented optimization algorithm enables the modified Rely compiler to successfully optimize our set of benchmark kernels to profitably exploit unreliable hardware platforms while preserving important reliability guarantees.

2. Example

Next, we present an example of how to apply our optimization algorithm to produce an approximate image scaling implementation.

2.1 Image Scaling

Figure 1 presents an implementation of the core part of an algorithm to scale an image to a larger size. The function `scale` takes as input the scaling factor `f` (a scaling factor of two doubles the image size in both dimensions), along with an integer array `src` that represents the contents of the image to be scaled and another integer array `dest` that represents the contents of the final scaled result.

The algorithm calculates the value of each pixel in the final result by mapping the pixel's location back to the original source image and then taking a weighted average of the neighboring pixels. The code for `scale` implements the outside portion of the algorithm where it enumerates over the height (`d_height`) and width (`d_width`) of the destination image. For each pixel accessed by `i` and `j` in the destination image, the algorithm keeps track of the corresponding location in the source image with the variables `si` and `sj`.

The function `scale_kernel` implements the core kernel of the scaling algorithm. On Lines 5-8, the algorithm computes a neighborhood of four nearby pixels and then fetches their pixels values on Lines 10-13. To average the pixel values together, the algorithm uses *bilinear interpolation*. Bilinear interpolation takes the weighted average of the each of the four neighboring pixels where the weight is given by the distance from the source coordinates `si` and `sj` to the location of each of the pixels. The algorithm computes these weights on Lines 15-18.

In the last step, the algorithm extracts each RGB color component of the pixel, computes the weighted average, and then returns the result (Lines 20-35).

2.2 Profiling and Specification.

The first step in the Rely workflow is to write reliability specifications for computations that are amenable to approximation and profitable to exploit. One way to achieve this is by taking the following steps:

Time Profiling. A developer will first profile his or her application using standard techniques to identify the regions of code in which his or her application spends the most time [17]. The full image scaling program consists of the code presented in Figure 1 along with the additional code to handle command line parameters and read/write images files. However, the program spends the majority of its time within `scale`'s inner loop.

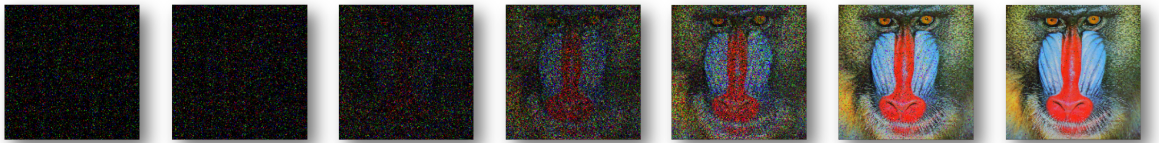
Reliability Profiling. The developer will next extract a computation into its own function or functions. This extraction allows the developer to 1) define an interface for the computation and 2) perform fault injection at the interface boundary to determine if the rest of the application is resilient to inaccuracies that could result from running the computation with approximation. For `scale`, we have already extracted its kernel into the function `scale_kernel`.

Given an extracted computation, a developer can then directly test if the full application is resilient to errors in the extracted computation. For example, we can implement this for `scale_kernel` with the following straightforward strategy:

```
 1  int scale_kernel(
 2    float i, float j,
 3    int[] src, int s_width)
 4  {
 5    int previ = floor(si);
 6    int nexti = ceil(si);
 7    int prevj = floor(sj);
 8    int nextj = ceil(sj);
 9
10    int ul = src[IDX(nexti, nextj, s_width)];
11    int ul = src[IDX(nexti, prevj, s_width)];
12    int lr = src[IDX(previ, prevj, s_width)];
13    int ll = src[IDX(previ, nextj, s_width)];
14
15    float ul_w = (cj - prevj) * (ci - previ);
16    float ur_w = (nextj - cj) * (ci - previ);
17    float lr_w = (nextj - cj) * (nexti - ci);
18    float ll_w = (cj - prevj) * (nexti - ci);
19
20    float coeff = 1.0/((nexti - previ) *
21                  (nextj - prevj));
22
23    int r = (int) coeff*
24             (ul_w * R(ul) + ur_w * R(ur) +
25              lr_w * R(lr) + ll_w * R(ll));
26
27    int g = (int) coeff *
28             (ul_w * G(ul) + ur_w * G(ur) +
29              lr_w * G(lr) + ll_w * G(ll));
30
31    int b = (int) coeff *
32             (ul_w * B(ul) + ur_w * B(ur) +
33              lr_w * B(lr) + ll_w * B(ll));
34
35    return COMBINE(r, g, b);
36  }
37
38  void
39  scale(
40     float f,
41     int[] src, int s_width,
42     int[] dest, int d_height, int d_width)
43  {
44    float delta = 1 / f;
45    float si = 0;
46
47    for (int i = 0; i < d_height; ++i) {
48      float sj = 0;
49      for (int j = 0; j < d_width; ++j) {
50
51        dest[IDX(i, j, d_width)] =
52          scale_kernel(si, sj, src, s_width);
53
54        sj += delta;
55      }
56      si += delta;
57    }
58  }
```

Figure 1: Rely Code for Image Scaling Kernel

```
#define R ...
int scale_kernel_with_faults(
  float i, float j, int[] src, int s_width)
{
   if (bernoulli(R)) {
      return scale_kernel(i, j, src, s_width);
   } else {
     return rand_int();
   }
}
```



(a) 0.20 (b) 0.40 (c) 0.60 (d) 0.80 (e) 0.90 (f) 0.99 (g) 0.999

Figure 2: Reliability Profiling for Image Scaling for Different Values of R

The function `scale_kernel_with_faults` uses a Bernoulli variable (as implemented by `bernoulli`) to return the correct result with probability R and return an incorrect result with probability 1 − R. Using this implementation, a developer can use a representative test suite to plot the quality of the whole program's result as a function of R.

Figure 2 presents a visual depiction of the results of scaling for different values of R. Note that very unreliable implementations (0.20-0.90) do not produce high quality results. However, as R reaches values in the range of 0.999 and above, the results may be an acceptable approximation of the result of a fully reliable implementation (with R = 1).

A developer can also provide a quantitative evaluation of the quality of the results via domain-specific quality metrics that compare the quality of an approximate output versus the intended output. For `scale` we use Peak-Signal-to-Noise Ratio (PSNR), which is a common quality metric in the image processing domain.

Using the quality metric and the plot, the developer can then set an appropriate reliability target for `scale_kernel`. For the remainder of this section, we use 0.99 as `scale_kernel`'s target reliability, which yields an image with a PSNR of 32dB when compared to the reference image.[1]

Reliability Specification. Given a reliability target, the developer next provides a reliability specification for the kernel. Using the reliability target of 0.99 for `scale_kernel` yields the following new signature for the function:

```
int<0.99 * R(i, j, src, s_width)>
scale_kernel
(float i, float j, int[] src, int s_width);
```

The additional annotation `0.99 * R(i, j, src, s_width)` on the return value specifies the reliability of the result. This annotation specifies two components:

- **Input Dependencies.** The reliability of the output of a function is a function of the reliability of the function's inputs. The term `R(i, j, src, s_width)` abstracts the *joint reliability* of the inputs on entry to the function, which is the probability that they all together contain the correct result.
- **Reliability Degradation.** The coefficient 0.99 expresses the *reliability degradation* of the function. Specifically, given a value for the joint reliability of the function's inputs, the coefficient expresses with what fraction of that probability that the return value is correct.

These two components together give a sound specification for the probability that the result of `scale_kernel` is correct.

Arrays. Rely also supports allocating arrays in unreliable memory. For example, the developer may place the pixel array `src` in an unreliable memory region named `urel` using the following signature for `scale_kernel`:

```
int<0.99 * R(i, j, src, s_width)>
scale_kernel
(float i, float j, int[] in urel src, int s_width);
```

[1] A PSNR of greater than 30dB is typically considered of acceptable quality.

Our optimization algorithm factors in the possibility of error in the reads of `src` when optimization the program. Specifically, the optimization algorithm will consider both the scenario when `src` is allocated in reliable memory and the case when `src` is allocated in unreliable memory. It will then report to the developer, if there is a different allocation strategy that gives better energy savings for the kernel. Specifically, it may be more profitable to allocate `src` in reliable memory and make additional arithmetic operations in the kernel unreliable.

2.3 Hardware Specification

Given the implementation of a computational kernel and the developer's reliability specification, our optimization algorithm will next automatically optimize the implementation by replacing some (or all) of the arithmetic operations in the kernel with unreliable versions. To do this, the optimization algorithm requires a *hardware specification* that provides the following:

Operation Reliability. The hardware specification specifies for which arithmetic operations the hardware exposes unreliable versions of an operation, along with their reliability.

Memory Reliability. The hardware specification specifies a list of unreliable memory regions in which a developer may allocate data, along with the reliability of reads and writes to each region.

Power Model Parameters. To build an power model for the program, the optimization algorithm needs 1) a specification of the relative portion of system energy consumed by the CPU versus memory, 2) the relative portion of the CPU energy consumed by the arithmetic-processing unit versus other, on-chip resources (e.g., registers and cache), and 3) the ratio in average power consumption of floating-point instructions and other non-arithmetic instructions relative to integer instructions.

Further, to compute the savings associated with selecting unreliable versions of an arithmetic operation, the optimization problem requires specifications of the expected power savings of executing an unreliable version of an operation as a percentage of the energy of a reliable version.

2.4 Reliability-Aware Optimization Algorithm

Our optimization algorithm casts the unreliable operation placement problem as an integer linear program. The integer linear program uses 1) a unique zero-one valued variable for each arithmetic operation in a kernel that indicates whether the operation is reliable or unreliable and 2) an objective function that characterizes the energy consumption of the kernel over a set of profiled execution traces. We use an integer linear program solver to find a configuration of the variables that minimizes the kernel's energy consumption.

Our algorithm additionally adapts Rely's reliability analysis to add an additional *reliability constraint* to the optimization problem such that all valid solutions to the integer linear program give a configuration of operations that ensures that the kernel satisfies the kernel's reliability specification.

Results. If we instantiate the optimization algorithm on the hardware specification given in Section 5.3, then the optimization algorithm achieves 23.9% power savings in the ALU. Though we do not present the resulting code here, the optimization algorithm sets the

$$r \in \mathcal{R} \cup \{\text{pc}, \text{bp}, \text{pw}\} \qquad n \in \text{Int}_\text{N}$$
$$a \in A \subseteq \text{Int}_\text{N} \qquad \kappa \in K = \{0,1\}$$
$$op \in O ::= \texttt{add} \mid \texttt{fadd} \mid \texttt{mul} \mid \texttt{fmul} \mid \texttt{cmp} \mid \texttt{fcmp} \mid \ldots$$
$$i \in I ::= r = op^\kappa\ r\ r \mid \texttt{jmp}\ r\ r \mid$$
$$r = \texttt{load}\ n \mid \texttt{store}\ n\ r \mid$$
$$r = \texttt{loada}\ r\ r \mid \texttt{storea}\ r\ r\ r$$

Figure 3: Assembly Language Syntax

vast majority of the arithmetic operations in `scale_kernel` to be unreliable – including the array index calculations on Lines 10-13.

Failure-Oblivious. A key aspect of our optimization approach is the realization that computations that are typically considered critical to the correctness of the program (such as array index calculations) may dominate a large portion of the computation. Specifically, 79% of all integer instructions in `scale_kernel` are used to compute the indices on lines 10-13.

Our hardware and compilation model therefore makes approximate computations *failure-oblivious* [23], meaning that they never fail due an invalid memory access, so that one can still approximate those computations. Instead, such invalid operations simply return a non-deterministic result and then continue executing.

3. Compilation and Semantics

The code of `scale` in Section 2 shows the syntax of Rely's language, which is a pointer-less C-like language with first-class single dimension arrays and reliability specifications. In this section, we present a machine model and a compilation model for Rely that captures the basic properties of unreliable hardware. We then present the hardware specifications required to do reliability-aware optimization.

3.1 Machine Model

Figure 3 presents the abbreviated syntax of the assembly language of a simple load/store machine. The instructions consist of ALU and FPU operations (such as add, multiply and compare), conditional branches, loads and stores from stack offsets (`load` and `store`) and also loads and stores from heap allocated arrays (`loada` and `storea`). Each operand of an instruction is stored in a register $r \in \mathcal{R}$ or – in the case for loads and stores – is a fixed N-bit (e.g., 32-bit or 64-bit) integer $n \in \text{Int}_\text{N}$.

Each arithmetic instruction is also affixed with a kind $\kappa \in K = \{0,1\}$ – such as $r = \texttt{add}^\kappa\ r_1\ r_2$ – that indicates that the instruction is either reliable ($\kappa = 0$) – and always produces the correct result – or unreliable ($\kappa = 1$) – and may therefore produce an incorrect result with some probability.

3.2 Compilation

Rely has a straightforward mapping from its source code to our target machine.

Expressions. Compilation of expressions proceeds in a standard way; we flatten expressions and store intermediate results in registers. We assume the register file is the same size as the maximum number of concurrently live intermediate values in the program.

Local Variables. We allocate local variables in a stack that we place in a reliable region of main system memory. Local variables are accessed via load and store instructions, which are allocated at fixed offsets from a frame base pointer, bp.

Arrays. We allocate arrays in main system memory. Given a hardware specification, main system memory may export multiple memory regions of different reliability. We allocate each array in the memory region specified by the developer.

To support bounds-checking, arrays include an additional header that specifies the length of the array in memory. Specifically, the address associated with an array points to a two-word structure that contains the address of the array's data and its length. This allocation strategy allows us to separate the reliability of the data stored in the array from the reliability of its metadata and, therefore, the contents of an array can be stored in unreliable memory.

3.3 Machine Semantics

Before we present the semantics of the machine's instructions, we first present several preliminary definitions.

Register Files, Memories, Programs, and Environments. A *register file* $\sigma \in \Sigma = R \to \text{Int}_\text{N}$ is a finite map from registers to machine integers. A *memory* $h \in H = A \to \text{Int}_\text{N}$ is a finite map from addresses to machine integers. A *program* $\gamma \in \Gamma = A \to I$ is a finite map from addresses to instructions. An environment $\varepsilon \in \text{E} = \Sigma \times H$ is a pair of a register file and a memory.

Arrays. Given the address of an array's metadata a, the function $\text{len} \in A \to \text{Int}_\text{N}$ returns the address of the length field in the array's metadata.

Ready Instructions and Configurations. A *ready instruction* is either an instruction $i \in I$ or the distinguished element "$\cdot$" that indicates that no instruction is ready to execute. A *configuration* is a ready instruction and environment pair.

Hardware Model. A hardware model $\psi \in (O \to \mathbb{R}) \times (A \times M_{op} \to \mathbb{R} \times \mathbb{R}) \times (I \to \text{Int}_\text{N}))$ is a triple of finite maps.

The projection π_op selects the first element of the hardware model, which is a finite map from operations to reliabilities. The reliability of an operation is the probability that the operation executes correctly (as given by a hardware specification).

The projection π_mem selects the second element of the hardware model, which is a finite map from address-*memory operation* pairs to reliabilities. A memory operation $mop \in M_{op} = \{ld, st\}$ is either a load (ld) or a store (st). A function call $\pi_\text{mem}(\psi)(a, mop)$ therefore gives the reliability of performing the memory operation mop on the address a.

The projection π_pwr selects the third element of the hardware model, which is a finite map from instructions to power consumptions as quantified by an integer unit value. This map therefore describes the power consumption of each individual operation.

Auxiliary Probability Distributions. 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. While this distribution is inherently tied to the properties of the underlying hardware, we define this distribution only to support a precise formalization of the dynamic semantics of a program; it does not need to be specified for a given hardware platform.

3.3.1 Rules

Figure 4 presents the semantics of our machine model. The small-step judgment $\langle \hat{i}, \varepsilon \rangle \xrightarrow[\gamma,\psi]{\lambda,p} \langle \hat{i}', \varepsilon' \rangle$ means that execution of the program γ from the configuration $\langle \hat{i}, \varepsilon \rangle$ takes a transition with label λ with probability p under a hardware model ψ and yields a configuration $\langle \hat{i}, \varepsilon' \rangle$.

A *transition label* $\lambda \in \{\text{C}, \langle \text{F}, n \rangle\}$ characterizes whether the transition executed correctly (C) or experienced a fault ($\langle \text{F}, n \rangle$). The value n that accompanies a faulty transition records the value that the fault inserted into the semantics of the program.

ALU/FPU-C

$$\frac{p = \psi(op)^{\kappa}}{\langle r = op^{\kappa}\ r_1\ r_2, \langle \sigma, h\rangle\rangle \xrightarrow[\gamma,\psi]{C,p} \langle\, \cdot\,, \langle \sigma[r \mapsto op(\sigma(r_1), \sigma(r_2))], h\rangle\rangle}$$

ALU/FPU-F

$$\frac{p = (1 - \pi_{\text{op}}(\psi)(op)) \cdot P_f(n \mid op, n_1, n_2)}{\langle r = op^{1}\ r_1\ r_2, \langle \sigma, h\rangle\rangle \xrightarrow[\gamma,\psi]{\langle \text{F}, n\rangle, p} \langle\, \cdot\,, \langle \sigma[r \mapsto op(\sigma(r_1), \sigma(r_2))], h\rangle\rangle}$$

LOAD-C

$$\frac{a = \sigma(\text{bp}) + n}{\langle r = \texttt{load}\ n, \langle \sigma, h\rangle\rangle \xrightarrow[\gamma,\psi]{C,1} \langle\, \cdot\,, \langle \sigma[r \mapsto h(a)], h\rangle\rangle}$$

STORE-C

$$\frac{a = \sigma(\text{bp}) + n}{\langle \texttt{store}\ n\ r, \langle \sigma, h\rangle\rangle \xrightarrow[\gamma,\psi]{C,1} \langle\, \cdot\,, \langle \sigma, h[a \mapsto \sigma(r)]\rangle\rangle}$$

LOAD-ARR1

$$\frac{a = h(\sigma(r_{arr})) + r_{idx} \qquad a \in \text{dom}(h) \qquad p = \pi_{\text{mem}}(\psi)(a, ld)}{\langle r = \texttt{loada}\ r_{arr}\ r_{idx}, \langle \sigma, h\rangle\rangle \xrightarrow[\gamma,\psi]{C,p} \langle\, \cdot\,, \langle \sigma[r \mapsto h(a)], h\rangle\rangle}$$

LOAD-ARR2

$$\frac{a = h(\sigma(r_{arr})) + r_{idx} \qquad a \notin \text{dom}(h)}{\langle r = \texttt{loada}\ r_{arr}\ r_{idx}, \langle \sigma, h\rangle\rangle \xrightarrow[\gamma,\psi]{C,1} \langle\, \cdot\,, \langle \sigma[r \mapsto n], h\rangle\rangle}$$

STORE-ARR1

$$\frac{n = h(\text{len}(\sigma(r_{arr}))) \qquad 0 \le \sigma(r_{idx}) < n \qquad a = h(\sigma(r_{arr})) + r_{idx} \qquad p = \pi_{\text{mem}}(\psi)(a, st)}{\langle \texttt{storea}\ r_{arr}\ r_{idx}\ r, \langle \sigma, h\rangle\rangle \xrightarrow[\gamma,\psi]{C,p} \langle\, \cdot\,, \langle \sigma, h[a \mapsto \sigma(r)]\rangle\rangle}$$

STORE-ARR2

$$\frac{n = h(\text{len}(\sigma(r_{arr}))) \qquad \neg(0 \le \sigma(r_{idx}) < n)}{\langle \texttt{storea}\ r_{arr}\ r_{idx}\ r, \langle \sigma, h\rangle\rangle \xrightarrow[\gamma,\psi]{C,1} \langle\, \cdot\,, \langle \sigma, h\rangle\rangle}$$

JMP-TRUE

$$\frac{\sigma(r_c) \neq 0}{\langle \texttt{jmp}\ r_c\ r, \langle \sigma, h\rangle\rangle \xrightarrow[\gamma,\psi]{C,1} \langle\, \cdot\,, \langle \sigma[\text{pc} \mapsto \sigma(r)], h\rangle\rangle}$$

JMP-FALSE

$$\frac{\sigma(r_c) = 0}{\langle \texttt{jmp}\ r_c\ r, \langle \sigma, h\rangle\rangle \xrightarrow[\gamma,\psi]{C,1} \langle\, \cdot\,, \langle \sigma, h\rangle\rangle}$$

FETCH

$$\frac{i = \gamma(\sigma(\text{pc}))\ \ \sigma' = \sigma[\text{pc} \mapsto \sigma(\text{pc}) + 1][\text{pw} \mapsto \sigma(\text{pw}) + \pi_{\text{pwr}}(\psi)(i)]}{\langle\, \cdot\,, \langle \sigma, h\rangle\rangle \xrightarrow[\gamma,\psi]{C,1} \langle i, \langle \sigma', h\rangle\rangle}$$

Figure 4: Machine Semantics (Abbreviated)

ALU/FPU. The semantics of an integer or floating-point operation $r = op^{\kappa}\ r_1\ r_2$ takes one of two possibilities:

- **Correct execution.** An operation executes correctly with probability $\pi_{\text{op}}(\psi)(op)^{\kappa}$. Therefore, if the operation is reliable ($\kappa = 0$) it executes correctly with probability 1. If it is unreliable ($\kappa = 1$), then it executes correctly with probability $\pi_{\text{op}}(\psi)(op)$.
  A correct execution proceeds with the rule [ALU/FPU-C] wherein the instruction reads its two registers r_1 and r_2 from the register file, performs the operation and stores the result back in register r.
- **Faulty execution.** An unreliable operation (where $\kappa = 1$) experiences a fault with probability $\pi_{\text{op}}(\psi)(op)$. A faulty execution stores into the destination register r a value n that is given by the errant result distribution for the operation, P_f. An important note here is that while the instruction may experience a fault, its faulty execution does not modify any state besides the provided destination register.

Load/Stores. The semantics of stack loads and stores is fully reliable because the stack is allocated in a reliable memory region.

Array Loads/Stores. The semantics of loads and stores of arrays is failure oblivious in that an out-of-bounds array access never forces the program to halt. These checks are implemented differently for loads and stores.

- **Loads.** Array loads to do not include an explicit bounds check. The rule [LOAD-Arr1] uses the pointer to the array's data $\sigma(r_{arr})$ and the index value passed in as an index (r_{idx}) to compute the address of the data (a). If that address is a valid memory location ($a \in \text{dom}(h)$) then the rule loads the value of the address with probability $\pi_{\text{mem}}(\psi)(a, ld)$. We elide the rule where the read from memory fails.
  The rule [Load-Arr2] reasons about the case when a is not a valid memory address ($a \notin \text{dom}(h)$). In this case, the semantics is free to place any value n into the destination register r. This semantic approach minimizes the overhead incurred on array reads by only requiring an implementation to check if the address is valid. On a modern architecture with virtual memory, this corresponds to modifying the application's page fault handler to handle traps on accesses to invalid addresses.
- **Stores.** Array stores require an array bounds check so that our analysis can perform modular reasoning in the presence of errant array indices. Specifically, if array stores were to write outside of the bounds of an array, then the *frame condition* for an analysis would include the entire program state.

Branches and Fetch. The rules [Fetch], [Jmp-T], and [Jmp-F] implement control flow transfers which all execute reliably. By preserving the reliability of control flow transfers, an approximate program always takes paths that exist in the static control flow of the original program. Note that while the control flow transfers themselves execute reliably, Rely allows the conditional expressions of `jmp` instructions to depend on unreliable computation. Therefore, an unreliable execution of the program may take a different path than a reliable execution of the program.

3.3.2 Reliability

In this section we restate the semantic reliability definitions from [3] for our machine semantics.

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

$$\langle \cdot, \varepsilon\rangle \xRightarrow[\gamma,\psi]{\tau,p} \varepsilon' \equiv \langle \cdot, \varepsilon\rangle \xrightarrow[\gamma,\psi]{\lambda_1,p_1} \cdots \xrightarrow[\gamma,\psi]{\lambda_n,p_n} \langle \texttt{skip}, \varepsilon'\rangle$$
$$\textit{where } \tau = \lambda_1, \ldots, \lambda_n \textit{ and } p = \prod_i p_i$$

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

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

$$\langle \cdot, \varepsilon\rangle \xRightarrow[\gamma,\psi]{p} \varepsilon' \textit{ where } p = \sum_{\tau \in \text{T}} p_\tau \textit{ such that } \langle \cdot, \varepsilon\rangle \xRightarrow[\gamma,\psi]{\tau,p_\tau} \varepsilon'$$

The big-step aggregate semantics enumerates over the set of all finite length traces and sums the aggregate probability that a program γ starts in an environment ε and terminates in an environment ε' given a hardware model ψ.

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

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

The paired execution semantics pairs an execution of the program γ given a fully reliable hardware model 1_ψ (where the reliability of each arithmetic and memory operation is 1.0) with its executions given another potentially unreliable hardware model ψ. Specifically, because the unreliable operations given by ψ introduce a probabilistic semantics for the program, if the executions for the two hardware models start in an environment ε, then if the reliable execution terminates in a ε, then the unreliable execution terminates in a *distribution* of environments, which is given by φ.

The paired execution semantics enables us to give a semantics to the reliability of a variable. Specifically, if we use the final environment ε' for the reliable hardware model and the distribution of final environments for the unreliable hardware model, then the reliability of a variable allocated a memory address a is $\sum_{\varepsilon_u \in \mathcal{E}(a)} \varphi(\varepsilon_u)$ where $\mathcal{E}(a) = \{\varepsilon_u \mid \pi_{\text{heap}}(\varepsilon_u)(a) = \pi_{\text{heap}}(\varepsilon')\}$.

4. Energy Optimization Algorithm

Given the semantics for the machine, we can now express the optimization problem. To do this we first augment our assembly language and provide an intermediate representation that includes *labels*:

$$\ell \in \mathcal{L}$$
$$i \in I ::= r = op^{\ell}\ r\ r$$

In this intermediate representation we augment each arithmetic instruction with a label $\ell \in \mathcal{L}$ instead of a kind. The label of each instruction in a program is unique. We then let a *kind configuration* $\theta \in \Theta = \ell \to K$ be a finite map from instruction addresses to kinds, which allows us to phrase the optimization problem as follows:

Definition 4. *Semantic Optimization Problem.*

$$\min_{\theta} \sum_{\varepsilon \in E_{\text{prf}}} \sum_{\varepsilon'} p \cdot \pi_{\text{pwr}}(\varepsilon') \textit{ where } \langle \cdot, \varepsilon \rangle \xRightarrow[\gamma[\theta],\psi]{p} \langle \cdot, \varepsilon' \rangle$$

This definition states that we are searching for a kind configuration θ such that when we replace the label of each arithmetic instruction with its corresponding kind ($\gamma[\theta]$), we minimize the total expected power consumption of the program over a set of profile inputs E_{prf}. The projection π_{pwr} returns the value of the pwr register in an environment.

4.1 Absolute Energy Model

We next present a model for estimating the absolute power consumption of a kind configuration of a program.

Average Power. Each instruction in the hardware specification may have a different power consumption associated with it. However, for the purposes of our model, let $\mathcal{E}_i$, $\mathcal{E}_f$, $\mathcal{E}_o$ be the average power of an ALU instruction, a FPU instruction, and other non-arithmetic instructions, respectively.

Power of Trace Set. First consider the power consumption of a set of instruction traces collected by profiling a set of inputs. Using the integer arithmetic n_{int}, floating-point n_{fp} , and other non-arithmetic n_o instruction counts, we characterize the total average power for each of the instruction classes:

$$E_{int} = n_{int} \cdot \mathcal{E}_{int}$$
$$E_{fp} = n_{fp} \cdot \mathcal{E}_{fp}$$
$$E_o = n_o \cdot \mathcal{E}_o$$

Together, these also characterize the total average ALU power:

$$E_{ALU} = E_{int} + E_{fp} + E_o$$

Power with Kind Configurations. We next estimate the power consumption given a kind configuration of the program by parameterizing our ALU power model over a given kind configuration.

We describe the set of traces by partitioning the set of instructions into three subsets: *IntInst* (the set of labels of integer arithmetic instructions) and *FPInst* (the set of labels of floating-point arithmetic instructions). For each instruction with a label ℓ, we also let n_a denote the number of times the instruction executes for the set of inputs. Finally, let α_{int} and α_{fp} be the average savings (i.e., percentage reduction in power consumption) from executing integer and floating-point instructions unreliably, respectively.

Using these definitions, we then restate our ALU power definitions as a function of a configuration θ as follows:

$$E_{int}(\theta) = \sum_{\ell \in IntInst} n_\ell \cdot (1 - \theta(\ell) \cdot \alpha_{int}) \cdot \mathcal{E}_{int}$$
$$E_{fp}(\theta) = \sum_{\ell \in FPInst} n_\ell \cdot (1 - \theta(\ell) \cdot \alpha_{fp}) \cdot \mathcal{E}_{fp}$$
$$E_{ALU}(\theta) = E_{int}(\theta) + E_{fp}(\theta) + E_o$$

CPU Energy. We model the energy consumption of the CPU as the combined energy consumed by the ALU and the other on-chip components, including the register file, cache, and other communication devices:

$$E_{CPU} = E_{ALU} + E_{SRAM}$$
$$E_{CPU}(\theta) = E_{ALU}(\theta) + E_{SRAM}$$

The term E_{SRAM} captures the power consumption of the other on-chip components. Note that because we aren't modeling savings for other on-chip components, E_{SRAM} need not be parameterized by θ for our parameterized power consumption model.

Memory Energy. We model the energy consumption of system memory (i.e., DRAM) using an estimate of the average power per second per byte of memory, $\mathcal{E}_{mem}$. Given t (the execution time of the kernel), α_{mem} (the savings associated with allocating data in unreliable memory), S_r and S_u (the number of bytes allocated in the reliable and unreliable memories, respectively), we model the power consumption of memory as follows:

$$E_{mem} = t \cdot \mathcal{E}_{mem} \cdot (S_r + S_u \cdot (1 - \alpha_{mem}))$$

System Energy. We model the energy consumed by the compute system as the combined energy used by the CPU and memory.

$$E_{sys} = E_{CPU} + E_{mem}$$
$$E_{sys}(\theta) = E_{CPU}(\theta) + E_{mem}$$

4.2 Relative Energy Model

Next, we present the numerical optimization problem that we use to optimize Rely programs. While the power model equations from Section 4.1 factor in a number of the concerns for optimizing the power consumption of the CPU, the models rely on several hardware design specific parameters, such as the average power of instructions.

However, we can use these equations to derive a numerical optimization problem that instead uses cross-design parameters (such as the relative power between instruction classes and the average savings for each instruction) to optimize the *relative energy*

of a configuration of the program.

$$\frac{E_{sys}(\theta)}{E_{sys}} = \frac{E_{CPU}(\theta)+E_{mem}}{E_{CPU}+E_{mem}} =$$
$$= \frac{E_{CPU}}{E_{CPU}} \cdot \frac{E_{CPU}(\theta)}{E_{CPU}+E_{mem}} + \frac{E_{mem}}{E_{CPU}+E_{mem}} =$$
$$= \mu_{CPU}\frac{E_{CPU}(\theta)}{E_{CPU}} + (1-\mu_{CPU})$$

Because E_{mem} is the same for both the reliable and the kind configured program, the memory model simplifies to the value 1. Note that this simplification requires the assumption that the kind configured program and the reliable program execute for approximately the same amount of time.

CPU Relative Energy. We apply the same reasoning to model the relative energy consumption of the CPU:

$$\frac{E_{CPU}(\theta)}{E_{CPU}} = \mu_{ALU}\frac{E_{ALU}(\theta)}{E_{ALU}} + (1-\mu_{ALU})$$

ALU Relative Energy. We also apply the same reasoning to model the relative energy consumption of the ALU:

$$\frac{E_{ALU}(\theta)}{E_{ALU}} = \mu_{int} \cdot \frac{E_{int}(\theta)}{E_{int}} + \mu_{fp} \cdot \frac{E_{fp}}{E_{fp}} + \mu_o$$

The parameters μ_{int}, μ_{fp}, and μ_o are computed from the counts of the instructions and the relative energy consumption of each class with respect to that of integer instructions. For example, if we let w_{fp} be the ratio of power consumption between floating point instructions and integer instructions (i.e, $w_{fp} = \frac{\mathcal{E}_{fp}}{\mathcal{E}_{int}}$), then $\mu_{fp} = \frac{w_{fp} \cdot n_{fp}}{n_{int}+w_{fp} \cdot n_{fp}+w_o \cdot n_o}$.

Final Parameters. The final machine parameters we need for this problem are the relative portion of power consumed by the CPU versus memory (μ_{CPU}), the relative power consumed by the ALU versus the other components of the chip (μ_{ALU}) and also the relative power of each instruction class versus an integer instruction(μ_{int}, μ_{fp}, and μ_o) .

4.3 Reliability-Aware Optimization Problem

Our model for relative power consumption leads naturally to a numerical optimization problem that minimizes the relative power consumption of the kind configured program, namely $\min_\theta(\frac{E_{sys}(\theta)}{E})$. By substituting our power model equations for relative savings and removing terms that do not depend on the kind configuration, we arrive at the optimization problem:

$$\min_\theta \left(\mu_{CPU} \cdot \mu_{ALU} \left(\mu_{int} \cdot \frac{E_{int}(\theta)}{E_{int}} + \mu_{fp} \cdot \frac{E_{int}(\theta)}{E_{fp}} \right) \right)$$

This is an *unconstrained* optimization problem, where the optimal solution is a kind configuration where all operations are unreliable. However, setting all operations to be unreliable may violate a computation's reliability bound. We therefore add a *reliability constraint* that constrains valid solutions to be only those that satisfy the computation's reliability bound.

4.3.1 Reliability Constraint

A reliability constraint is a predicate P of the following form:

$$R_f := \rho \mid \rho^\ell \mid \mathcal{R}(V) \mid R_f \cdot R_f$$
$$P := R_f \leq R_f \mid P \wedge P$$

Specifically, a predicate is either a conjunction of predicates or a comparison between *reliability factors*. A reliability factor is either a 1) real-number ρ, 2) a *kinded reliability* ρ^ℓ which expresses the reliability of an operation labeled ℓ as a function of the kind configuration θ and its reliability ρ, 3) the reliability of a set of registers, stack offsets, and arrays $\mathcal{R}(V)$, or 4) a product of reliability factors.

As in Section 3, the meaning of a reliability factor is given by the final environment $\varepsilon \in \mathrm{E}$ of the reliable execution of the program and the distribution of environments $\varphi \in \Phi$ that result from an unreliable execution of the program – as specified by the paired execution relation (Definition 3). Therefore, $[\![P]\!] \in \mathcal{P}(\mathrm{E} \times \Phi \times \Theta)$ and $[\![R_f]\!] \in (\mathrm{E} \times \Phi \times \Theta) \to \mathbb{R}$.

For example, the denotation of $\mathcal{R}(V)$ is the total probability that the unreliable execution halts in an environment where all the registers, stack offsets, and arrays in V have the same value as in the reliable execution of the program. For a kinded reliability, $[\![\rho^\ell]\!](\varepsilon, \varphi, \theta) = \rho^{\theta(\ell)}$, denotes that the reliability of the operation is equal to 1 if $\theta(\ell) = 0$ and ρ if $\theta(\ell) = 1$.

4.3.2 Computing Reliability Constraints

We compute constraints using a precondition generation analysis as is done in [3]. Below we present a selection of the rules for an analyzer $C \in I \to P \to P$ that takes two inputs: an instruction and a reliability constraint postcondition. The analyzer produces a precondition that when satisfied, ensures the post condition holds after the instruction terminates.

$$C(r = op^\ell\ r_1\ r_2, Q) = Q[(\pi_{\mathrm{op}}(\psi)(op)^\ell \cdot \mathcal{R}(\{r_1, r_2\} \cup X))/\mathcal{R}(r \cup X)] \quad (1)$$
$$C(r = \texttt{load}\ n, Q) = Q[\mathcal{R}(n \cup X)/\mathcal{R}(\{r\} \cup X)] \quad (2)$$
$$C(\texttt{store}\ n\ r, Q) = Q[\mathcal{R}(r \cup X)/\mathcal{R}(\{n\} \cup X)] \quad (3)$$

ALU/FPU. Equation 1 presents the generator rule for ALU/FPU operations. The rule works by substituting the reliability of the destination register r with the reliability of its operands and the reliability of the operation itself. The substitution $\mathcal{R}(\{r_1, r_2\} \cup X)/\mathcal{R}(r \cup X)]$ matches all occurrences of the destination register in the set of values in a reliability term and replaces them with the input registers, r_1 and r_2. The substitution additionally multiplies in the factor $\pi_{\mathrm{op}}(\psi)(op)^\ell$, which expresses the reliability of the operation as a function of its kind, and its reliability $\pi_{\mathrm{op}}(\psi)(op)$.

Load/Store. The rules for loads and stores are similar to that for ALU/FPU instructions with the exception that the substitution operates on both stack offsets and registers.

Array Load/Store. We elide the rules for array loads and stores, but they are similar to standard loads and stores with the additional modification that the substitution includes the reliability of the array's index calculation and the reliability of reading from the array's memory region.

Control Flow. We also elide the rules for control flow. Our analysis of `if` conditionals relies on the fact that the high-level Rely language has structured control flow and therefore it is straightforward to use the structure of the high-level program to identify the instructions that correspond to an `if` statement. As in [3], our analysis includes the reliability of an `if` statement's condition when computing the reliability of variables modified under an `if` statement. Our analysis of `while` statements also follows that from [3].

4.3.3 Optimization Problem with Reliability Constraint

For a given kernel, our reliability constraint analysis computes a reliability constraint of the form $\bigwedge_{i,j} \rho_i \cdot \mathcal{R}(V_i) \leq \nu_j \cdot \mathcal{R}(V_i)$ where $\rho_i \cdot \mathcal{R}(V_i)$ is a reliability factor for a developer-provided specification of an output and $\nu_j \cdot \mathcal{R}(V_j)$ is a lower bound on the reliability of that output. Here each ρ_i is a real-valued constant whereas each ν_j is a product of real-valued constants and kinded reliabilities (i.e., quantities of the form ρ^ℓ).

If this constraint is valid for a given kind configuration, then that kind configuration of the program satisfies the developer-provided reliability specification.

Constraint Validity Intuition. To check the validity of this constraint, we can use the observation that the reliability of any subset of the variables V_j is greater than or equal to the reliability of V_j as a whole. Specifically,

$$V_i \subseteq V_j \Rightarrow \mathcal{R}(V_j) \leq \mathcal{R}(V_i)$$

With this observation we can then soundly check the validity of each inequality in the constraint by checking $\rho_i \leq \nu_j$ and $V_i \subseteq V_j$.

Reliability-Aware Optimization Problem. We build the constraint for our reliability-aware numerical optimization problem by preprocessing the reliability constraint and then encoding it into the optimization problem. Specifically, for each inequality conjunct $\rho_i \cdot \mathcal{R}(V_i) \leq \nu_j \cdot \mathcal{R}(V_i)$ in the constraint, we can immediately check if $V_i \subseteq V_j$. This leaves the additional constraint that $\rho_i \leq \nu_j$.

The computed reliability expression ν_j has the form $\Pi_k \rho_k^{\ell_k}$ (where k iterates over all instructions in the trace, computed by the analysis). Therefore, if we take the logarithm of both sides of the inequality, we obtain the expression

$$\log(\rho_i) \leq \sum_k \ell_k \log(\rho_k). \tag{4}$$

We note that the expression on the right side is linear with respect to all labels ℓ_k. Each label is an integer variable that can take a value 0 or 1. The reliabilities ρ_i are constants, and therefore one can compute ahead of time their logarithms.

Final Optimization Problem Statement. We can state the optimization problem given the previous reliability constraint:

$$\begin{array}{ll} \text{Minimize:} & \left(\mu_{CPU} \cdot \mu_{ALU} \left(\mu_{int} \cdot \frac{E_{int}(\theta)}{E_{int}} + \mu_{fp} \cdot \frac{E_{int}(\theta)}{E_{fp}}\right)\right) \\ \text{Constraints:} & \log(\rho_i) \leq \sum_{k_j} \ell_{k_j} \log(\rho_{k_j}) \qquad \forall i, j \\ \text{Variables:} & \ell_1, \ldots, \ell_n \in \{0,1\} \end{array}$$

We note that since the variables $\ell_1, \ldots, \ell_n$ are integers, this optimization problem is an instance of an integer linear program. While in general, this problem is NP complete, the existing solvers that can successfully solve many classes of integer linear programs.

Implementation. We have implemented the reliability-aware optimization framework using OCaml. The framework consists of several passes. The translation pass produces an equivalent C program for an input file with Rely functions. The instrumentation pass instruments the C program to collect traces of instructions that are used to compute the frequencies of instructions in the energy expression, n_{int}, n_{fp}, n_o. The analysis pass computes the objective function and the reliability constraints. We use Gurobi mixed integer programming solver to solve the generated optimization problem [8]. Finally, the transformation pass uses the solution of the optimization program to generate the Rely program with inserted unreliable instructions and specifications of the function's parameters stored in the unreliable memory.

5. Evaluation

We present an evaluation of Rely for two application scenarios: selective fault tolerance for guarding against transient errors and power optimization via compilation to unreliable hardware.

5.1 Benchmarks

To empirically test Rely's effect on program accuracy and power usage, we consider a set of benchmarks from several application domains. The applications were selected because they either tolerate some amount of error in the output or are robust to errors.

- **Scale**: It scales an image by a factor provided by the user. The kernel computation in scale computes the output pixel value by interpolating over neighboring source image pixels.
- **Discrete Cosine Transform (DCT)**: DCT is a popular compression algorithm that is used in various lossy image and audio compression methods. The kernel computation performs the conversion of an 8x8 image subregion into frequency domain coefficients.
- **Inverse Discrete Cosine Transform (IDCT)**: IDCT reconstructs an image from the coefficients generated by the DCT. The kernel computation reconstructs the 8x8 pixel grid from a frequency domain grid.
- **Newton**: The Newton Benchmark finds the root of a function by applying newton's root-finding algorithm to a variety of different initial guesses. The kernel computation finds a root for a single initial point.
- **Black-Scholes**: The Black-Scholes benchmark computes the price of European Put and Call options using the analytical Black-Scholes formula. The kernel computation in Black-Scholes computes the price of an option given its initial value.

Table 1 presents the overview of the benchmark computations. For each computation, Column 2 ("Size") presents the number of lines of code of the benchmark computation. Column 3 ("Kernel") presents the number of lines of kernel computation that is a candidate for optimization. Column 4 ("Instruction in Kernel %") presents the percentage of instructions that the execution spends in the kernel computation. Column 5 ("Representative Input Number") presents the number of representative inputs collected for each computation. Column 6 ("Accuracy Metric") presents the accuracy metric of the computation.

Representative Inputs. For each application, we have selected several representative inputs. The analysis uses these inputs to obtain the estimates of the instruction mixes and construct the objective function of the optimization problem.

Accuracy Metrics. We have used standard accuracy metrics for the benchmark computations. For the three image processing benchmarks we have used peak signal to noise ratio between images that two versions of the benchmark produce. For Newton benchmark we have used absolute difference between the value that the two versions of the computation produce. For Blackscholes benchmark we have used the average of absolute differences of the computed option prices.

Result Checkers. Two of the benchmark computations have built-in checker computations that ensure that the intermediate results of the computation are well formed. Checker computations typically execute for only a small fraction of time of the benchmark. If the checker computation fails, the unreliable computation needs to be re-executed. The Newton checker checks whether the result that the computation produces is a root of the candidate function. The Blackscholes checker uses *no-arbitrage* bound on the price of each option to filter out the executions that produce unrealistically large or small option prices.

5.2 Reliability Specification Computation

We perform the time and reliability profiling to 1) determine the amount of time that the computation spends in the computational kernels and 2) assess the sensitivity of the output of the benchmark computations to different tolerable reliability specifications of their computational kernels.

Benchmark	Size (LoC)	Kernel (LoC)	Instruction in Kernel %	Representative Input Number	Accuracy Metric
scale	218	88	93.43%	13	Peak Signal-to-Noise Ratio
dct	532	50	99.20%	13	Peak Signal-to-Noise Ratio
idct	532	88	98.86%	9	Peak Signal-to-Noise Ratio
newton	169	38	88.54%	5	Absolute Difference
blackscholes	494	143	99.68%	5	Average of Abs. Diff.

Table 1: Benchmark Description

Benchmark	Index Integer %	Remaining Integer %	Floating Point %	Load/Store %	Other %	Optimization Variables	Reliability Constraints
scale	15.76%	4.30%	18.05%	46.99%	14.90%	147	4
dct	25.47%	4.09%	15.73%	41.29%	13.42%	121	1
idct	13.07%	3.11%	26.36%	40.97%	16.49%	104	1
newton	0.00%	8.69%	36.15%	46.47%	8.69%	22	1
blackscholes	0.00%	1.75%	39.77%	50.88%	7.60%	77	2

Table 2: Benchmark Profiling and Optimization Problem Statistics

Execution Profiling. We performed execution profiling of the original computation to guide the search for computation kernels using Callgrind [1]. Column 5 of Table 1 presents the percentage of instructions that the computation spends in the kernel computation relative to the execution time of the overall computation modulo the functions that perform I/O operations on the input and output files. For each benchmark, the kernel computations execute more than 88% of the instructions of the program.

Columns 2 to 6 in Table 2 present the percentages of executed instruction types. We note that the significant portion of time the programs execute arithmetic – integer and floating point – and memory load/store operations. The majority of the other operations, reported in Column 6, are branching and cast operations. The integer operations are divided in those that compute array indices and the operations that compute the kernel's results. We note that in the three benchmarks that operate on arrays, more than 79% of the integer instructions are calculating array indices. Therefore, Rely's support for failure-oblivious computing makes a significant portion of the integer instructions candidates for approximation.

Reliability Constraints. The Rely's reliability analysis constructs the constraints that ensure that the computation meets its reliability specifications. Columns 7 and 8 in Table 2 present the number of binary variables in the generated constraints and the total number of constraints. The number of variables is smaller than 150 and the number of constraints is at most 4. Solving each of the generated optimization problems with Gurobi takes less than a second on an Intel Xeon desktop machine.

Reliability Specifications. To find acceptable reliability specifications, we manually modified each benchmark to return the correct value of the kernel computation with probability equal to the reliability specification and otherwise produce random output values (scalar variables or arrays). The results for the image processing benchmarks are presented in Table 3. The acceptable PSNR rates for image and video compressions are 30 dB and above. Therefore, the developer can select reliability 0.99 as the (acceptable) reliability specification of the kernel computations in IDCT and Scale. Likewise, the developer can select reliability 0.999 as the reliability specification of DCT.

We performed the same experiment for computations with checkers. For 1000 runs of BlackScholes (on an input with 64000 options) and Newton computations we have not observed any error that propagated to the output of the program. However, to offset for the additional execution time of the computation, a developer may select higher reliability specifications.

Benchmark	Reliability	Experimental PSNR
scale	0.99	32.17
	0.999	42.22
	0.9999	52.16
dct	0.99	27.58
	0.999	32.94
	0.9999	38.54
idct	0.99	32.54
	0.999	44.43
	0.9999	49.10

Table 3: Software Specification SNR

5.3 Selective Fault Tolerance

Researchers have investigated a number of techniques for building fault tolerant systems that include both hardware and software techniques [14, 19–21]. Specifically, replication at the instruction level executes the instruction on two redundant functional units and compares the results of the computation. If these results differ, the computation is repeated.

One additional way to improve on the performance of instruction-level replication approach is through *dereplication* where operations that are less critical to the behavior of the program are not replicated so as to trade accuracy of the program's result for increased performance. In this section, we consider selective dereplication to reduce the power consumption of a computation.

Architectural Model. To perform this experiment we consider the modified superscalar design based on the Alpha microprocessor proposed by Ray, et al. [20]. In this design, a single instruction is duplicated in the instruction fetch and decode stage such that two copies of the instruction are placed in the reorder buffer. The execute stage of the design then schedules each instruction on two of the CPUs available functional units. The CPU compares the results of the two functional units; if they are not the same, then the CPU re-executes the original instruction.

Power Specifications. To derive the expressions for power savings of the architecture, we have used the power numbers from [16] to assign costs to each of the stages of executing a standard non-replicated instruction versus replicated instruction. By doubling the energy consumption of the Execute stage in the pipeline we obtain the savings of non-replicated instruction execution of 40% for integer and 37% for floating point arithmetic instructions.

Operation Reliability Specifications. In [13, Figure 1], Mukherjee at al. provide an estimate of 200 to 2000 soft faults caused by radiation per 10^6 hours of operation of an Alpha family microprocessor. Given the frequency of the target CPU of 1 GHz, the probability

Benchmark	Kernel Reliability	ALU Savings
scale	$1-10^{-6}$	23.8%
	$1-10^{-7}$	17.9 %
	$1-10^{-8}$	1.8 %
dct	$1-10^{-3}$	27.0 %
	$1-10^{-4}$	11.4 %
	$1-10^{-5}$	3.1 %
idct	$1-10^{-3}$	23.3%
	$1-10^{-4}$	18.8%
	$1-10^{-5}$	10.9%
newton	$1-10^{-5}$	25.2%
	$1-10^{-6}$	14.8%
	$1-10^{-7}$	13.9%
blackscholes	$1-10^{-6}$	24.1 %
	$1-10^{-7}$	3.5 %
	$1-10^{-8}$	0.4 %

Table 4: Dereplication Results

of a failure per cycle is approximately equal to $r_0 = 10^{-18}$. We take this probability as the baseline tolerable soft fault rate.

Further, we assume that the replicated components in the microprocessor are designed to satisfy this specification on tolerable fault rate. Since the fault is observed in the replicated execution only if both executions fail simultaneously, the probability of failure is equal to $r_0 = r^2$, where r is the fault rate of individual replicated component. Therefore, we will use the fault rate $r = 10^{-9}$ for a reliability of a non-replicated ALU instruction. All memory operations are fully reliable – the computation of the memory addresses is always performed on both functional units.

Optimization. We evaluate the savings of the processor's functional unit for varying tolerable reliability of the computation's kernel. For each specification of the tolerable reliability degradation introduced by the kernel computation, we execute the optimization algorithm presented in Section 4 and estimate the overall savings of the processor's functional unit. Since we are interested in the arithmetic unit computation, we set the architectural factor μ_{CPU} and μ_{ALU} to 1.

Table 4 presents the summary of savings we obtain for the benchmark computations. Column 2 ("Reliability") presents the tolerable reliability of the kernel computation. For each computational kernel we have used three reliability specifications that correspond to the mild, medium, and aggressive reliability goal optimizations (relative to the base reliability rate r. Column 3 ("ALU Savings") presents the ALU savings that the framework finds.

The overall savings of the computations for the aggressive reliability specifications range from 23.3% (idct) to 27% (dct). For this case, the optimization algorithm generates alternative Rely programs in which most arithmetic operations (integer and floating point) are unreliable, i.e., they require only a single functional unit. The numbers in the first row for each benchmark present the maximum savings that the optimization algorithm produces for our computations. For the remaining two (stricter) tolerable reliability bounds, the optimization algorithm generates versions of the computation with fewer unreliable instructions, but always finds the instructions that can be made unreliable.

5.4 Approximate Hardware Configuration Selection

Researchers have previously investigated approximate hardware architectures that can trade reliability or accuracy of the computation for additional energy savings. In this section we focus on the model of approximate hardware proposed by EnerJ [7, 24], which defines an approximate ISA that makes it possible to control unreliable and reliable execution of the computation at the granularity of individual instructions.

Benchmark	Rel.	Mm	mM	MM	mm
scale	0.99	10.44 %	9.01 %	11.26 %	8.73 %
dct	0.999	9.97 %	9.58 %	11.82 %	8.83 %
idct	0.99	10.4 %	8.48 %	10.73 %	8.2 %
newton	0.999	1.19%	1.95%	1.95 %	1.19%
blackscholes	0.999	1.1 %	0.27 %	0.27 %	1.1 %

Table 5: Approximate Hardware Results

Power and Reliability Specifications. We evaluate the expected savings of the computations generated by Rely's optimization algorithm that execute on several hardware configurations presented in [24, Table 2]. This table contains several configurations of unreliable operations and memories (denoted as *mild*, *medium* and *aggressive*) that can potentially trade reliability for energy savings.

We evaluate the analysis on four configurations of the system, which consists of the combination of *mild* (which we denote as "m") and *medium* (which we denote as "M") configurations for integer and floating point instructions and memories from [24, Table 2]. To compute the overall system savings, we use the server configuration parameters specified in [24, Section 5.4].

Optimization. Table 5 presents the results of the optimization of the programs using Rely's optimization algorithm. Column 2 ("Rel.") presents the target reliability that we set according to the exploration in Section 5.2. The following four columns present overall system savings for the combination of memories and instructions: the first letter represents the configuration of the memory (mild or Medium) and the second letter represents the configuration of the arithmetic instructions (mild or Medium). For instance, "Mm" denotes medium memory configuration (capital "M") and mild instruction configuration ("m").

For three out of five benchmarks, the configuration that satisfies the reliability constraints and delivers the maximum energy savings is the one that sets the Medium configuration for both arithmetic instructions and the memory stores. The Newton benchmark has the maximum savings for the medium configuration of the arithmetic instructions. The Blackscholes benchmark is more sensitive to the reliability of the arithmetic instructions; therefore the algorithm finds the mild arithmetic instruction configurations that deliver maximum system savings. We note that for these hardware specifications the majority of the system savings comes from storing data in unreliable memories – the Blackscholes and Newton computations, which do not have array parameters, have smaller overall energy savings.

6. Related Work

Programming Models for Approximate Hardware. Rely [3] is a language that allows specification of computations that execute on unreliable hardware and the analysis that ensures that the computation that executes on unreliable hardware satisfies its reliability specification. Flikker [11] is a set of C language extensions that allows a developer to specify data that can be stored in approximate memories. EnerJ [24] is a language that allows the developer to specify unreliable data that can be stored in unreliable memory or computed using unreliable operations and the type system that ensures the isolation of unreliable computations. Unlike these previous techniques, which are mainly manual, our technique allows for automation of the part of the process of developing applications for unreliable hardware platforms, while the guaranteeing that the developer's reliability specification is satisfied.

Unreliable Hardware Platforms. Researchers have previously developed hardware platforms that explicitly improve the reliability of the computations by trading off energy or performance, e.g., [14, 19, 20]. Researchers have previously proposed multiple hardware architectures that improve the performance of processors [7, 9, 10, 15, 18, 24] or memories [11, 24] at the expense of the increased error rates.

7. Conclusion

As the need for energy-efficient computing becomes more acute, unreliable hardware platforms become an increasingly attractive target for computationally intensive applications that must execute efficiently. But successfully navigating the resulting accuracy versus energy tradeoff space requires precise, detailed, and complex reasoning about how the unreliable hardware platform interacts with the computation to generate these tradeoffs. We present a new system that automatically maps the computation onto the underlying unreliable hardware platform, minimizing energy consumption while ensuring that the computation executes accurately enough. This system is capable of generating significant energy savings while relieving developers of the need to manage the complex, low-level details of assigning different parts of the computation to unreliable hardware components. Such systems are clearly required if developers are to produce software that can effectively exploit modern energy-efficient unreliable hardware platforms.

References

[1] Callgrind (Valgrind Tool). http://valgrind.org/.

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

[3] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. OOPSLA, 2013.

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

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

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

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

[8] Gurobi. http://www.gurobi.com/.

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

[10] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra. Ersa: error resilient system architecture for probabilistic applications. DATE'10.

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

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

[13] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO-36*, 2003.

[14] S.S. Mukherjee, M. Kontz, and S.K. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. ISCA, 2002.

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

[16] K. Natarajan, H. Hanson, S.W. Keckler, C.R. Moore, and D. Burger. Microprocessor pipeline energy analysis. IPSLED, 2003.

[17] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. PLDI, 2007.

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

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

[20] J. Ray, J. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. MICRO, 2001.

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

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

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

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

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