Chisel: Reliability- and Accuracy-Aware Optimization of Approximate Computational Kernels

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

Abstract

The *accuracy* of an approximate computation is the distance between the result that the computation produces and the corresponding fully accurate result. The *reliability* of the computation is the probability that it will produce an acceptably accurate result. Emerging approximate hardware platforms provide *approximate operations* that, in return for reduced energy consumption and/or increased performance, exhibit reduced reliability and/or accuracy.

We present Chisel, a system for reliability- and accuracy-aware optimization of approximate computational kernels that run on approximate hardware platforms. Given a combined reliability and/or accuracy specification, Chisel automatically selects approximate kernel operations to synthesize an approximate computation that minimizes energy consumption while satisfying its reliability and accuracy specification.

We evaluate Chisel on five applications from the image processing, scientific computing, and financial analysis domains. The experimental results show that our implemented optimization algorithm enables Chisel to optimize our set of benchmark kernels to obtain energy savings from 8.7% to 19.8% compared to the original (exact) kernel implementations while preserving important reliability guarantees.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Processors – Optimization

Keywords Approximate Computing

1. Introduction

Emerging approximate hardware platforms provide operations that, in return for reduced energy consumption, may produce less accurate and/or incorrect results [13, 18, 19, 21, 27, 29, 36]. Target application domains include computations that either 1) contain approximate components that naturally tolerate some percentage of inaccurate and/or incorrect operations, or 2) come with efficient checkers that can detect an unacceptably inaccurate result and enable the application to recompute the result if desired. Examples of such application domains include machine learning, multimedia, information retrieval, scientific, and financial analysis applications. Many of these applications have one or more approximate computational kernels that consume the majority of the execution time [16, 26].

In previous work, we developed Rely [7], a language for expressing and analyzing computations that run on approximate hardware platforms. These hardware platforms provide reliable and unreliable versions of standard arithmetic and logical instructions as well as reliable and unreliable memories. Rely enables a developer to manually identify unreliable instructions and variables that can be stored in unreliable memories. The developer also provides a *reliability specification*, which identifies the minimum required probability with which the kernel must produce a correct result. The Rely analysis then verifies that, with the identified unreliable instructions and variables, the kernel satisfies its reliability specification for all inputs.

Rely requires the developer to navigate the tradeoff between reliability and energy savings (because the developer is responsible for identifying the unreliable operations and data). But the developer must redo this identification every time the computation is ported to a new approximate hardware platform with different reliability and energy characteristics.

1.1 Chisel

We present Chisel, a new optimization framework that automatically selects approximate instructions and data that may be stored in approximate memory, given the exact kernel computation and the associated reliability and/or accuracy specification. Chisel automatically navigates the tradeoff space and generates an approximate computation that maximizes energy savings (according to an energy model for the hardware platform) while satisfying its combined reliability and accuracy specification. Chisel can therefore reduce the effort required to develop efficient approximate computations and enhance the portability of these computations.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2585-1/14/10.
http://dx.doi.org/10.1145/10.1145/2660193.2660231

1.2 Reliability and Accuracy Specifications

A Chisel program consists of code written in an implementation language (such as C) and kernel functions written in the Rely base language [7]. The kernel function can compute the return value, but may also write computed values into array parameters passed by reference into the kernel. Reliability specifications of the form `<r*R(x1, ..., xn)>` are integrated into the type signature of the kernel. Here `r` specifies the probability that the kernel (in spite of unreliable hardware operations) computes the value correctly. The term `R(x1, ..., xn)` is a *joint reliability factor* that specifies the probability that `x1,...,xn` all have correct values at the start of the kernel. In the following specification, for example:

```
int <.99 * R(x)> f(int[] <.98*R(x)> x);
```

the return value has reliability at least `.99` times the reliability of `x`; when `f` returns, the probability that all elements in the array `x` (passed by reference into `f`) have the correct value is at least `.98` times the reliability of `x` at the start of `f`.

Chisel also supports combined reliability and accuracy specifications of the following form (these specifications are *relational* in that they specify the combined accuracy and reliability with respect to the fully accurate exact computation):

```
<d, r*R(d1 >= D(x1), ..., dn >= D(xn))>
```

Here `d` is a maximum acceptable difference between the approximate and exact result values, `r` is the probability that the kernel computes a value within `d` of the exact value, and the term `R(d1 >= D(x1), ..., dn >= D(xn))` is a joint reliability factor that specifies the probability that each `xi` is within distance `di` of the exact value at the start of the computation. If `r=1`, then the specification is a pure accuracy specification; if `d=0` and all the `di=0`, then the specification is a pure reliability specification.

1.3 Reliability- and Accuracy-Aware Optimization

Chisel reduces the problem of selecting approximate instructions and variables allocated in approximate memories to an integer linear program whose solution minimizes an objective function that models the energy consumption of the kernel. The integer linear program also contains reliability and/or accuracy constraints that ensure that the solution satisfies the specification. For each instruction in the kernel, Chisel specifies a zero-one valued *configuration variable* that indicates whether the instruction should be exact (zero) or approximate (one). To generate the optimization objective, Chisel uses the execution traces of the kernel on representative inputs. To generate the constraints, Chisel statically analyzes the kernel.

A solution to the integer linear program provides a configuration of the kernel computation that minimizes energy consumption while satisfying the specification.

Chisel works with a hardware specification provided by the designers of the approximate hardware platform [21, 36]. This specification gives Chisel the hardware parameter values it needs to optimize the approximate computation.

1.4 Sensitivity Profiling

To help the developer obtain appropriate reliability and accuracy specifications, Chisel provides an optional sensitivity profiler. The profiler works with an end-to-end sensitivity metric, which compares the end-to-end results of the exact and approximate executions of the Chisel program to define the acceptability requirements for the outputs that the program produces. Specifically, the difference between the exact and approximate executions must be below a specified sensitivity bound. A sensitivity profiler (which performs function-level noise injection to estimate the sensitivity of the program's result to noise in the results that the kernel computes) can help the developer identify specifications that produce acceptable end-to-end results.

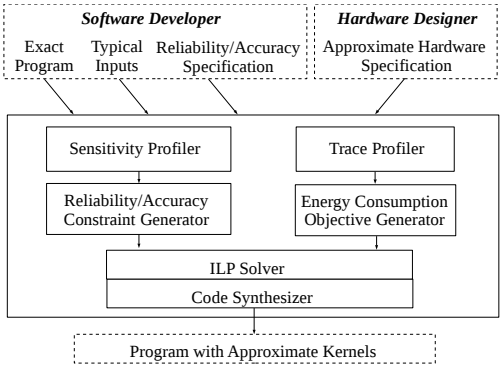

Figure 1: Chisel Overview

1.5 System Overview

Figure 1 presents an overview of the resulting Chisel system. The developer provides the Chisel program along with reliability and/or accuracy specifications for the approximate kernels (optionally obtaining these specifications via sensitivity profiling on representative inputs). The hardware designer provides a hardware specification, which specifies the reliability and accuracy information for individual instructions and approximate memory. Chisel analyzes the kernels, generates the integer linear program (we use the Gurobi solver [15]), then uses the solution to generate approximate code that satisfies its specification.

1.6 Contributions

The paper makes the following contributions:

- **Specification Language:** It presents the Chisel reliability and accuracy specification language. Chisel specifications are integrated into the type signatures of Chisel kernel functions and enable the developer to state the probabilistic reliability and/or accuracy constraints that approximate implementations of kernels must satisfy for the Chisel program to produce acceptable end-to-end results.
- **Optimization Algorithm.** It presents the Chisel optimization algorithm, which reduces the problem of selecting approximate instructions and data (which can be stored in approximate memories) to an integer linear program that minimizes energy consumption while satisfying the reliability and/or ac-

curacy specification. This algorithm automates the navigation of the reliability and accuracy versus energy consumption tradeoff space. It also eliminates the need to manually develop different approximate implementations for different hardware platforms.

- **Hardware Model.** It presents an abstract hardware model for programs that execute on approximate hardware and defines its semantics. The model defines an approximate hardware specification, which consists of reliability, accuracy, and energy consumption specifications for each hardware operation.
- **Extensions.** It presents a set of extensions to the basic Chisel optimization framework. These extensions enable Chisel to work with larger granularity operations, to model overhead associated with switching between exact and approximate modes of operation on approximate hardware platforms that support these two modes, to reason modularly about function calls within approximate kernels, and to optimize the placement of data accessed by multiple kernels in either exact or approximate memory.
- **Experimental Results.** It presents experimental results for five Chisel benchmark applications and a set of hardware platforms with unreliable operations. These results show that Chisel is able to produce approximate kernel implementations that are from 8.7% to 19.83% more energy efficient than the exact implementations. These kernel computations exploit a significant portion of the maximum possible energy savings offered by the unreliable hardware platforms (between 43% and 95% of the maximum savings), while preserving each kernel's reliability guarantee to produce acceptable end-to-end results.

2. Example

Figure 2 presents an implementation of an algorithm that scales an image to a larger size. It consists of the function `scale` and the function `scale_kernel`.

The function `scale` takes as input the scaling factor `f` (which increases the image size in both dimensions), along with integer arrays `src`, which contains the pixels of the image to be scaled, and `dest`, which contains the pixels of the resulting scaled image. 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 outer portion of the algorithm, which enumerates over the pixels in the destination image.

The function `scale_kernel` implements the core kernel of the scaling algorithm. The algorithm computes the location in the array `src` of the pixel's neighboring four pixels (Lines 4-5), adjusts the locations at the image edges (Lines 7-14), and fetches the pixels (Lines 16-19). To average the pixel values, the algorithm uses *bilinear interpolation*. Bilinear interpolation takes the weighted average of the four neighboring pixel values. The weights are computed as the distance from the source coordinates `i` and `j` to the location of each of the pixels (Lines 21-24). In the last step, the algorithm extracts each RGB color component of the pixel, computes the weighted average, and then returns the result (Lines 26-33).

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

Figure 2: Rely Code for Image Scaling Kernel

2.1 Sensitivity Profiling

Chisel's sensitivity profiler assists the developer in deriving the reliability specification of the kernel. The sensitivity profiler takes these three inputs from the developer:

- **Sensitivity Metric.** A function that compares the outputs of the original and approximate executions. It produces a numerical value that characterizes the difference between the two outputs. For computations that produce images, such as `scale`, a typically used metric is Peak-Signal-to-Noise Ratio (PSNR).
- **Sensitivity Goal.** The value of the sensitivity metric that characterizes the acceptable output quality of the approximate program.

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

Figure 3: Sensitivity Profiling for Image Scaling for Different Values of `r`

- **Sensitivity Testing Procedure.** A developer can write fault injection wrappers that inject noise into the computation. In general, the developer may use these wrappers to explore the sensitivity of the program's results to various coarse-grained error models. For `scale_kernel`, a developer can implement the following simple sensitivity testing procedure, which returns a random value for each color component:

```
int scale_kernel_with_errors(float i, float j, int[] src,
                             int s_height, int s_width) {
        return COMBINE(rand()%256, rand()%256, rand()%256);
}
```

Chisel's sensitivity profiler automatically explores the relation between the probability of approximate execution and the quality of the resulting image for the set of representative images. Conceptually, the profiler transforms the program to execute the correct implementation of `scale_kernel` with probability `r`, which represents the target reliability. The framework executes the faulty implementation `scale_kernel_with_errors` with probability `1-r`. The framework uses binary search to find the probability `r` that causes the noisy program execution to produce results with acceptable PSNR. The profiler can also plot the quality of the result as a function of `r`.

Figure 3 presents a visual depiction of the results of scaling for different values of `r`. Note that implementations with low reliability (0.20-0.80) do not produce acceptable results. However, as `r` reaches values in the range of 0.99 and above, the results become an acceptable approximation of the result of the original (exact) implementation. For the remainder of this section, we use 0.995 as `scale_kernel`'s target reliability, which yields images with an average PSNR of 30.9 dB.

2.2 Reliability Specification

The derived reliability specification for the kernel is:

```
int<0.995 * R(i, j, src, s_height, s_width)> scale_kernel
(float i, float j, int[] src, int s_height, int s_width);
```

The reliability specification of `scale_kernel` appears as part of the type signature of the function. The additional reliability information `0.995 * R(i, j, src, s_height, s_width)` specifies the reliability of the return value:

- **Input Dependencies.** The reliability of the return value is a function of the reliability of the function's inputs. The term `R(i, j, src, s_height, s_width)` represents 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 0.995 expresses the *reliability degradation* of the function. Specifically, the coefficient is the probability that the return value is correct given that all input variables have the correct values on entry to the function.

Since the specification does not explicitly state the acceptable absolute difference, it is by default `d=0`. Therefore, whenever the computation executes without errors, it should produce an exact result.

Arrays. The Rely base language contains annotations on the array parameters that specify that it is allocated in approximate memory. For instance, the following signature of `scale_kernel` would state that the pixel array `src` is in an approximate memory region named `urel`:

```
int<...> scale_kernel (..., int[] src in urel, ...);
```

To generate such annotations, Chisel explores the possibility that the array passed as a `src` parameter may be allocated in the approximate memory. Specifically, Chisel's optimization problem encodes both alternatives, i.e., when `src` is allocated in an exact memory and when it is allocated in an approximate memory. Chisel will report to the developer whether this alternative allocation strategy (which may save additional energy) still satisfies the reliability specification. The developer can then annotate the array's allocation statement to indicate that the compiler or the runtime system should allocate the array in an approximate memory.

Lower Bound on Sensitivity Metric. Starting with a reliability specification for our example kernel, it is also possible to obtain an analytic *lower bound* for the sensitivity metric. Specifically, the PSNR for the exact resulting image d and the approximate image d' is

$$\text{PSNR}(d,d') = 20\cdot\log_{10}(255) - 10\cdot\log_{10}\left(\frac{1}{3hw}\sum_{i=1}^{h}\sum_{j=1}^{w}\sum_{c\in\{R,G,B\}}(d_{ijc}-d'_{ijc})^2\right).$$

The constants h and w are the height and width of the image and R, G, and B are the color components of a pixel. Each color component is a value between 0 and 255.

The kernel computation computes the value of d'_{ijc} for all three RGB components correctly with probability r. In this case, $\sum_{c\in\{R,G,B\}}(d_{ijc}-d'_{ijc})^2=0$. With probability $1-r$, the kernel computation can compute the value of d'_{ijc} incorrectly. The upper bound on the expected error is then $\sum_{c\in\{R,G,B\}}(d_{ijc}-d'_{ijc})^2 \leq 3\cdot 255^2$. Therefore, the lower bound on the expected PSNR metric is

$$\text{PSNR}(d,d') \geq -10\cdot\log_{10}(1-r).$$

For a reliability specification $r = 0.995$, we can obtain that the expected PSNR is greater than 23.01 dB for any image (and for the typical images used in profiling it is greater than 30.9 dB).

2.3 Hardware Specification

To automatically optimize the implementation of the computation, the optimization algorithm requires a *hardware specification* of the approximate hardware, consisting of:

Operation and Memory Reliability. The hardware specification identifies 1) approximate arithmetic operations and 2) the approximate regions of the main and cache memories. The specification contains the reliability and (optionally) the accuracy loss of each arithmetic operation. It also contains the probability that read and write operations to approximate main memory and cache complete successfully.

Energy Model Parameters. To compute the savings associated with selecting approximate arithmetic operation, the energy model specifies the expected energy savings of executing an approximate version (as a percentage of the energy of the exact version). To compute the savings associated with allocating data in approximate memory, the energy model specifies the expected energy savings for memory cells.

To compute system energy savings, the energy model also provides 1) a specification of the relative portion of the system energy consumed by the CPU versus memory, 2) the relative portion of the CPU energy consumed by the ALU, cache, and other on-chip resources, and 3) the ratio of the average energy consumption of floating-point instructions and other non-arithmetic instructions relative to integer instructions.

2.4 Optimization Results

Chisel generates expressions that characterize the energy savings and reliability of `scale_kernel`. These expressions are parameterized by an unknown *configuration* of the approximate kernel, which specifies which operations and array parameters may be approximate or must be exact. This configuration is the solution to the optimization problem. For the hardware platforms in Section 9, the optimization algorithm delivers 19.35% energy savings, which is over 95% of the maximum possible energy savings for this computation (which occurs when the reliability bound is zero, and therefore all operations and the `src` and `dest` arrays can be approximate).

When the result of the function is assigned directly to an array variable, like in the case of the `dest` array, the optimization treats this variable (unless specified otherwise by the developer) as another array parameter of the kernel function that can be specified as approximate. Chisel identifies both `src` and `dest` arrays as potentially approximate. Chisel also identifies around 20% of the arithmetic operations as approximate. These operations are in the part of the computation that performs bilinear interpolation. For instance, the assignment to the variable `lr_w` on line 24 uses the inexact multiplication operation "`*`.".

Identifying the kernel's array parameters as approximate informs the developer that the kernel can satisfy its reliability specification with the array allocated in approximate memory.

$$r \in R \cup \{\mathrm{pc}\} \qquad n \in \mathrm{Int_N}$$
$$a \in A \subseteq \mathrm{Int_N} \qquad \kappa \in K = \{0,1\}$$
$$op \in Op ::= \texttt{add} \mid \texttt{fadd} \mid \texttt{mul} \mid \texttt{fmul} \mid \texttt{cmp} \mid \texttt{fcmp} \mid \ldots$$
$$i \in I ::= r = op^{\kappa}\ r\ r \mid \texttt{jmp}\ r\ r \mid$$
$$r = \texttt{load}\ r \mid \texttt{store}\ r\ r$$

Figure 4: Assembly Language Syntax

Given this information, the developer can use a predefined API call at the array allocation site to allocate the array in approximate memory across the entire application.

Final Sensitivity Validation. Using the specified sensitivity bound and metric, the framework can evaluate the generated approximate kernel computation on a set of (previously unseen) production inputs. For our example benchmark, the average PSNR on a set of production inputs is 32.31 dB.

3. Hardware Specification and Semantics

The code of `scale` in Section 2 illustrates the syntax of the Rely base language, which is a pointer-less C-like language with first-class one-dimensional arrays and reliability specifications. In this section, we present a hardware model and a compilation model for Chisel that captures the basic properties of approximate hardware.

3.1 Hardware Specification

We consider a single-CPU architecture that exposes an ISA with approximation extensions and an approximate memory hierarchy.

3.1.1 Syntax

Figure 4 presents the abbreviated syntax of the assembly language of the architecture.

Operands. Each operand is either a register $r \in R$ or a fixed N-bit (e.g., 32-bit or 64-bit) integer $n \in \mathrm{Int_N}$. Floating point numbers are integers coded with the IEEE 754 representation.

Instructions. Each instruction $i \in I$ is either an ALU/FPU arithmetic operation (such as add, multiply and compare), a conditional branch to an address (`jmp`), or a load/store from memory (`load` and `store`).

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

3.1.2 Hardware Specification

The reliability portion of the hardware specification $\psi \in (Op \rightarrow \mathbb{R}) \times (\mathbb{R} \times \mathbb{R}) \times (\mathbb{R} \times \mathbb{R})$ is a triple of structures that specify the reliability of instructions, the approximate memory region, and the approximate cache region, respectively. In Sections 6.1 and 7.1, we extend the hardware specification to include the hardware's accuracy and energy parameters, respectively.

ALU/FPU-C

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

ALU/FPU-F

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

Figure 5: Semantics Of Arithmetic Operations

Instructions. The projection π_{op} selects the first element of the hardware specification, which is a finite map from operations to reliabilities. The reliability of an operation is the probability that the operation executes correctly.

Memories. The hardware exposes an exact main memory region and an approximate memory region. The projection π_{mem} selects the second element of the hardware specification, which is a pair of reliabilities (r_{ld}, r_{st}) that denote the reliability of loading and storing a value in the approximate memory region, respectively.

Caches. The hardware exposes an exact cache and an approximate cache. The projection $\pi_{\$}$ selects the third element of the hardware specification, which is a pair of reliabilities (r_{ld}, r_{st}) that denote the reliability of loading and storing a value in the approximate cache, respectively.

3.2 Hardware Semantics

Register Files, Memories, Memory Configurations, 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* $m \in M = A \to \text{Int}_{\text{N}}$ is a finite map from addresses to machine integers. A memory configuration $\xi \in \Xi = A \to K$, maps an address a to a kind κ that designates whether the memory at a is configured as exact ($\kappa = 0$) or approximate ($\kappa = 1$). A *program* $\gamma \in \Gamma = A \to I$ is a finite map from addresses to instructions. An *environment* $\varepsilon \in \text{E} = \Sigma \times M$ is a register file and memory pair.

Ready Instructions and Configurations. A *ready instruction* $\hat{i} \in \hat{I} := i \mid \cdot$ is either an instruction $i \in I$ or the distinguished element "$\cdot$" that indicates that the next instruction needs to be fetched from memory (as determined by the pc register). A *configuration* $\langle \hat{i}, \varepsilon \rangle$ is a ready instruction and environment pair.

Errant Result Distributions. The discrete probability distribution $P_f(n_f \,|\, op, n_1, n_2)$ models the manifestation of an 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 and n_2 produces a value n_f different from the correct result of the operation.

Arithmetic Instructions. Figure 5 presents the inference rules for arithmetic operations. We present the remaining rules in Section A of the Appendix [22]. The small-step judgment $\langle \hat{i}, \varepsilon \rangle \xrightarrow[\gamma,\psi,\xi]{\lambda,p} \langle \hat{i}', \varepsilon' \rangle$ denotes that execution of the program γ from the configuration $\langle \hat{i}, \varepsilon \rangle$ under a hardware model ψ and a memory configuration ξ takes a transition with label λ with probability p, yielding 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 in a faulty transition records the value that the fault inserted into the semantics of the program. The semantics of an arithmetic operation $r = op^{\kappa}\ r_1\ r_2$ takes one of two possibilities:

- **Correct execution [ALU/FPU-C].** An operation executes correctly with probability $\pi_{\text{op}}(\psi)(op)^{\kappa}$. Therefore, if the operation is exact ($\kappa = 0$) it executes correctly with probability 1. If it is approximate ($\kappa = 1$), 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 registers r_1 and r_2 from the register file, performs the operation, and then stores the result back in register r.
- **Faulty execution [ALU/FPU-F].** An operation with a kind $\kappa = 1$ experiences a fault with probability $1 - \pi_{\text{op}}(\psi)(op)$. 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. Note that while the instruction may experience a fault, its faulty execution does not modify any state besides the destination register.

Control Flow. Control flow transfer instructions, such as `jmp`, always correctly transfer control to the destination address. Preserving the reliability of control flow transfers guarantees that an approximate program always takes paths that exist in the static control flow graph of the program. We note that while control flow transfers themselves execute correctly, the argument to a control transfer instruction (e.g., the test condition of a `jmp`) may depend on approximate computation. Therefore, an approximate program may take a path that differs from that of the original (exact) program.

Loads and Stores. The semantics of loads and stores are similar to arithmetic operation semantics in that each operation can either execute correctly or encounter a fault. The memory configuration ξ determines if an accessed address's memory region is exact (all operations on the region execute correctly) or approximate (operations may encounter a fault). As with the destination register of arithmetic operations, if a store instruction encounters a fault, then only the contents of the destination address are modified.

Data stored in the approximate memory region may be placed in the approximate cache. We conservatively model the cache as a buffer that affects the probability of correctly executing load and store operations. We discuss the cache's semantics in Section A.2 of the Appendix [22].

3.3 Compilation and Runtime Model

Data Layout. The compilation and runtime system stores the program's instructions and the stack in the exact memory region. The system represents arrays with a header and a separately allocated chunk of memory that contains the array's data. The header contains the length of the array's data and the address of the array's data in memory. The system allocates the header in exact main memory and allocates the data chunk in either

exact or approximate memory based upon Chisel's optimization results. This allocation strategy enables the system to separate the reliability of the array's metadata from the reliability of the data stored in the array.

To support our formalization of reliability, we define a *variable allocation* $\upsilon \in V \to \mathcal{P}(A)$ as a finite map from a *program variable* $v \in V$ to the address (or set of addresses in the case of an array) in memory at which the variable has been allocated by the compilation and runtime system.

Array Loads/Stores. The compilation system uses the bounds information of each array to provide a failure oblivious [34] semantics for array loads and stores. Specifically, the compiled program includes a bounds check for each access. If an index for a load is out of bounds, then the check returns an arbitrary value for the load. If the access is a store, then the check elides the write to the array. This semantics enables Chisel-optimized programs to continue executing even if an array access is out-of-bounds due to approximation.

4. Preliminary Definitions

We next present several definitions that enable us to precisely specify the configuration of approximate programs along with their reliability, accuracy, and energy consumption.

4.1 Configurable Approximate Programs

We augment our program representation to create an intermediate representation that includes *labels*, where each label $\ell \in \mathcal{L}$ is uniquely associated with an instruction or a program variable. Labels enable Chisel to separately mark each instruction and variable as either exact or approximate.

Instructions. We augment each arithmetic instruction to have a label instead of a kind:

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

Program Variables. We define the finite map $\chi \in V \to \mathcal{L}$ that maps each variable in the program to a unique label.

Kind Configurations. We also define a *kind configuration* $\theta \in \Theta = \mathcal{L} \to K$ as a finite map from labels to kinds that denotes a selection of the kind (i.e., exact or precise) of each of the program's instructions and variables. The set of kind configurations denotes that set of all possible optimized programs that Chisel can generate. We also define the substitution $\gamma[\theta]$ as the program generated by substituting each label ℓ in the program γ by the corresponding kind given by θ (namely, $\theta(\ell)$).

4.2 Big-step Semantics

We use the following big-step semantics to later define an approximate program's reliability and accuracy.

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

$$\langle \cdot,\varepsilon\rangle \underset{\gamma,\psi,\xi}{\overset{\tau,p}{\Longrightarrow}} \varepsilon' \equiv \langle \cdot,\varepsilon\rangle \underset{\gamma,\psi,\xi}{\overset{\lambda_1,p_1}{\longrightarrow}} \ldots \underset{\gamma,\psi,\xi}{\overset{\lambda_n,p_n}{\longrightarrow}} \langle \cdot,\varepsilon'\rangle$$
$$\textit{where } \tau = \lambda_1,\ldots,\lambda_n,\ p = \prod_{i=1}^{n} p_i \textit{ and final}(\langle \cdot,\varepsilon'\rangle,\gamma)$$

The big-step trace semantics is a reflexive transitive closure of the small-step execution relation that records a trace of the program's execution. A *trace* $\tau \in \mathrm{T} ::= \cdot \mid \lambda :: \mathrm{T}$ is a sequence of small-step transition labels. The *probability of a trace*, p, is the product of the probabilities of each transition. The predicate *final* $\subseteq (\hat{I} \times \mathrm{E}) \times \Gamma$ indicates that the program cannot make a transition from the configuration.

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

$$\langle \cdot,\varepsilon\rangle \underset{\gamma,\psi,\xi}{\overset{p}{\Longrightarrow}} \varepsilon' \textit{ where } p = \sum_{\tau \in \mathrm{T}} p_\tau \textit{ such that } \langle \cdot,\varepsilon\rangle \underset{\gamma,\psi,\xi}{\overset{\tau,p_\tau}{\Longrightarrow}} \varepsilon'$$

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 specification ψ and memory configuration ξ.

5. Reliability Constraint Construction

Chisel generates optimization constraints via a *precondition* generator. Similar to Rely, Chisel's generator produces a precondition that if valid before the execution of the kernel, guarantees that the kernel produces acceptable outputs with reliability at least that given in its specification. Chisel then transforms the generated precondition into an optimization constraint.

5.1 Reliability Predicates

Chisel's generated preconditions are *reliability predicates* that characterize the reliability of an approximate program. A reliability predicate P has the following form:

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

Specifically, a predicate is either a conjunction of predicates or a comparison between *reliability factors*, R_f. A reliability factor is either a real number ρ, a *kinded reliability* ρ^{ℓ}, a *joint reliability factor* $\mathcal{R}(O)$ of a set of register and variable operands $O \subseteq R \cup V$, or a product of reliability factors.

The denotation of a predicate $[\![P]\!] \in \mathcal{P}(\mathrm{E} \times \Phi \times \Theta \times \Upsilon)$ is the set of environment, approximate *environment distribution*, kind configuration, and variable allocation quadruples that satisfy the predicate. An environment distribution $\varphi \in \Phi = \mathrm{E} \to \mathbb{R}$ is a probability distribution over possible approximate environments.

The denotation of a reliability factor $[\![R_f]\!] \in \mathrm{E} \times \Phi \times \Theta \times \Upsilon \to \mathbb{R}$ is the real-valued reliability that results from evaluating the factor for a given quadruple. For example, $[\![\rho]\!](\varepsilon,\varphi,\theta,\upsilon) = \rho$ and $[\![\rho^{\ell}]\!](\varepsilon,\varphi,\theta,\upsilon) = [\![\rho]\!](\varepsilon,\varphi,\theta,\upsilon)^{\theta(\ell)}$. The denotation of $\mathcal{R}(O)$ is the probability that an approximate environment ε_a sampled from φ has the same value for all operands in O as the environment ε:

$$[\![\mathcal{R}(O)]\!](\varepsilon,\varphi,\theta,\upsilon) = \sum_{\varepsilon_a \in \mathcal{E}(\varepsilon,O,\upsilon)} \varphi(\varepsilon_a), \tag{1}$$
$$\text{where } \mathcal{E}((\sigma,m),O,\upsilon) =$$
$$\{(\sigma_a,m_a) \mid \forall o.\ o \in R \Rightarrow \sigma_a(o) = \sigma(o) \wedge$$
$$o \in V \Rightarrow \forall a \in \upsilon(o).\ m_a(a) = m(a)\}.$$

The function $\mathcal{E}(\varepsilon,O,\upsilon)$ is the set of environments in which the values of all operands O are the same as in ε.

5.2 Semantics of Reliability

Given the semantics of reliability predicates, we define the reliability of an approximate program with a Hoare-triple-like semantics defined for the program's *paired execution semantics*:

Definition 3 (Paired Execution Semantics)**.**

$$\langle\cdot,\langle\varepsilon,\varphi\rangle\rangle \Downarrow^{\theta,\chi,\upsilon}_{\gamma,\psi} \langle\varepsilon',\varphi'\rangle \textit{ such that } \langle\cdot,\varepsilon\rangle \underset{\gamma[0_\theta],\psi,0_\xi}{\overset{\tau,p_\tau}{\Longrightarrow}} \varepsilon' \textit{ and}$$
$$\varphi'(\varepsilon'_a) = \sum_{\varepsilon_a\in \mathrm{E}} \varphi(\varepsilon_a)\cdot p_a \textit{ where } \langle\cdot,\varepsilon_a\rangle \underset{\gamma[\theta],\psi,\xi}{\overset{p_a}{\Longrightarrow}} \varepsilon'_a \textit{ and}$$
$$\forall v\in V.\ \forall a\in \upsilon(v).\ \xi(a) = \theta(\chi(v))$$

The paired execution semantics pairs a program's exact execution with its approximate executions.

Exact Execution. The semantics specifies the program's exact execution via a big-step execution that uses the *exact kind and memory configurations* 0_θ and 0_ξ that both return 0 for all inputs.

Approximate Execution. Because approximate operations can produce different results with some probability, the natural representation for the environments of a program's approximate execution is a probability distribution that specifies the probability that the execution is in a particular environment. The semantics specifies the distributions of the approximate execution's initial and final environments with the distributions φ and φ', respectively. The relationship between these two distributions is given by a summation over big-step executions, each of which use the potentially approximate kind and memory configurations θ and ξ. For each program variable v, ξ maps the variable's addresses ($\upsilon(v)$) to either the exact or approximate memory according to the kind specified for the variable ($\theta(\chi(v))$).

Reliability Transformer. Reliability predicates and the semantics of approximate programs are connected through the view of a program as a *reliability transformer*. Namely, similar to the standard Hoare triple relation, if an environment and distribution pair $\langle\varepsilon,\varphi\rangle$ satisfy a reliability predicate P, then the program's paired execution transforms the pair to a new pair $\langle\varepsilon',\varphi'\rangle$ that satisfy a predicate Q. We formalize the reliability transformer relation in Section B of the Appendix [22].

5.3 Reliability Precondition Generator

Given a predicate targeted to be true after the execution of a program (a *postcondition*), Chisel's reliability precondition generator produces a precondition that when true before the execution of the program, ensures that the postcondition is true after. Namely, the precondition, program, and postcondition satisfy the reliability transformer relation.

The precondition generator operates backwards on the program's instruction sequence, starting at the program's last instruction and ending at the program's first. The generator starts with an initial postcondition that is a conjunction of terms of the form $\rho_{spec,i}\cdot\mathcal{R}(V_{spec,i}) \le \mathcal{R}(\{v_i\})$ and the term $\rho_{spec,ret}\cdot\mathcal{R}(V_{spec,ret}) \le \mathcal{R}(\{r_{ret}\})$. The left-hand side of the inequalities represent the reliability specification of the array parameters and the return value, respectively. Each v_i is an array parameter and r_{ret} is the register that contains the return value. The postcondition asserts that the reliability of each output array and the function's return value must be at least that given in their specification.

5.3.1 Reliability Precondition Analysis

The reliability precondition generator is a function $C \in I \times P \to P$ that takes as inputs an instruction and a postcondition and produces a precondition as output. The analysis rules for arithmetic instructions and memory accesses are as follows:

$$C(r = op^\ell\ r_1\ r_2, Q) = Q[\rho^\ell_{op}\cdot\mathcal{R}(\{r_1,r_2\}\cup X))/\mathcal{R}(\{r\}\cup X)]$$
$$C(r_1 = \texttt{load}\ r_2, Q) = Q[\rho^{\chi(\eta(r_2))}_{ld}\cdot\mathcal{R}(\{\eta(r_2)\}\cup X)/\mathcal{R}(\{r_1\}\cup X)]$$
$$C(\texttt{store}\ r_1\ r_2, Q) = Q[\rho^{\chi(\eta(r_1))}_{st}\cdot\mathcal{R}(\{r_2\}\cup X)/\mathcal{R}(\{\eta(r_1)\}\cup X)]$$

ALU/FPU. The first equation 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 $Q[\mathcal{R}(\{r_1,r_2\}\cup X)/\mathcal{R}(\{r\}\cup X)]$ matches all occurrences of the destination register r in a reliability term that occur in the predicate Q and replaces them with the input registers, r_1 and r_2. The substitution also multiplies in the factor ρ^ℓ_{op}, which expresses the reliability of the operation *op* as a function of its label's kind configuration, and its reliability $\rho_{op} = \pi_{\mathrm{op}}(\psi)(op)$.

Load/Store. The second and the third equations present the rules for loads and stores from potentially approximate memory. The rules use the auxiliary *register mapping* generated by the compiler ($\eta \in R \to V$) that maps the address operand register to the program variable that is read or written.

The minimum reliability of a load from a potentially approximate variable, ρ_{ld}, is equal to the probability that the read from memory, the write to a cache location, and the read from that cache location all execute correctly, $\pi_1(\pi_{\mathrm{mem}}(\psi))\cdot\pi_1(\pi_{\$}(\psi))\cdot\pi_2(\pi_{\$}(\psi))$. The reliability of a store to a potentially approximate variable, ρ_{st}, assuming a write-through cache, is equal to the reliability of a memory store, $\pi_2(\pi_{\mathrm{mem}}(\psi))$.

This rule presents the semantics of strong updates for scalar program variables. In Section B of the Appendix [22] we present the rules for weak updates of array variables.

Control Flow. We elide the rules for control flow. Our analysis relies on the fact that the Rely base language has structured control flow and therefore it is straightforward to map assembly instructions to high-level program structures, such as `if` statements and `while` loops. Working with these structures, our analysis is similar to Rely's analysis. Specifically, for `if` statements the resulting precondition ensures that both branches of the `if` satisfy the postcondition (inclusive of the probability that the condition executes correctly). The analysis of bounded `while` loops is conceptually similar to loop unrolling whereas the analysis of unbounded loops sets the reliability of variables that are unreliably updated within the body of the loop to zero.

5.3.2 Final Precondition

For a given kernel, our analysis computes a precondition that is a conjunction of the terms of the form

$$\rho_{spec} \cdot \mathcal{R}(V_{spec}) \leq r \cdot \mathcal{R}(V),$$

where $\rho_{spec} \cdot \mathcal{R}(V_{spec})$ is a reliability factor for a developer-provided specification of an output and $r \cdot \mathcal{R}(V)$ is a lower bound on the output's reliability computed by the analysis.

Each ρ_{spec} is a real-valued constant and each r is a product of a real-valued constant and kinded reliabilities of the form

$$\rho \cdot \Pi_k \rho_k^{\ell_k}.$$

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

5.4 Optimization Constraint Construction

Validity Checking. To check the validity of this precondition, we use the observation that the reliability of any subset of a set of variables is greater than or equal to the reliability of the set as a whole [7, Proposition 1]. Specifically,

$$V \subseteq V_{spec} \Rightarrow \mathcal{R}(V_{spec}) \leq \mathcal{R}(V). \quad (2)$$

Therefore, Chisel can soundly ensure the validity of each inequality in the precondition by verifying that $\rho_{spec} \leq r$ and $V \subseteq V_{spec}$.

Constraint Construction. Given a precondition, Chisel next generates an optimization constraint. For each inequality in the precondition, Chisel immediately checks if $V \subseteq V_{spec}$. For the test $\rho_{spec} \leq r$, recall that the reliability expression r has the form $\rho \cdot \Pi_k \rho_k^{\ell_k}$. Given that the denotation of ρ^ℓ under a configuration θ is $\rho^{\theta(\ell)}$, Chisel produces a final optimization constraint by taking the logarithm of both sides of the inequality:

$$\log(\rho_{spec}) - \log(\rho) \leq \sum_k \theta(\ell_k) \cdot \log(\rho_k). \quad (3)$$

We note that the expression on the right side is linear with respect to all labels ℓ_k. Each label's kind is an integer variable that can take a value 0 or 1. The reliabilities ρ are constants and their logarithms are immediately computable.

6. Accuracy Constraint Construction

To exploit the capabilities of architectures that have variable precision floating point units [12, 17, 44, 45], we now present Chisel's analysis that unifies reasoning about both reliability and accuracy. Specifically, we extend reliability predicates with the ability to characterize the difference in the value of a variable between the kernel's exact and approximate executions. Then, our constraint generator produces linear expressions of kind configurations that characterize how the numerical error emerges and propagates through the kernel.

6.1 Accuracy Specification

Approximate Hardware Specification. For each approximate floating point operation *op*, we extend the definition of the hardware specification ψ from Section 3.1 to also include the accuracy specification of the variable-accuracy instructions. The specification of a variable-accuracy instruction consists of the reliability r and the number of mantissa bits that are computed fully accurately c (which determines the maximum error of the operation). Each operation produces an approximate result (with error whose magnitude bound is determined by c) with probability r. With probability $1-r$, the operation can produce an arbitrarily inaccurate result.

Function Specification. We extend the syntax of reliability specifications from the Rely base language to include a specification of acceptable accuracy loss. The extended specification has the following form:

```
<d, r * R( d1 >= D(x1), ..., dn >= D(xn) )>
```

The constant `d` specifies the maximum acceptable difference between the results of the exact and approximate executions. The constant `r` specifies the probability with which the approximate execution will produce a result within distance `d` of the exact result. The constraints `di >= D(xi)` specify that the non-negative value `di` is the maximum absolute difference between the values of the function's parameter `xi` at the beginning of the exact and approximate kernel executions.

Interval Specification. We extend function specifications to enable developers to specify the intervals of values of a function's parameters. Because the accuracy analysis relies on an internal interval analysis, the precision of the results of this analysis depends on the precision of the intervals specified for the inputs to the function. To specify the parameter interval, a developer precedes a function's declaration with an annotation of the form `@interval(p,a,b)`, denoting that the value of the parameter `p` is within the interval $[a,b]$.

6.2 Accuracy Predicates and Reliability Predicates

We next present the syntax and semantics of the predicates generated by the accuracy analysis:

$$\begin{array}{lcl} Q_A & := & R_D \geq R_A \mid Q_A \wedge Q_A \\ R_A & := & R_D \mid R_D \cdot \ell \mid R_D \cdot \Delta(v) \mid R_A + R_A \\ R_D & := & d \mid \infty \mid R_D \cdot R_D \end{array}$$

An accuracy predicate Q_A is a conjunction of accuracy predicates or a comparison between a product of non-negative real constants extended by infinity R_D and an accuracy expression. An accuracy expression R_A has one of four forms: a constant term R_D; a product of a constant term and a label ℓ; a product of a constant term and a *distance operator* $\Delta(v)$ that relates the values of the variable v in an exact and an approximate execution; or an addition of two accuracy expressions.

Figure 6 presents the denotational semantics of accuracy expressions and predicates. Accuracy expressions and predicates have a similar semantics to that of standard logical predicates over numerical expressions. The main point of departure is the semantics of the distance operator, which is the absolute difference between the value of a variable in an exact environment ε and its corresponding value in an approximate environment ε_a. For notational purposes, we define implication as:

$$Q_{A1} \Rightarrow Q_{A2} \equiv [\![Q_{A1}]\!] \subseteq [\![Q_{A2}]\!].$$

$$[\![R_D]\!] \in R^+ \cup \{\infty\} \qquad [\![d]\!] = d \qquad [\![\infty]\!] = \infty \qquad [\![0 \cdot \infty]\!] = 0 \qquad [\![R_D \cdot \infty]\!] = \infty \qquad [\![R_{D1} \cdot R_{D2}]\!] = [\![R_{D1}]\!] \cdot [\![R_{D2}]\!]$$

$$[\![R_A]\!] \in \mathrm{E} \times \mathrm{E} \times \Theta \times \Upsilon \to R^+ \cup \{\infty\} \qquad [\![R_D \cdot \ell]\!](\varepsilon, \varepsilon_a, \theta, \upsilon) = [\![R_D]\!] \cdot \theta(\ell) \qquad [\![\Delta(v)]\!](\varepsilon, \varepsilon_a, \theta, \upsilon) = \max_{a \in v(v)} |\pi_2(\varepsilon_a)(a) - \pi_2(\varepsilon)(a)|$$

$$[\![R_D \cdot \Delta(v)]\!](\varepsilon, \varepsilon_a, \theta, \upsilon) = [\![R_D]\!] \cdot [\![\Delta(v)]\!](\varepsilon, \varepsilon_a, \theta, \upsilon) \qquad [\![R_{A1} + R_{A2}]\!](\varepsilon, \varepsilon_a, \theta, \upsilon) = [\![R_{A1}]\!](\varepsilon, \varepsilon_a, \theta, \upsilon) + [\![R_{A2}]\!](\varepsilon, \varepsilon_a, \theta, \upsilon)$$

$$[\![Q_A]\!] \in \mathcal{P}(\mathrm{E} \times \mathrm{E} \times \Theta \times \Upsilon) \qquad [\![R_D \geq R_A]\!] = \{(\varepsilon, \varepsilon_a, \theta, \upsilon) \mid [\![R_D]\!] \geq [\![R_A]\!](\varepsilon, \varepsilon_a, \theta, \upsilon)\} \qquad [\![Q_{A1} \wedge Q_{A2}]\!] = [\![Q_{A1}]\!] \cap [\![Q_{A2}]\!]$$

Figure 6: Accuracy Predicate Semantics

$$AE \in Exp \to R_A \qquad AE(n) = 0 \qquad AE(x) = \Delta(x)$$

$$AE(e_1\ op^{\ell}\ e_2) = \pi_1(\mathsf{propagation}_{op,\mathcal{I}}(e_1,e_2)) \cdot AE(e_1) + \pi_2(\mathsf{propagation}_{op,\mathcal{I}}(e_1,e_2)) \cdot AE(e_2) + \ell \cdot \mathsf{maxerr}_{op,\psi,\mathcal{I}}(e_1,e_2)$$

$$\begin{array}{rcl}
C^*_{\psi,\mathcal{I}} & \in & S \times P' \to P' \\
C^*_{\psi,\mathcal{I}}(x \texttt{ = } e, Q_R) & = & \textbf{let } Q'_A = Q_A[AE(e)/\Delta(x)] \textbf{ in } \ Q_R\,[RE_{assign,\psi}(x,e) \cdot \mathcal{R}^*(Q'_A)/\mathcal{R}^*(Q_A)] \\
C^*_{\psi,\mathcal{I}}(\texttt{if } x_b\ s_1\ s_2, Q_R) & = & \textbf{let } Q'_A = Q_A \wedge 0 \geq \Delta(x_b) \textbf{ in} \\
 & & \quad C^*_{\psi,\mathcal{I}}(s_1, Q_R\,[\mathcal{R}^*(Q'_A)/\mathcal{R}^*(Q_A)]) \wedge C^*_{\psi,\mathcal{I}}(s_2, Q_R\,[\mathcal{R}^*(Q'_A)/\mathcal{R}^*(Q_A)]) \\
C^*_{\psi,\mathcal{I}}(x \texttt{ = } \phi(x_1,x_2), Q_R) & = & \textbf{let } Q'_A = Q_A[\Delta(x_1)/\Delta(x)] \textbf{ and } Q''_A = Q_A[\Delta(x_2)/\Delta(x)] \textbf{ in} \\
 & & \quad Q_R\,[\mathcal{R}^*(Q'_A)/\mathcal{R}^*(Q_A)] \wedge Q_R\,[\mathcal{R}^*(Q''_A)/\mathcal{R}^*(Q_A)]
\end{array}$$

Figure 7: Accuracy (Q_A) and Extended Reliability (Q_R) Precondition Construction

Extended Reliability Predicates. To specify Chisel's extended reliability precondition generator, we extend the reliability predicate definition from Section 5.1 by adding a *generalized joint reliability factor*, $\mathcal{R}^*(Q_A)$. $\mathcal{R}^*(Q_A)$ denotes the probability that the exact environment ε and an approximate environment ε_a sampled from φ together satisfy the accuracy predicate Q_A:

$$[\![\mathcal{R}^*(Q_A)]\!](\varepsilon, \varphi, \theta, \upsilon) = \sum_{\varepsilon_a \in \mathcal{E}(\varepsilon,\theta,\upsilon,Q_A)} \varphi(\varepsilon_a),$$

where $\mathcal{E}(\varepsilon,\theta,\upsilon,Q_A) = \{\varepsilon_a \mid (\varepsilon,\varepsilon_a,\theta,\upsilon) \in [\![Q_A]\!]\}$. The syntax of the extended reliability predicates is $R'_f := R_f \mid \mathcal{R}^*(Q_A)$ and $P' := R'_f \leq R'_f \mid P' \wedge P'$.

This definition of joint reliability factors subsumes the definition of the standard joint reliability factors $\mathcal{R}(V)$ (Section 5.1, Equation 1). Specifically, the set of variables that have the same value in the exact and approximate program executions can be represented using accuracy predicates that bound the acceptable absolute difference of each variable by zero:

$$[\![\mathcal{R}(V)]\!](\varepsilon, \varphi, \theta, \upsilon) = [\![\mathcal{R}^*(\bigwedge_{v \in V} 0 \geq \Delta(v))]\!](\varepsilon, \varphi, \theta, \upsilon).$$

6.3 Extended Reliability Precondition Generator

We now present the precondition generator of extended reliability predicates. For simplicity, we present this analysis at the level of the constructs in the Rely base language. This analysis is applicable to kernel computations without unbounded loops.

Figure 7 presents the selection of rules of the combined accuracy and reliability analysis. The precondition generator takes as input a statement s and an extended reliability postcondition Q_R and generates a precondition Q'_R, such that if Q'_R holds before the paired execution of the statement s, then Q_R holds after the paired execution of s. The final precondition generated for a full program is a precondition from which the program satisfies both its reliability and accuracy specification.

6.3.1 Interval Analysis

To compute the absolute error induced by variable-accuracy arithmetic operations (given the number of accurately computed mantissa bits), an accuracy analysis requires the intervals of the operation's inputs. Therefore, we define an auxiliary interval analysis that computes the intervals of expressions computed within the kernel. These intervals include the maximum absolute errors induced by the variable-accuracy floating point operations.

The analysis produces a mapping $\mathcal{I} : \mathcal{L} \to (\text{Float} \times \text{Float}) + \textit{Unbounded}$, which yields the interval of values to which each expression (identified by its label $\ell \in \mathcal{L}$) evaluates. The set Float contains all floating point numbers that the target platform can represent. A special symbol *Unbounded* indicates that the interval is unbounded (due to e.g., a possible overflow or divide by zero).

The analysis operates in a forward fashion, using the standard rules of interval arithmetic. To provide a conservative estimate of the error that the approximate execution may produce, for every arithmetic operation $e_1\ op^{\ell}\ e_2$ the interval analysis extends the computed interval of the result of the exact operation $[a,b]$ with an error term δ, which represents the maximum absolute error of the approximate operation. The resulting interval is then $\mathcal{I}(\ell) = [a-\delta, b+\delta]$. The computation of conservative intervals is inspired by the analysis presented in [11].

To compute the error term for an arithmetic operation, the analysis uses the function $\mathsf{maxerr}_{op,\psi,\mathcal{I}}(e_1,e_2)$, which returns the maximum error when the operation *op* operates on only a fraction of the inputs' mantissa bits and the intervals of the operands are $\mathcal{I}(\mathsf{loc}(e_1))$ and $\mathcal{I}(\mathsf{loc}(e_2))$. The function loc returns

the label of an expression. If any operand interval is unbounded, then the result interval is also unbounded.

6.3.2 Analysis of Arithmetic Expressions

The function *AE* in Figure 7 produces an expression that bounds the absolute error of an arithmetic expression.

Error Propagation. The function $\mathsf{propagation}_{op,\mathcal{I}}(e_1,e_2)$ returns a pair of real-valued *error propagation coefficients* (k_1,k_2) that specify how sensitive the result of the operation is to the changes of the first and the second operand, respectively.

To compute the coefficients for each operation, we use the observation that for a differentiable function $f(x,y)$ defined on a bounded interval and inputs with errors $\hat{x}=x+\delta_x$ and $\hat{y}=y+\delta_y$, one can show that $|f(x,y)-f(\hat{x},\hat{y})| \leq k_1 \cdot |\delta_x| + k_2 \cdot |\delta_y|$. The constants $k_1 = \max_{x,y}\left|\frac{\partial f(x,y)}{\partial x}\right|$ and $k_2 = \max_{x,y}\left|\frac{\partial f(x,y)}{\partial y}\right|$ can be computed from the input intervals when the partial derivatives of f are bounded. Note that the input intervals include the bounds for the errors δ_x and δ_y.

We can use this observation to specify the error propagation functions for the four arithmetic operations:

$$\mathsf{propagation}_{+,\mathcal{I}}(e_1,e_2) = (1,1)$$
$$\mathsf{propagation}_{-,\mathcal{I}}(e_1,e_2) = (1,1)$$
$$\mathsf{propagation}_{*,\mathcal{I}}(e_1,e_2) = (\max_{y\in\mathcal{I}_2}|y|, \max_{x\in\mathcal{I}_1}|x|)$$
$$\mathsf{propagation}_{\div,\mathcal{I}}(e_1,e_2) = (\max_{y\in\mathcal{I}_2}|1/y|, \max_{x\in\mathcal{I}_1, y\in\mathcal{I}_2}|x/y^2|) \text{ when } 0\notin\mathcal{I}_2.$$

Recall that the conservative interval analysis incorporates the maximum error that can propagate from the operands. Therefore, the intervals of the operands $\mathcal{I}_1=\mathcal{I}(\mathsf{loc}(e_1))$ and $\mathcal{I}_2=\mathcal{I}(\mathsf{loc}(e_2))$ incorporate the upper bounds for the errors in the operands. If either interval is unbounded or the divisor's interval includes 0, then the corresponding coefficient will be infinity (∞), indicating that the operand's value is *critical*, i.e., the kernel's result is highly sensitive to its change.

Error Induced by Approximation. The analysis uses the function $\mathsf{maxerr}_{op,\psi,\mathcal{I}}(e_1,e_2)$ to compute the maximum error induced by the approximate arithmetic expression when the inputs are in $\mathcal{I}(\mathsf{loc}(e_1))$ and $\mathcal{I}(\mathsf{loc}(e_2))$. If either of the intervals is unbounded, then the function returns ∞.

The propagation and approximation-induced errors are additive because for two continuous functions f and $\hat{f}$, it follows from the triangle inequality that $|f(x,y) - \hat{f}(\hat{x},\hat{y})| \leq |f(x,y) - f(\hat{x},\hat{y})| + |f(\hat{x},\hat{y}) - \hat{f}(\hat{x},\hat{y})|$. Therefore, the total absolute error is bounded by the sum of the error that propagates through the operands, characterized by the propagation coefficients from $\mathsf{propagation}_{op,\mathcal{I}}(\cdot)$, and the induced error, $\mathsf{maxerr}_{op,\psi,\mathcal{I}}(\cdot)$. To control whether to approximate the operation, the generator multiplies the induced error with the operation's label.

6.3.3 Analysis of Statements

Figure 7 presents the selection of rules for the precondition generator for statements, C^*. We present the remaining rules in Section C of the Appendix [22]. The precondition generator for statements operates backwards, from the end to the beginning of the kernel function. It transforms the extended reliability predicate Q_R, starting from the predicate that is the conjunction of the terms $\rho_{spec,i}\cdot\mathcal{R}^*(Q_{spec,i}) \leq \mathcal{R}^*(d_{spec,i} \geq \Delta(v_i))$ for each kernel's array parameter v_i. The analysis rules for statements are analogous to those from the reliability analysis in Section 5. The main difference between the two analyses is in the propagation of the accuracy predicate Q_A within the reliability factors, as opposed to propagating sets of variables.

Kernel Preprocessing. Before precondition generation, a preprocessing pass flattens conditionals and transforms the kernel's code to an SSA form (as in [7]). In addition, the preprocessing pass also unrolls finitely bounded loops.

Assignment. The assignment operator modifies the accuracy predicate by substituting the occurrences of $\Delta(x)$, the distance operator for the variable x with the assignment's accuracy expressions $AE(e)$. The generator substitutes the joint reliability factor $\mathcal{R}^*(Q_A)$ with the product of the reliability expression $RE_{assign,\psi}(x,e)$, generated by the analysis from Section 5, and the joint reliability factor of the new accuracy predicate Q'_A.

Control Flow. For the conditional statement, both branches must satisfy the predicate. Note that the preprocessing pass flattens the conditional by extracting the variable x_b, which is assigned the expression that computes the boolean condition. The predicate $0 \geq \Delta(x_b)$ therefore states that the computation affecting the statement's condition cannot be computed with reduced accuracy (which could cause control flow divergence). This predicate simplifies the accuracy analysis so that it need not consider all possible combinations of divergent paths for the kernel's exact and approximate executions.

The rule for phi-nodes substitutes the distance operator with the variable's name for the distance operators of the alternative variable names in each of the branches to address the dependence of the variable's value on the control flow.

Array Operations. We present the analysis of array operations in Section C of the Appendix [22]. They are analogous to the rule for the assignment statement, but also ensure that the variable-accuracy computation does not affect the array index computation.

6.3.4 Final Precondition

The precondition generator generates a precondition that is a conjunction of terms of the form:

$$\rho_{spec}\cdot\mathcal{R}^*(Q_{spec}) \leq r\cdot\mathcal{R}^*(Q_A).$$

The accuracy predicate Q_{spec} (given by the specification) is a conjunction of terms of the form

$$d \geq \Delta(v), \tag{4}$$

where each d is a constant and each v is a function parameter.

The accuracy predicate Q_A (produced by the precondition generator) is a conjunction of terms of the form

$$d_{spec} \geq \sum_j \Delta(v_j)\cdot\prod_l d_{j,l} + \sum_k \ell_k \cdot \prod_l d_{k,l}. \tag{5}$$

The constant d_{spec} comes from the specification and the analysis computes coefficients $d_{j,l}$ and $d_{k,l}$. The first sum on the right side of the inequality represents how the error in the parameters propagates to the output and the second sum represents the error caused by the approximate execution of the arithmetic operators.

6.4 Optimization Constraint Construction

If the final precondition generated by the analysis is valid, then the program satisfies its accuracy specification. The validity problem for a precondition leads to a natural method for generating an optimization constraint that limits the set of possible kind configurations of the program to only those that satisfy the program's accuracy specification.

Predicate Validity. Similar to the procedure in Section 5.4, we demonstrate the validity of each of the final precondition's conjuncts,

$$\rho_{spec} \cdot \mathcal{R}^*(Q_{spec}) \leq r \cdot \mathcal{R}^*(Q_A),$$

by showing that 1) the reliability coefficient on the right side of the inequality is bounded from below by that on the left side, specifically that $\rho_{spec} \leq r$, and 2) the generalized joint reliability factor on the left side of the inequality is bounded above by that on the right side, specifically that $\mathcal{R}^*(Q_{spec}) \leq \mathcal{R}^*(Q_A)$.

Bounding the reliability coefficients (and generating appropriate optimization constraints) follows from the techniques presented in Section 5.4. To bound the generalized reliability factors, we generalize the ordering property for joint reliability factors (Equation 2) as follows:

Proposition 1 (Generalized Reliability Factor Ordering)**.**

$$\textit{If } Q_{A1} \Rightarrow Q_{A2} \textit{ then } \mathcal{R}^*(Q_{A1}) \leq \mathcal{R}^*(Q_{A2}).$$

This property follows from the fact that, if Q_{A1} implies Q_{A2}, then the set of approximate program environments that satisfy the predicate Q_{A1} is a subset of the environments that satisfy the predicate Q_{A2}. Therefore, $\mathcal{R}^*(Q_{A1})$, the probability of the environments satisfying Q_{A1}, must be less than or equal to $\mathcal{R}^*(Q_{A2})$, the probability of the environments satisfying Q_{A2}.

Constraint Construction. Given the generalized reliability factor ordering, Chisel's goal is to generate an optimization constraint that ensures that $Q_{spec} \Rightarrow Q_A$ (which therefore ensures that the corresponding conjunct in the precondition is valid). Chisel constructs this constraint via the observation that Q_{spec} has the form $\bigwedge_j d_j \geq \Delta(v_j)$ (Section 6.3.4, Equation 4). Therefore, it is sound to replace each occurrence of $\Delta(v_j)$ in Q_A with the corresponding d_j, yielding a predicate of the form:

$$d_{spec} \geq \sum_j d_j \cdot \prod_l d_{j,l} + \sum_k \ell_k \cdot \prod_l d_{k,l}. \tag{6}$$

The constraint generator takes this accuracy predicate and constructs the optimization constraint. First, it rearranges terms and simplifies numerical constants ($d^* = \sum_j d_j \cdot \prod_l d_{j,l}$ and $d_k^* = \prod_l d_{k,l}$). Since $d_k^* \cdot \ell_k$ denotes the multiplication of the constant d_k^* and the kind configuration $\theta(\ell_k)$, the generator then produces the following optimization constraint for a conjunct:

$$d_{spec} - d^* \geq \sum_k d_k^* \cdot \theta(\ell_k).$$

Identifying Critical Operations. As it generates the optimization constraint, the constraint generator identifies all accuracy expressions in which the coefficient d_k^* has the value ∞ (Section 6.3.2). Such expressions indicate that small deviations in the result of an intermediate operation or a variable value may cause a large deviation of the kernel's output. The constraint generator sets the corresponding kind configuration $\theta(\ell_k)$ to 0 (exact) and removes all terms with such assigned configurations from the final accuracy constraints.

7. Energy Objective Construction

We now define a set of functions that operate on traces of the original program to model the energy consumption of the exact and approximate program executions.

7.1 Absolute Energy Model

Energy Model Specification. We extend the hardware specification from Section 3.1 with the relative energy savings for each approximate arithmetic operation (for simplicity we use α_{int} for all integer and α_{fp} for all floating point instructions) and approximate memory and cache regions (α_{mem} and α_{cache}). The specification also contains the relative energy consumption of the system's components (μ_{CPU}, μ_{ALU}, and μ_{cache}) and relative instruction class energy rates (w_{fp} and w_{oi}).

Energy of System. We model the energy consumed by the system (E_{sys}) when executing a program under configuration θ with the combined energy used by the CPU and memory:

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

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

$$E_{CPU}(\theta) = E_{ALU}(\theta) + E_{cache}(\theta) + E_{other}.$$

Energy of ALU. Each instruction in the hardware specification may have a different energy consumption associated with it. However, for the purposes of our model, we let $\mathcal{E}_{int}$, $\mathcal{E}_{fp}$, $\mathcal{E}_{oi}$ be the average energy consumption (over a set of traces) of an ALU instruction, a FPU instruction, and other non-arithmetic instructions, respectively.

Using the instructions from the traces that represent kernel execution on representative inputs, we derive the following sets: *IntInst* is the set of labels of integer arithmetic instructions and *FPInst* is the set of labels of floating-point arithmetic instructions. For each instruction with a label ℓ, we also let n_ℓ 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 energy consumption) from executing integer and floating-point instructions approximately, respectively. Then, the ALU's energy consumption is:

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

This model assumes that the instruction count in the approximate execution is approximately equal to the instruction count in the exact execution.

Memory Energy. We model the energy consumption of the system memory (i.e., DRAM) using an estimate of the average energy per second per byte of memory, $\mathcal{E}_{mem}$. Given the execution time of all kernel invocations, t, the savings associated with allocating data in approximate memory, α_{mem}, the size of allocated arrays, S_ℓ, and the configurations of array variables in the exact and approximate memories, $\theta(\ell)$, we model the energy consumption of the memory as follows:

$$E_{mem}(\theta) = t \cdot \mathcal{E}_{mem} \cdot \sum_{\ell \in ArrParams} S_\ell \cdot (1 - \theta(\ell) \cdot \alpha_{mem}).$$

Cache Memory Energy. We model the energy consumption of cache memory, $\mathcal{E}_{cache}$, similarly. Let S_c be the size of the cache, α_{cache} the savings of approximate caches. In addition, we need to specify the strategy for determining the size of approximate caches. We analyze the strategy that scales the size of approximate caches proportional to the percentage of the size of the arrays allocated in the approximate main memory. If c_u is the maximum fraction of the approximate cache lines, the energy consumption of the cache is

$$E_{cache}(\theta) = t \cdot \mathcal{E}_{cache} \cdot S_c \cdot (1 - c_u \cdot \frac{\sum_\ell S_\ell \theta(\ell)}{\sum_\ell S_\ell} \cdot \alpha_{cache}).$$

7.2 Relative Energy Model

While the energy model equations from Section 7.1 capture basic properties of energy consumption, the models rely on several hardware design specific parameters, such as the average energy of instructions.

However, we can use these equations to derive a numerical optimization problem that instead uses cross-design parameters (such as the relative energy between instruction classes and the average savings for each instruction) to optimize energy consumption of the program relative to an exact configuration of the program, 0_θ (Section 4.2). For each energy consumption modeling function in the previous section we introduce a corresponding function that implicitly takes 0_θ as its parameter. For example, for the energy consumption of the system, we let $E_{sys} \equiv E_{sys}(0_\theta)$.

System Relative Energy. The energy model contains a parameter that specifies the relative portion of energy consumed by the CPU versus memory, μ_{CPU}. Using this parameter, we derive the relative system energy consumption as follows:

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

CPU Relative Energy. The energy model contains a parameter that specifies the relative portion of energy consumed by the ALU, μ_{ALU}, and cache, μ_{cache} (and $\mu_{other} = 1 - \mu_{ALU} - \mu_{cache}$). We can then derive the relative CPU energy consumption similarly to that for the whole system:

$$\frac{E_{CPU}(\theta)}{E_{CPU}} = \mu_{ALU} \cdot \frac{E_{ALU}(\theta)}{E_{ALU}} + \mu_{cache} \cdot \frac{E_{cache}(\theta)}{E_{cache}} + \mu_{other}.$$

ALU Relative Energy. We apply similar reasoning to derive the relative energy consumption of the ALU:

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

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

Memory And Cache Relative Energy. Applying similar reasoning to the memory subsystem yields the following:

$$\frac{E_{mem}(\theta)}{E_{mem}} = \frac{1}{H} \cdot \frac{t'}{t} \cdot \sum_{\ell \in ArrParams} S_\ell \cdot (1 - \theta(\ell) \cdot \alpha_{mem})$$
$$\frac{E_{cache}(\theta)}{E_{cache}} = \frac{1}{H} \cdot \frac{t'}{t} \cdot \sum_{\ell \in ArrParams} S_\ell \cdot (1 - c_u \cdot \theta(\ell) \cdot \alpha_{cache}),$$

where $H = \sum_\ell S_\ell$ is the total size of heap data. The execution time ratio t'/t denotes possibly different execution time of the approximate program. One can use the results of reliability profiling to estimate this ratio.

8. Optimization Problem Statement

We now state the optimization problem for a kernel computation:

Minimize: $\frac{E_{sys}(\theta)}{E}$

Constraints: $\log(\rho_{spec,i}) - \log(\rho_i) \leq \sum_k \theta(\ell_{k_i}) \cdot \log(\rho_{k_i})$

$d_{spec,i} - d_i^* \geq \sum_k d_{k_i} \cdot \theta(\ell_{k_i}) \qquad \forall i$

Variables: $\theta(\ell_1), ..., \theta(\ell_n) \in \{0,1\}$

The decision variables $\theta(\ell_1), ..., \theta(\ell_n)$ are the configuration kinds of arithmetic instructions and array variables. Since they are integers, the optimization problem belongs to the class of integer linear programs.

Complexity. The number of constraints for a single program path is linearly proportional to the number of kernel outputs (the return value and the array parameters). The number of paths that Chisel's precondition generator produces is in the worst case exponential in the number of control flow divergence points. However, in practice, one can use the simplification procedure from [7, Section 5.4], which can identify most of the path predicates as redundant and remove them during the analysis. Out of the remaining predicates, Chisel can immediately solve those that involve only numerical parameters and pass only the optimization constraints with kind configurations to the optimization solver.

The number of decision variables is proportional to the number of instructions and array parameters in a kernel. In general, integer linear programming is NP complete with respect to the number of decision variables. However, existing solvers can successfully and efficiently solve many classes of integer linear programs with hundreds of variables.

We describe two techniques that can reduce the size of the generated optimization problem. First, the precondition generator can create constraints at coarser granularities. For example, a single decision variable may represent program statements, basic blocks, or loop bodies. (Section 8.1). Second, Chisel can separately optimize the invoked functions that implement hierarchically structured kernels (Section 8.4).

Extensions. In the rest of this section we describe several extensions to Chisel's optimization algorithm.

8.1 Operation Selection Granularity

When the number of decision variables in the optimization problem for a large kernel computation is too large to solve given the computational resources at-hand, a developer may instruct the optimizer to mark all instructions in a block of code with the same kind (i.e., all exact or all approximate). The optimization algorithm assigns a single label ℓ to all operations within this block of code. This approach reduces the number of decision variables and – therefore – the resources required to solve the optimization problem.

8.2 Overhead of Operation Mode Switching

Some approximate architectures impose a performance penalty when switching between exact and approximate operation modes due to e.g. dynamic voltage or frequency scaling. Therefore, for these architectures it is beneficial to incorporate the cost of switching into the optimization problem. For example, the constraint generator can produce additional constraints that bound the total switching overhead [38].

To specify this additional constraint, we let ℓ_i and ℓ_{i+1} be the labels of two adjacent arithmetic instructions. Next, we define auxiliary counter variables $s_i \in \{0,1\}$ such that

$$s_i \geq \theta(\ell_i) - \theta(\ell_{i+1}) \wedge -s_i \leq \theta(\ell_i) - \theta(\ell_{i+1}).$$

Finally, we specify the constraint $\sum_i s_i \leq B$ to limit the total number of mode changes to be below the bound B.

8.3 Array Index Computations and Control Flow

Instead of relying on support for failure-oblivious program execution (Section 3.3), Chisel can further constrain the set of optimized instructions to exclude instructions that compute array indices and/or affect the flow of control. To ensure that approximate computation does not affect an expression that computes an array index or a branch condition, a dependence analysis can compute the set of all instructions that contribute to the expression's value. Chisel then sets the labels of these instructions to zero to indicate that the instructions must be exact.

8.4 Function Calls

To analyze function calls, one can use the following strategies:

Inlining. Chisel's preprocessor inlines the body of the called function before the precondition generation analyses.

Multiple Existing Implementations. A called function f may have multiple implementations, each with its own reliability specification. The specification of each of the m implementations of f consists of the function's reliability specification $\rho_{f,i} \cdot \mathcal{R}(\cdot)$ and estimated energy savings $\alpha_{f,i}$.

For n calls to the function f in the kernel, the constraint generator specifies the labels $\ell_{f,1,1}, \ldots, \ell_{f,m,n}$. The reliability expression for a k-th call site becomes $\prod_i \rho_{f,i}^{\ell_{f,i,k}}$. The relative ALU energy consumption expression for the same call site is $\mu_{f,k} \cdot (1 - \sum_i \theta(\ell_{f,i.k}) \cdot \alpha_{f,i})$. A trace profiler can record the count of instructions that the exact computation spends in each called function to calculate the parameters $\mu_{f,k}$.

We also specify a constraint $\sum_{i=1}^{m} \theta(\ell_{f,i.k}) = 1$ for each call site to ensure that the optimization procedure selects exactly one of the alternative implementations of f.

Inferring Reliability Specification. Instead of selecting from one of the predefined reliability specifications, one can use the optimization procedure to find the acceptable reliability degradation of the called function f that will satisfy the reliability specification of the caller function. The constraint generator can then be extended to directly model the logarithm of the reliability as a continuous decision variable $\rho'(\ell_f) \leq 0$ (ℓ_f is the label of f).

For the energy consumption expression, the optimization requires the developer to provide a function $\alpha_f(\rho'(\ell_f))$, which specifies a lower bound on the energy savings. To effectively use an optimization solver like Gurobi, this function is required to be linear (the optimization problem is a mixed integer linear program), or quadratic (the optimization problem is a mixed integer quadratic program).

8.5 Hardware with Multiple Operation Specifications

To support hardware platform with arithmetic operations and memory regions with multiple reliability/savings specifications $(\rho_{op,i}, \alpha_{op,i})$, we can use an approach analogous to the one for functions with multiple implementations. Specifically, each arithmetic operation can be analyzed as one such function. Analogously, to specify one of k approximate memory regions for a parameter v, the generator defines the labels $\ell_{v,1}, \ldots, \ell_{v,k}$. It

generates the reliability expression $\prod_i \rho_{mop,i}^{\ell_{v,i}}$ for each memory operation and the memory savings expression $\sum_i \theta(\ell_{v,i}) \cdot \alpha_{mem,i}$ for each array parameter. To select a single memory region, the generator produces the constraint $\sum_i \theta(\ell_{v,i}) = 1$.

8.6 Multiple Kernels

A program may contain multiple approximate kernels. To adapt Chisel's workflow, we consider appropriate modifications to the reliability profiling and the optimization.

Reliability Profiling. The approximate execution of one kernel may affect the inputs and the execution of the other kernels. Therefore, to find the reliability specifications of multiple kernels, the reliability profiler enumerates parts of the induced multidimensional search space. First, the one-dimensional profiler (Section 2.1) finds the lower reliability bound of each kernel. Then, to find the configuration of kernel reliability specifications that yield an acceptably accurate result, the profiler can systematically explore the search space, e.g. using strategies analogous to those that find configurations of accuracy-aware program transformations [16, 24, 26, 40] from a finite set of possible configurations. The profiler then returns configurations that closely meet the accuracy target, ordered by the reliability of the most time-consuming kernels.

Optimization. The optimization algorithm for multiple kernels needs to consider only the allocation of arrays, since the ALU operations are independent between kernels.

The multiple kernel optimization operates in two main stages. In the first stage, it computes the energy savings of each individual kernel for all combinations of shared array variables. Conceptually, if the shared variables are labeled as ℓ_1,...,ℓ_k, the optimization algorithm calls the basic optimization problems for all combinations of the kind configurations $\theta(\ell_1)$,...,$\theta(\ell_k)$, while pruning the search tree when the algorithm already identifies that a subset of labels cannot satisfy the reliability bound.

In the second stage, the analysis searches for the maximum joint savings of the combination of m kernels. It searches over the combination of individual kernel results for which all array parameters have the same kind configuration, i.e., $\theta^{(1)}(\ell_i) = ... = \theta^{(m)}(\ell_i)$ for each $i \in \{1,...,k\}$. The algorithm returns the combination of kernels with maximum joint energy savings, which is a sum of the kernels' savings weighted by the fraction of their execution time. While, in general, the number of individual optimization problems may increase exponentially with the number of shared array variables k, this number is typically small and the search can remain tractable.

9. Evaluation

We evaluate Chisel for several applications over a parameterized space of approximate hardware designs. Our evaluation consists of the following parts:

- **Sensitivity Profiling.** We present how sensitivity profiling can help developers effectively identify an appropriate reliability specification for an application.
- **Optimization Problem Size.** We present statistics that characterize the size of Chisel's optimization problem.
- **Energy Savings.** We present the percentage of potential energy savings that Chisel uncovered.
- **Output Quality.** We present the resulting end-to-end sensitivity metric for the execution of the synthesized approximate benchmarks on a set of test inputs.

9.1 Chisel Implementation

We have implemented Chisel using OCaml. The framework consists of several passes. The translation pass produces an equivalent C program for an input file with Rely functions. The trace profiler pass instruments the C program to collect instruction traces used to compute the frequencies of instructions (n_{int},n_{fp}, and n_{oi}) in the energy objective.

The analysis pass generates the objective, the reliability constraints, and the accuracy constraints. To solve the optimization problem, we use Gurobi mixed integer programming solver [15]. Finally, the transformation pass uses the optimization problem solution to generate a kernel code with approximate instructions and memory annotations.

The framework also contains a fault injection pass, which, given the approximate kernel and the hardware error model, injects errors at approximate operations and collects the execution statistics of the computation.

9.2 Hardware Reliability and Energy Specifications

We use the reliability and energy specifications for approximate hardware presented in [36, Table 2] to instantiate our approximate hardware specification, ψ. We reproduce this table in Section D of Appendix [22]. It defines three configurations, denoted as *mild*, *medium* and *aggressive*, for arithmetic instructions, caches, and main memories respectively. We consider only the unreliable arithmetic operations (that produce the correct results with specified probability) and unreliable memories.

System Parameters. To compute the overall system savings (Section 7.2), we use the server configuration parameters specified in [36, Section 5.4]: CPU consumes $\mu_{CPU} = 55\%$ of energy and the main memory consumes the remaining 45%; the ALU consumes $\mu_{ALU} = 65\%$ of CPU's energy and the cache consumes the remaining $\mu_{cache} = 35\%$ energy.

The sizes of the reliable and approximate regions of the main memory are determined before the execution of the kernel computations and remain fixed until all kernel computations finish. We assume that the capacity of the approximate region of the cache (that can store approximate heap data) is twice that of the reliable cache that contains instructions and reliable data, and therefore $c_u = 67\%$.

Error Model. The error injection pass and its runtime insert faults in the synthesized computation with the frequency specified by the hardware specification. For integer and floating point ALU operations, the error model returns a fully random result (as in [36]). For read and write memory errors, the error model flips from one (with highest probability) up to three bits (with lowest probability) in the word.

Benchmark	Size (LoC)	Kernel (LoC)	Time in Kernel %	Array Parameter Count / Heap %	Representative Inputs (Profile/Test)	Sanity Test	Sensitivity Metric
scale	218	88	93.43%	2 / 99%	13 (5/8)	×	Peak Signal-to-Noise Ratio
dct	532	62	99.20%	2 / 98%	13 (5/8)	×	Peak Signal-to-Noise Ratio
idct	532	93	98.86%	2 / 98%	9 (3/6)	×	Peak Signal-to-Noise Ratio
blackscholes	494	143	65.11%	6 / 84.4%	24 (8/16)	✓	Relative Portfolio Difference
sor	173	23	82.30%	1 / 99%	20 (6/14)	✓	Average Relative Difference

Table 1: Benchmark Description

9.3 Benchmarks

We implemented a set of benchmarks from several application domains. The benchmarks were selected because they tolerate some amount of error in the output.

Scale. Scales an image by a factor provided by the user. The kernel computes the output pixel value by interpolating over neighboring source image pixels.

Discrete Cosine Transform (DCT). A compression algorithm used in various lossy image and audio compression methods. The kernel computes a frequency-domain coefficient of an 8x8 image block.

Inverse Discrete Cosine Transform (IDCT). Reconstructs an image from the coefficients generated by DCT. The kernel reconstructs a single pixel from the frequency domain grid.

Black-Scholes. Computes the price of a portfolio of European Put and Call options using the analytical Black-Scholes formula. The kernel calculates the price of a single option. Our implementation is derived from the benchmark from the PARSEC benchmark suite [42].

Successive Over-relaxation (SOR). The Jacobi SOR computation is a part of various partial differential equation solvers. The kernel averages the neighboring matrix cells computed in the previous iteration. It is derived from the benchmark from the SciMark 2 suite [43].

Table 1 presents an 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 ("Time in Kernel %") presents the percentage of instructions that the execution spends in the kernel computation. Column 5 ("Array Parameter Count/Heap %") presents the number of array arguments and the percentage of heap allocated space that these variables occupy. Column 6 ("Representative Inputs") presents the number of representative inputs collected for each computation. Column 7 ("Sanity Test") presents whether the computation contains a sanity test that ensures the integrity of its result. Column 8 ("Sensitivity Metric") presents the sensitivity metric of the computation.

Representative Inputs. For each benchmark, we have selected several representative inputs. The analysis uses a subset of these inputs (designated as "Profile") to obtain the estimates of the instruction mixes and construct the objective function of the optimization problem. We use the remaining inputs (designated as "Test") to evaluate the synthesized approximate computation.

Benchmark	Reliability Bound	Sensitivity metric	
		Average	Conservative
scale	0.995	30.93 ± 0.95 dB	23.01 dB
dct	0.99992	27.74 ± 1.32 dB	22.91 dB
idct	0.992	27.44 ± 0.49 dB	20.96 dB
blackscholes	0.999	0.005 ± 0.0005	0.05
sor	0.995	0.058 ± 0.034	≥ 1.0

Table 2: Software Specification PSNR

Sensitivity Metrics. For the three image processing benchmarks (Scale, DCT, and IDCT) we use peak signal to noise ratio between images produced by the original and the synthesized versions of the benchmark. Specifically for DCT, the sensitivity metric first converts the image from the frequency domain and computes the PSNR on the resulting image.

For Blackscholes, we have used the relative difference between the sum of the absolute errors between the option prices and the absolute value of the price of the portfolio (the sum of all option values returned by the fully accurate program). For SOR, the sensitivity metric is the average relative difference between the elements of the output matrix.

Sanity Tests. Two of the benchmark computations have built-in sanity test computations that ensure that the intermediate or final results of the computation fall within specific intervals. These computations typically execute for only a small fraction of the total execution time. The Blackscholes sanity test uses a *no-arbitrage* bound [3] on the price of each option to filter out executions that produce option prices that violate basic properties of the Black-Scholes model. The SOR benchmark checks whether the computed average is between the minimum and maximum array value.

If the sanity test computation fails, the approximate computation may skip updating the result (as in SOR), or reexecute the computation (as in Blackscholes). In the case of reexecution, the overall savings are scaled by the additional execution time of the kernel computation.

9.4 Sensitivity Profiling

To find reliability specifications for the benchmark applications, the sensitivity profiler relates the reliability degradation of a kernel computation with its end-to-end sensitivity metric.

Methodology. For each profiling input, we perform 100 fault injection experiments. As in Section 2.1, we use the sensitivity profiler to compute the average sensitivity metric value (over the space of possible injected faults) for multiple reliability bounds using a developer-provided sensitivity testing procedure. For each benchmark, we select one reliability bound that yields

Benchmark	Optimization Variables	Reliability Constraints
scale	147	4
dct	121	1
idct	104	1
blackscholes	77	2
sor	36	1

Table 3: Optimization Problem Statistics

an acceptable sensitivity metric. We also analytically derive conservative estimates of the average sensitivity metric.

Table 2 presents the final reliability specifications for the benchmarks. Column 2 presents the reliability bound. Column 3 presents the average metric obtained from sensitivity testing. Column 4 presents the analytic conservative lower bound on the average sensitivity metric.

Image Benchmarks. For Scale and IDCT, the sensitivity testing procedure (like the one from Section 2.1) modifies a single pixel. For DCT, the sensitivity testing procedure changes a single coefficient in the 8x8 DCT matrix. To compute the lower bound on the average PSNR, we use an analytical expression from Section 2.1. Note that DCT has smaller average PSNR than the other two benchmarks, as a single incorrectly computed coefficient can make 64 pixels in the final image incorrect.

Blackscholes. The sensitivity testing procedure conservatively estimates the error at the end of the computation by returning either the upper or the lower no-arbitrage bound (whichever is more distant from the exact option value). For reliability bound 0.999, the average absolute error is $\$150.6$ ($\pm\$1.63$), while the average value of the portfolio is $\$28361.4$. Therefore, the relative error of the portfolio price is approximately 0.50%. To derive a conservative analytic expression for the deviation, we use the no-arbitrage bound formula, while assuming that the price of the portfolio is at least $\$4000$ (e.g., each option in a portfolio with 4K options is worth at least a dollar) and the strike price is less than $\$200$.

SOR. For SOR, the sensitivity testing strategy returns a random value within the typical range of the input data. Since the computation performs multiple updates to the elements of the input matrix, the worst-case relative error typically exceeds 100%.

9.5 Optimization Problem Solving

Chisel's optimization algorithm constructs the optimization problem and calls the Gurobi solver. Table 3 presents for each benchmark the number of variables (Column 2) and the number of constraints (Column 3) constructed by Chisel. For each of these problems, Gurobi took less than a second to find the optimal solution (subject to optimality tolerance bound 10^{-9}) on an 8-core Intel Xeon E5520 with 16 GB of RAM.

9.6 Energy Savings

We next present the potential savings that Chisel's optimization uncovered. We relate the savings obtained for the traces of profiled inputs to 1) the maximum possible savings when the reliability bound is 0.0 and 2) the savings for the previously unseen test inputs.

Methodology. To get the statistics on the approximate execution of the benchmarks, we run the version of the benchmark transformed using the fault injection pass. We ran the benchmark 100 times for each test input. To estimate the energy savings, we use the instruction counts from the collected traces and the expressions derived in Section 7.

Results. Table 5 presents the system savings that the Chisel's optimization algorithm finds for the kernel computations. Column 2 ("Reliability Bound") presents the target reliability that we set according to the exploration in Section 9.4. Column 3 ("Potential Savings") presents the maximum possible savings when all instructions and memory regions have medium configuration and the result's reliability bound is 0.0 – so that all operations can be unreliable and all arrays can be stored in unreliable memory.

The remaining columns present the system savings when running the approximate kernels for different hardware specifications. We represent the system configurations as triples of the form *CPU/Cache/Main*, denoting the reliability/saving configuration of CPU instructions, cache memories, and main memories, respectively. We use the letters "m" and "M"' to denote the mild and medium reliability/savings configuration of the system component from [36, Table 2]. We omit the aggressive configurations as they yield no savings or only small savings for the reliability bounds of our benchmarks. For instance, the configuration "M/m/M" denotes a medium configuration for CPU instructions, mild configuration for the cache memory, and medium configuration for the main memory. The column "Profile" contains the savings that Chisel finds for the inputs used in sensitivity profiling. The column "Test" contains savings computed from the traces of inputs not used during sensitivity profiling.

Overall, for these benchmarks and hardware specification, the majority of savings (over 95%) come from storing data in unreliable memories. For Scale and SOR, Chisel marks all array parameters and a significant portion of instructions as unreliable for the configuration "M/M/M". For Scale, the optimization achieves over 95% (19.35% compared to 20.28%) of the maximum savings. For SOR it obtains more than 98% of the maximum possible savings.

In general, the hardware parameters affect the result that Chisel produces. For instance, Chisel cannot apply any approximation for the medium main memory configuration for DCT (which is the benchmark with the strictest reliability bound) – it produces a kernel in which all operations are reliable. However, for mild memory and cache configurations, the optimization can obtain up to 43% of the maximum possible savings.

For IDCT, Chisel obtains greater savings for mild ("m") configurations of the unreliable memories, because it can place both array parameters to the kernel as unreliable, for the savings of 67% of the maximum possible savings. When the memory configurations are at the medium ("M") level, Chisel can place only one array parameter in unreliable memory.

Benchmark	Reliability Bound	Potential Savings	m/m/m		M/m/m		M/M/m		M/m/M		M/M/M	
			Profile	Test	Profile	Test	Profile	Test	Profile	Test	Profile	Test
scale	0.995	20.28%	14.11%	14.16%	14.22%	14.28%	15.39%	15.42%	18.17%	18.20%	19.35%	19.36%
dct	0.99992	20.09%	6.73%	6.72%	6.73%	6.73%	0.00%	–	8.72 %	8.72%	0.00 %	–
idct	0.992	19.96%	13.38 %	13.38%	13.40%	13.40%	7.34%	7.34%	8.70 %	8.70%	9.32 %	9.32%
blackscholes	0.999	17.39%	9.87 %	9.79%	9.90%	9.81%	5.38%	5.35%	6.36%	6.32%	4.40 %	4.52%
sor	0.995	20.07%	14.52%	14.50%	14.83%	14.87%	16.07%	16.07%	18.81%	18.70%	19.83%	19.43 %

Table 4: Energy Savings (Configurations: 'm' denotes mild and 'M' medium CPU/Cache/Memory approximation)

Benchmark	Reliability Bound	m/m/m	M/m/m	M/M/m	M/m/M	M/M/M
scale	0.995	44.79 ± 2.51	35.30 ± 1.95	34.07 ± 1.19	33.13 ± 1.33	32.31 ± 1.08
dct	0.99992	30.34 ± 3.84	30.37 ± 4.41	–	29.76 ± 4.81	–
idct	0.992	31.28 ± 0.80	30.45 ± 0.75	30.36 ± 0.19	30.36 ± 0.18	30.35 ± 0.20
blackscholes	0.999	0.0002 ± 0.00004	0.0006 ± 0.00008	0.0005 ± 0.00006	0.0005 ± 0.0008	0.0005 ± 0.0008
sor	0.995	0.029 ± 0.022	0.051 ± 0.032	0.046 ± 0.038	0.086 ± 0.090	0.080 ± 0.074

Table 5: Sensitivity Metric Results for Test Inputs

For Blackscholes, Chisel also selects different combinations of unreliable input array parameters based on the configurations of the main and cache memories and exposes up to 57% of the maximum possible savings. Blackscholes reexecutes some of its computation (when detected by the sanity test), but this reexecution happens for only a small fraction of the options (less than 0.03% on average) and has a very small impact on program's execution time and energy consumption.

For all benchmarks, the energy savings obtained on the test inputs typically have a deviation less than 3% from the savings estimated on the profiling inputs.

9.7 Output Quality

We next present the end-to-end sensitivity metrics results for the executions of programs with synthesized kernels.

Methodology. We instrumented the unreliable operations selected by the optimizer and injected errors in their results according to the hardware specification and error model.

Results. Table 5 presents the end-to-end sensitivity of the optimized benchmarks. Columns 1 and 2 present the benchmark and the reliability bound. The remaining columns present the mean and the standard deviation of the distribution of the error metric. The number of faults per execution ranges from several (Blackscholes) to more than a thousand (DCT and IDCT).

The sensitivity metric of Scale, DCT, and IDCT is the average PSNR metric (higher value of PSNR means better accuracy). We note that the value of the metric for the synthesized computation is similar to the sensitivity profiling results (Table 2). The accuracy of Scale and IDCT increases for the mild configuration of arithmetical operations, as the frequency of faults and therefore the number of faulty pixels caused by computation decreases. The higher variance in DCT is caused by the inputs of a smaller size, where each fault can significantly impact PSNR.

The accuracy of blackscholes exceeds the accuracy predicted by the sensitivity testing (up to 0.06% on test inputs vs. 0.5% in sensitivity testing). The error injection results for SOR are less accurate than the sensitivity profiling results for medium main memory configurations (8.0% vs. 5.8%). We attribute this lower accuracy to the fact that the sensitivity profiling does not inject errors in the read-only edge elements of the input matrix.

9.8 Kernel Transformations

We now focus on the kernels that Chisel's optimization algorithm generated. For each benchmark, we examined the kernel with maximum energy savings.

Scale. We discussed the transformation in Section 2.4.

DCT. Chisel places the array that contains the pixels of the output image in the unreliable memory. All arithmetic operations remain reliable, as they all occur in a nested loop.

IDCT. Chisel places both arrays (these arrays contain the pixels of the source and output image) in unreliable memory. Chisel also selects 14% of the arithmetic instructions as unreliable. The instrumented instructions include those that affect the condition of one of the inner bounded loops. Since this loop executes at most 8 iterations (which is enforced by the language semantics), this transformation does not have a visible impact on the energy consumption of the kernel.

Blackscholes. Chisel places 5 out of 6 input arrays in unreliable memory. These arrays contain different input parameters for computing the blackscholes equation. In addition, Chisel selects 7% of the arithmetic operations as unreliable that fit within the reliability bound.

SOR. Chisel places the input array in unreliable memory and selects 82% of the arithmetic operations as unreliable. These unreliable instructions do not affect the control flow.

10. Related Work

Accuracy and Reliability Specifications. Researchers have previously used dynamic sensitivity testing to obtain combined accuracy and reliability guarantees for approximate computations [2, 8, 24, 26, 32, 33, 35, 40]. These guarantees are statistical in that they are based on end-to-end sensitivity testing on representative inputs. Researchers have also developed static analysis techniques that provide similar guarantees [5, 6, 9–11, 25, 47].

Snap combines input fuzzing with dynamic execution and influence tracing to quantitatively characterize the sensitivity of the computation to changes to the input [8]. ASAC [35] characterizes the sensitivity of the computation to changes in the intermediate program data. Bao et al. [2] use whitebox sampling to find discontinuities in numerical computations.

Chisel's sensitivity profiling quantitatively relates the rate of incorrect kernel results to the quality of the result that the program produces. Chisel's sensitivity profiling differs from previous techniques in the source of the noise (incorrect kernel results as opposed to changes in the computation, inputs, or program data) and the goal of the analysis. The goal of the Chisel sensitivity analysis is to obtain the Chisel reliability specifications. The goal of Snap, in contrast, is to identify input fields, intermediate program data, and program regions that must execute correctly and those that can tolerate errors. The goal of ASAC is to discover approximable program data.

In contrast to these dynamic techniques, researchers have developed static program analyses for reasoning about programs transformed using accuracy-aware transformations [5, 6, 10, 25, 47], for verifying continuity of computations [9, 10], and for verifying the precision of numerical computations [11]. Researchers have also developed techniques for reasoning about reduced-bitwidth floating point computations [14, 28]. In comparison, Chisel's analysis unifies static reasoning about reliability and accuracy, and dynamic reasoning about performance/energy with the goal to navigate the tradeoff space induced by approximate hardware platforms.

Software Approximate Computation. Researchers have developed many systems that apply approximate computing techniques in software to reduce the amount of energy and/or time required to execute computations running on standard exact, reliable hardware platforms [20, 24, 25, 32, 33, 40, 47]. In Chisel, the source of the approximation is the hardware – Chisel synthesizes acceptably reliable and accurate software that executes on unreliable approximate hardware platforms.

Approximate Hardware Platforms. Researchers have previously proposed multiple hardware architecture designs that improve the performance of processors [12, 13, 17–19, 27, 29, 36, 41, 44, 45] or memories [13, 21, 37, 39] at the expense of decreased reliability or accuracy.

Programming Models for Approximate Hardware. Rely provides a specification language that developers can use to specify computational reliability requirements and an analysis that verifies that Rely programs satisfy these requirements when run on unreliable hardware platforms [7]. Flikker provides a set of C language extensions that enable a developer to specify data that can be stored in approximate memories [21]. EnerJ provides a type system that a developer can use to specify unreliable data that can be stored in unreliable memory or computed using unreliable operations [36]. The EnerJ type system ensures the isolation of unreliable computations. More recently, it was extended to support the specification and inference of quantitative reliability types [4]. Unlike these previous techniques, which rely solely on the developer to identify reliable and unreliable operations and data, Chisel automates the selection of unreliable operations and data while ensuring that the generated program satisfies its reliability specification.

ExpAX is a framework for expressing accuracy and reliability constraints for a subset of the Java language [31]. ExpAX uses a genetic programming optimization algorithm to search for approximations that minimize the energy consumption of the computation over a set of program traces. Chisel, in contrast, uses mathematical programming to guarantee that the resulting program satisfies its reliability specification (the genetic algorithm in ExpAX provides no such guarantee).

Topaz is a task-based language that allows the developer to specify tasks that execute on approximate hardware that may produce arbitrarily inaccurate results. Topaz includes an outlier detector that automatically detects and reexecutes unacceptably inaccurate tasks [1].

Mathematical Optimization in Program Analysis. There is a long history of using mathematical optimization to solve traditional compiler optimization problems such as instruction scheduling and register allocation [30]. EPROF uses integer linear programming to schedule parallel streaming applications, taking into account the execution time, energy consumption, and task error rate [46]. Saputra et al. use integer linear programming to place instructions that dynamically scale the voltage and clock rate of the underlying hardware platform. The goal is to exploit the tradeoff between execution time and energy consumption [38].

We have previously used linear programming as a component of an approximation algorithm that finds an ε-optimal expected error/performance tradeoffs for map-fold computations automatically transformed using randomized program transformations [47]. Chisel similarly uses mathematical programming to optimize energy while providing reliability and accuracy guarantees. In general, we see mathematical programming, with its ability to optimize an objective while preserving a set of constraints, as a natural fit for many approximate computing problems, which typically aim to optimize a resource consumption objective, such as energy or time, while providing acceptable execution, which may be captured by the constraints in the mathematical program.

11. Conclusion

As the need for energy-efficient computing becomes more acute, approximate hardware platforms become an increasingly attractive target for computationally intensive applications that must execute efficiently. But successfully navigating the resulting reliability and/or accuracy versus energy tradeoff space requires precise, detailed, and complex reasoning about how the approximate hardware platform interacts with the approximate computation. We present a new system that automatically maps the computation onto the underlying approximate hardware platform and minimizes energy consumption while ensuring that the computation executes with acceptable reliability and/or accuracy. 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 approximate hardware components. Such systems are clearly required if developers are to produce software that can effectively exploit emerging energy-efficient approximate hardware platforms.

Acknowledgments

We thank Abbas Banaiyan, Harshad Kasture, Deokhwan Kim, Velibor Misic, Majid Shoushtari, Stelios Sidiroglou, and the anonymous referees for the useful comments on the previous versions of this work. We note our previous technical report [23].

This research was supported in part by NSF (Grants CCF-1036241, CCF-1138967, and IIS-0835652), DOE (Grant DE-SC0008923), and DARPA (Grants FA8650-11-C-7192, FA8750-12-2-0110, and FA-8750-14-2-0004).

References

[1] S. Achour and M. Rinard. Energy-efficient approximate computation in Topaz. Technical Report MIT-CSAIL-TR-2014-016, MIT, 2014.

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

[3] J. Birge. *Optimization Methods in Dynamic Portfolio Management (Chapter 20)*. Elsevier, 2007.

[4] B. Boston, A. Sampson, D. Grossman, and L. Ceze. Tuning approximate computations with constraint-based type inference. WACAS, 2014.

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

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

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

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

[9] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. POPL, 2010.

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

[11] E. Darulova and V. Kuncak. Sound compilation of reals. POPL, 2014.

[12] P. Düben, J. Joven, A. Lingamneni, H. McNamara, G. De Micheli, K. Palem, and T. Palmer. On the use of inexact, pruned hardware in atmospheric modelling. *Philosophical Transactions of the Royal Society*, 372, 2014.

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

[14] A. Gaffar, O. Mencer, W. Luk, P. Cheung, and N. Shirazi. Floating-point bitwidth analysis via automatic differentiation. FPT, 2002.

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

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

[17] H. Kaul, M. Anders, S. Mathew, S. Hsu, A Agarwal, F. Sheikh, R. Krishnamurthy, and S. Borkar. A 1.45ghz 52-to-162gflops/w variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm cmos. ISSCC, 2012.

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

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

[20] T. Lin, S. Tarsa, and H. T. Kung. Parallelization primitives for dynamic sparse computations. HotPar, 2013.

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

[22] S. Misailovic, M. Carbin, S. Achour, Z Qi, and M. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels (appendix). http://groups.csail.mit.edu/pac/chisel, 2014.

[23] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard. Reliability-aware optimization of approximate computational kernels with rely. Technical Report MIT-CSAIL-TR-2014-001, MIT, 2014.

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

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

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

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

[28] W. Osborne, R. Cheung, J. Coutinho, W. Luk, and O. Mencer. Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. FPL, 2007.

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

[30] J. Palsberg and M. Naik. Ilp-based resource-aware compilation. *Multiprocessor Systems-on-Chips, Elsevier*, 2004.

[31] J. Park, X. Zhang, K. Ni, H. Esmaeilzadeh, and M. Naik. Expectation-oriented framework for automating approximate programming. Technical Report GT-CS-14-05, Georgia Institute of Technology, 2014.

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

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

[34] 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.

[35] P. Roy, R. Ray, C. Wang, and W. Wong. Asac: automatic sensitivity analysis for approximate computing. LCTES, 2014.

[36] 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.

[37] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. MICRO, 2013.

[38] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. LCTES/SCOPES, 2002.

[39] M. Shoushtari, A. Banaiyan, and N. Dutt. Relaxing manufacturing guard-bands in memories for energy savings. Technical Report CECS TR 10-04, UCI, 2014.

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

[41] R. St Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger. General-purpose code acceleration with limited-precision analog computation. ISCA, 2014.

[42] Parsec Benchmark Suite. http://parsec.cs.princeton.edu/.

[43] SciMark2 Benchmark Suite. math.nist.gov/scimark2/.

[44] J. Tong, D. Nagle, and R. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Transactions on Very Large Scale Integrated Systems*, 2000.

[45] S. Venkataramani, V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. MICRO, 2013.

[46] Y. Yetim, S. Malik, and M. Martonosi. Eprof: An energy/performance/reliability optimization framework for streaming applications. ASP-DAC'12.

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