ExpAX: A Framework for Automating Approximate Programming

Jongse Park Xin Zhang Kangqi Ni Hadi Esmaeilzadeh Mayur Naik
Georgia Institute of Technology
{jspark, xin.zhang, vincent.nkq}@gatech.edu {hadi, naik}@cc.gatech.edu

Abstract

We present ExpAX, a framework for automating approximate programming. ExpAX consists of these three components: (1) a programming model based on a new kind of program specification, which we refer to as *error expectations*. Our programming model enables programmers to implicitly relax the accuracy constraints without explicitly marking operations as approximate; (2) an approximation safety analysis that automatically infers a safe-to-approximate set of program operations; and (3) an optimization that automatically marks a subset of the safe-to-approximate operations as approximate while statistically adhering to the error expectations. We evaluate ExpAX on a diverse set of Java applications. The results show that ExpAX provides significant energy savings (up to 35%) with large reduction in programmer effort (between 3× to 113×) while providing formal safety and statistical quality-of-result guarantees.

1. Introduction

Energy efficiency is a first-class design constraint in computer systems. Its potential benefits go beyond reduced power demands in servers and longer battery life in mobile devices. Improving energy efficiency has become a requirement due to limits of device scaling and the dark silicon problem [7]. As per-transistor speed and efficiency improvements diminish, radical departures from conventional approaches are needed to improve the performance and efficiency of general-purpose computing. One such departure is general-purpose approximate computing. Conventional techniques in energy-efficient computing navigate a design space defined by the two dimensions of performance and energy, and typically trade one for the other. General-purpose approximate computing explores a third dimension, that of error. It concerns methodically relaxing the robust digital abstraction of near-perfect accuracy and trading the accuracy of computation for gains in both energy and performance.

Recent research [1, 5, 6, 8–16, 18, 21, 25, 27] has shown that many emerging applications in both cloud and mobile services inherently have tolerance to approximation. These applications span a wide range of domains including web search, big-data analytics, machine learning, multimedia, cyber-physical systems, pattern recognition, and many more. In fact, there is an opportunity due to the current emergence of approximation-tolerant applications and the growing unreliability of transistors as technology scales down to atomic levels [7]. For these diverse domains of applications, providing automated programming models and compiler optimizations for approximation can provide significant opportunities to improve performance and energy efficiency at the architecture and circuit level by eliminating the high tax of providing full accuracy [6, 8, 9, 18].

As Figure 1 shows, state-of-the-art programming models for approximate computing such as EnerJ [21] and Rely [4] require programmers to *manually and explicitly* declare low-level details, such as the specific variables and operations to be approximated, and provide safety [21] or quality-of-result guarantees [4]. In contrast, we focus on providing an automated framework, ExpAX, that allows programmers to express concise, high-level, and intuitive *error expectation specifications*, and automatically infers the subset of program operations to approximate while statistically satisfying those error expectations. ExpAX enables programmers to implicitly declare *which* parts of the program are safely approximable while explicitly expressing *how much* approximation is acceptable. ExpAX has three phases: (1) programming, (2) analysis, and (3) optimization that automatically select which operations to approximate while providing formal safety and statistical quality-of-result guarantees.

First, ExpAX allows programmers to *implicitly* relax the accuracy constraints on program data and operations by *explicitly* specifying error expectations on program outputs (Section 2). Our programming model provides the syntax and semantics for specifying such high-level error expectations. In this model, the program itself without the specified expectations carries the strictest semantics in which approximation is not allowed. Programmers add the expectations to implicitly relax the accuracy requirements without explicitly specifying where the approximation is allowed. Second, ExpAX includes an *approximation safety analysis* that automatically infers a candidate set of program operations that can be approximated without violating program safety (Section 3). The program outputs on which the programmer has specified error expectations are inputs to this analysis. Third, ExpAX includes an optimization framework that selectively marks a subset of the candidate operations as approximate while statistically adhering to the specified error expectations (Section 4). The error expectations guide the optimization procedure to strike a balance between quality-of-result degradation and energy savings.

The optimization is a two-phase procedure. In the first phase, it uses a genetic algorithm to identify and exclude safe-to-approximate operations that if approximated lead to significant quality degradation. In the second phase, a greedy algorithm further refines the remaining approximable opera-

```
@Approx int foo (
    @Approx int x[][],
    @Approx int y[]) {
  @Approx int sum := 0;
  for i = 1 .. x.length
    for j = 1 .. y.length
      sum := sum + x[i][j] * y[j];
  return sum;}
```

(a) EnerJ [21]

```
int <0.90*R(x, y)> foo (
      int <R(x)> x[][] in urel,
      int <R(y)> y[] in urel) {
  int sum := 0 in urel;
  for i = 1 .. x.length
    for j = 1 .. y.length
      sum := sum +. x[i][j] *. y[j];
  return sum;}
```

(b) Rely [4]

```
int foo (int x[][], int y[]) {
  int sum := 0;
  for i = 1 .. x.length
    for j = 1 .. y.length
      sum := sum + x[i][j] * y[j];
  accept magnitude(sum) < 0.10;
  return sum;
}
```

(c) ExpAX

Figure 1: For the same code, (a) EnerJ requires 4 annotations on program variables, (b) Rely requires 8 annotations on program variables and operations, while (c) ExpAX requires only 1 annotation on the output.

tions and provides statistical guarantees that the error expectations will be satisfied. The optimization procedure is parameterized by a system specification that models error and energy characteristics of the underlying hardware. While the programmer specifies the error expectations in a high-level, hardware-independent manner, our optimization procedure automatically considers the low-level hardware parameters without exposing them to the programmer. ExpAX thereby enhances the portability of the approximate programs. We have implemented ExpAX for the full Java language and evaluate it on a diverse set of applications. The results show that ExpAX provides significant energy savings (up to 35%) with large reduction in programmer effort (between 3× to 113× compared to the state-of-the-art EnerJ [21] approximate programming language) while providing formal safety and statistical quality-of-result guarantees with high 95% confidence intervals.

$$
\begin{array}{rlrl}
(\textit{real constant}) & r \in \mathbb{R} & (\textit{real variable}) & v \in \mathbb{V} \\
(\textit{real expression}) & e \in \mathbb{R} \cup \mathbb{V} & (\textit{object variable}) & p \in \mathbb{P} \\
(\textit{variable}) & a \in \mathbb{V} \cup \mathbb{P} & (\textit{object field}) & f \in \mathbb{F} \\
(\textit{real operation}) & \delta \in \Delta & (\textit{allocation label}) & h \in \mathbb{H} \\
(\textit{operation label}) & l \in \mathbb{L} & (\textit{expectation label}) & k \in \mathbb{K}
\end{array}
$$

$$
\begin{array}{rrcl}
(\textit{error}) & c & \in & \mathbb{R}_{0,1} = [0, 1] \\
(\textit{error magnitude}) & \xi & \in & (\mathbb{R} \times \mathbb{R}) \to \mathbb{R}_{0,1} \\
(\textit{expectation}) & \phi & ::= & \mathsf{rate}(v) < c \\
 & & | & \mathsf{magnitude}(v) < c \text{ using } \xi \\
 & & | & \mathsf{magnitude}(v) > c \text{ using } \xi \text{ with rate} < c' \\
(\textit{statement}) & s & ::= & v := \delta^l(e_1, e_2) \mid \phi^k \\
 & & | & \mathsf{precise}(v) \mid \mathsf{accept}(v) \\
 & & | & p := \mathsf{new}\ h \mid p_1 := p_2 \\
 & & | & a := p.f \mid p.f := a \\
 & & | & \mathsf{assume}(v) \mid s_1; s_2 \mid s_1 + s_2 \mid s^*
\end{array}
$$

Figure 2: Language syntax.

$$
\begin{array}{rcccl}
(\textit{error model}) & \epsilon & \in & \mathbb{E} & = (\mathbb{R}_{0,1} \times \mathbb{R}) \\
(\textit{energy model}) & j & \in & \mathbb{J} & = (\mathbb{R}_{0,1} \times \mathbb{R}) \\
(\textit{system spec}) & \mathcal{S} & \in & & \Delta \to (\mathbb{E} \times \mathbb{J})
\end{array}
$$

Figure 3: System specification.

2. Expectation-Oriented Programming Model

This section presents our programming model that allows programmers to specify bounds on errors that are acceptable on program data in a system-independent manner, called *expectations*. In doing so, it frees programmers from specifying which program operations to approximate, and thereby from reasoning about the error models and energy models of the systems on which the program will run. We present the main features of our programming language (Section 2.1) and illustrate them on a real-world example (Section 2.2). We then describe *system specifications*, which provide the error models and energy models of approximable program operations on the underlying system (Section 2.3).

2.1 Language

We assume a simple imperative language whose abstract syntax is shown in Figure 2. It supports two kinds of program data, real numbers and object references, and provides standard constructs to create and manipulate object references. It also includes standard control-flow constructs for sequential composition, branches, and loops. We elide conditionals in branches and loops, executing them non-deterministically and instead using the $\mathsf{assume}(v)$ construct that halts execution if $v \leq 0$. For instance, the statement "if $(v > 0)$ then s" can be expressed as "$\mathsf{assume}(v)$; s" in our language. For brevity, we elide other data types (e.g., arrays and records) and procedure calls; they are supported in our implementation for the full Java language, presented in Section 5.

Statements $\mathsf{precise}(v)$ and $\mathsf{accept}(v)$ are generated automatically by our approximation framework and are not used directly by programmers; we describe them in Section 3. The two remaining kinds of statements that are intended for programmers, and that play a central role in our approximation framework, are *operations* δ and *expectations* ϕ. Operations are the only places in the program where approximations may occur, providing opportunities to save energy at the cost of introducing computation errors. We assume each operation has a unique label l. Conversely, expectations are the only places in the program where the programmer may specify acceptable bounds on such errors. We assume each expectation has a unique label k.

An expectation allows the programmer to express, at a certain program point, a bound on the error in the data value of a certain program variable. We allow three kinds of expectations that differ in the aspect of the error that they bound: the error *rate*, the error *magnitude*, or both. Appendix A presents the formal semantics of expectations. We describe them informally below:

- Expectation $\mathsf{rate}(v) < c$ states that the rate at which an error is incurred on variable v should be bounded by c. Specifically, suppose this expectation is executed n_2 times in an execution, and suppose the value of v each time this expectation is executed deviates from its precise value n_1 times. Then, the ratio n_1/n_2 should be bounded by c.
- Expectation $\mathsf{magnitude}(v) < c$ using ξ states that the normalized magnitude of the error incurred on variable v should be bounded by c. Unlike the error rate, which can be computed universally for all variables, the error magnitude is application-specific: each application domain may use a different metric for quantifying the magnitude of error, such as signal-to-noise ratio (SNR), root mean squared error, relative error, etc. Hence, this expectation asks the programmer to specify how to compute this metric, via a function ξ that takes two arguments—the potentially erroneous value of v and the precise value of v—and returns the normalized magnitude of that error.
- Expectation $\mathsf{magnitude}(v) > c$ using ξ with rate $< c'$ allows to bound both the error rate and the error magnitude: it states that the rate at which the error incurred on variable v exceeds normalized magnitude c is bounded by c'.

2.2 Example

We illustrate the above three types of expectations on the sobel benchmark shown in Figure 4 that performs image edge detection. The edge detection algorithm first converts the image to grayscale (lines 6–8). Then, it slides a 3×3 window over the pixels of the grayscale image and calculates the gradient of the window's center pixel to its eight neighboring pixels (lines 9–16). For brevity, we omit showing the body of the `build_window` function. Since the precise gradient calculation is compute intensive, image processing algorithms use a Sobel filter (lines 25–35), which gives an estimation of the gradient. Thus, the application is inherently approximate.

We envision a programmer specifying acceptable bounds on errors resulting from approximations in the edge detection application by means of three expectations indicated by the *accept* keyword in the figure. The first expectation is specified on the entirety of the `output_image` (line 17). It states that less than 0.1 (10%) magnitude of error (root-mean-squared difference of pixels of the precise and approximate output) is acceptable. The second expectation specifies that on less than 35% of the grayscale pixel conversions, the error magnitude (relative error) can exceed 0.9 (90%). The third expectation specifies that up to 25% of the times `gradient` is calculated, any amount of error is acceptable. These specifications capture the domain knowledge of the programmer about the application and their error expectations. Further, the specified expectations serve to implicitly identify any operations contributing to the computation of data that can be potentially approximated. In practice, only the first expectation specification on `output_image` (line 17) suffices for our approximation safety analysis (Section 3) to identify the approximable subset of operations. But our optimization framework (Section 4) can consume all three expectations, using them together to guide the optimization process that ultimately determines which approximable operations to approximate in order to provide statistical quality-of-result guarantees on a given system. We next describe the system specification that is needed for this purpose.

```
void detect_edges(Pixel[][] input_image,
2                  Pixel[][] output_image) {
  float[][] gray_image = new float[WIDTH][HEIGHT];
4 float[][] window = new float[3][3];
  float gradient;
6 for(int y = 0; y < HEIGHT; ++y)
    for(int x = 0; x < WIDTH; ++x)
8     gray_image[x][y] = to_grayscale(input_image[x][y]);
  for(int y = 0; y < HEIGHT; ++y)
10  for(int x = 0; x < WIDTH; ++x) {
      build_window(gray_image, x, y, window);
12    gradient = sobel(window);
      output_image[x][y].r = gradient;
14    output_image[x][y].g = gradient;
      output_image[x][y].b = gradient;
16  }
  accept magnitude[output_image] < 0.1;
18 }
float to_grayscale(Pixel p) {
20 float luminance;
  luminance = p.r * 0.30 + p.g * 0.59 + p.b * 0.11;
22 accept magnitude[luminance] > 0.9 with rate < 0.35;
  return luminance;
24 }
float sobel(float[][] p) {
26 float x, y, gradient;
  x =  (p[0][0] + 2 * p[0][1] + p[0][2]);
28 x += (p[2][0] + 2 * p[2][1] + p[2][2]);
  y =  (p[0][2] + 2 * p[1][2] + p[2][2]);
30 y += (p[0][0] + 2 * p[1][1] + p[2][0]);
  gradient = sqrt(x * x + y * y);
32 gradient = (gradient > 0.7070) ? 0.7070 : gradient;
  accept rate[gradient] < 0.25;
34 return gradient;
}
```

Figure 4: Expectation-oriented programming example.

2.3 System Specification

The system specification provides the error model and the energy model of the system on which our programs execute. Our approximation framework is parametric in the system specification to allow tuning the accuracy/energy tradeoff of a program in a manner that is optimal on the given system.

We adopt system specifications of the form shown in Figure 3. Such a specification $\mathcal{S}$ specifies an error model ϵ and an energy model j for each approximable operation. In our formalism, this is every operation $\delta \in \Delta$ on real-valued data. Error models and energy models are specified in our framework as follows:

- An error model ϵ for a given operation is a pair (c, r) such that c is the rate at which the operation, if run approximately, is expected to compute its result inaccurately; moreover, the magnitude of the error in this case is $\pm r$.
- An energy model j for a given operation is also a pair (c, r) such that r is the energy that the operation costs, if run precisely, while c is the fraction of energy that is saved if the same operation is run approximately.

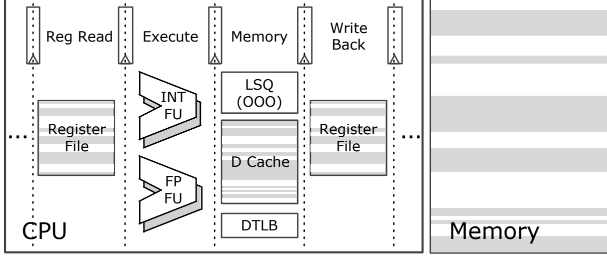


Figure 5: The architecture model resembles Truffle [9] that interleaves execution of precise and approximate instructions.

While the above simple abstract system specification is sufficient for the formalism, we develop a rich and comprehensive model for the implementations. We assume the architecture model shown in Figure 5 that resembles the dual-voltage Truffle [9] architecture that interleaves the execution of precise and approximate instructions. The ISA supports both approximate operations and approximate storage, and a bit in the opcode identifies whether the instruction is the approximate or the precise version. This ISA enables ExpAX to selectively choose which instructions to approximate to provide both formal safety and statistical quality-of-result guarantees. Writing and reading from approximate storage and performing approximate integer operations (e.g., approximate add) are performed with over-scaled low voltage that results in less energy dissipation but sometimes erroneous results. Main memory (DRAM) supports approximation by reduced refresh rate. Bit-width reduction is used for the floating-point functional units. As illustrated in gray in Figure 5, in our model, register files, data caches, and memory can store a mixture of approximate and precise data. The functional units can switch between approximate (low voltage) and precise (nominal voltage) mode. However, instruction fetch, decode, and commit are always precise. This model ensures that executing approximate instructions will not affect the precise state of the program. Our system specification incorporates the overhead of keeping these parts of the system precise. Based on these approximation techniques and the energy and error models in [21], we derive four system specifications shown in Table 1. While these simplified models provide good-enough energy and error estimates, ExpAX can easily replace these models with more accurate system specifications. In fact, a strength of ExpAX is that its modularity and system independence allow system specifications as plugins.

3. Approximation Safety Analysis

We develop a static analysis that automatically identifies the largest set of operations that are safe to approximate under a given set of programmer-specified expectations. We first define the concrete semantics of the approximation safety property and formulate the problem of finding the largest set of safe-to-approximate operations (Section 3.1). We then present a static analysis that checks whether approximating a given set of operations satisfies that property, and prove its soundness (Section 3.2). Finally, we describe how we use this analysis in an instantiation of a CEGAR framework [28] to efficiently compute the largest set of safe-to-approximate operations (Section 3.3).

3.1 The Approximation Safety Property

We regard three kinds of operations as unsafe to approximate: i) those that may cause memory safety violations, e.g., dereference null pointer or index arrays out of bounds; ii) those that may affect the control-flow, which in turn affects program termination and functional correctness (i.e., application-specific) properties of the program; and iii) those that may affect program outputs that are not explicitly specified by the programmer as approximable via expectations.

To uniformly identify operations that are unsafe to approximate, we introduce two constructs in our language, denoted precise(v) and accept(v). These constructs are automatically generated at relevant program points by the compiler. The precise(v) construct requires that variable v hold its precise (i.e., unapproximated) value. The accept(v) construct arises from expectations provided by the programmer: it only retains the variable v specified in the expectation and strips away the error bounds (rate and magnitude). Intuitively, accept(v) indicates that the programmer can tolerate an approximate value for variable v, and that v will be regarded as holding a precise value thereafter, as far as the approximation safety property is concerned.

Example. Consider the program on the left below which we will use as a running example in the rest of this section:

```
                        L={1,2,5,6}  L={2,5,6}    L={2,6}
 1: v1 := input();      {{v1}}       {{}}         {{}}
 2: v2 := input();      {{v1,v2}}    {{v2}}       {{v2}}
 3: precise(v1);        {T}          {{v2}}       {{v2}}
 4: while (v1 > 0) {    {T}          {{v2},T}     {{v2}}
 5:   v1 := f(v1);      {T}          {{v1,v2},T}  {{v2}}
 6:   v2 := g(v2);      {T}          {{v1,v2},T}  {{v2}}
 7:   precise(v1);      {T}          {T}          {{v2}}
 8: }                   {T}          {{v2},T}     {{v2}}
 9: accept(v2);         {T}          {{},T}       {{}}
10: precise(v2);        {T}          {{},T}       {{}}
11: output(v2);         {T}          {{},T}       {{}}
```

The precise(v1) construct is introduced on lines 3 and 7 to ensure that loop condition v1 $>$ 0 executes precisely. The precise(v2) construct on line 10 is introduced to ensure that the value of variable v2 output on line 11 is precise. The programmer specifies an expectation on v2 on line 9, denoted by the accept(v2) construct, which allows the operations writing to v2 on lines 2 and 6 to be approximated without violating the precise(v2) requirement on line 10. However, the operations writing to v1 on lines 1 and 5 cannot be approximated as they would violate the precise(v1) requirement on line 3 or 7, respectively. □

To formalize the approximation safety property, we define semantic domains (Figure 6) and a concrete semantics (Figure 7). Each program state ω (except for distinguished states error and halt described below) tracks a *tainted set*

Table 1: Summary of the approximation techniques and the four system specifications derived from [21].

Operation	Technique		Mild	Medium	High	Aggressive
Integer Arithmetic/Logic	Voltage Overscaling	Timing Error Probability	10^{-6}	10^{-4}	10^{-3}	10^{-2}
		Energy Saving	12%	22%	26%	30%
Floating Point Arithmetic	Bit-width Reduction	Mantissa Bits	16 *bits*	8 *bits*	6 *bits*	4 *bits*
		Energy Saving	32%	78%	82%	85%
Double Precision Arithmetic	Bit-width Reduction	Mantissa Bits	32 *bits*	16 *bits*	12 *bits*	8 *bits*
		Energy Saving	32%	78%	82%	85%
SRAM Read (Reg File/Data Cache)	Voltage Overscaling	Read Upset Probability	$10^{-16.7}$	$10^{-7.4}$	$10^{-5.2}$	10^{-3}
		Energy Saving	70%	80%	85%	90%
SRAM Write (Reg File/Data Cache)	Voltage Overscaling	Write Failure Probability	$10^{-5.6}$	$10^{-4.9}$	10^{-4}	10^{-3}
		Energy Saving	70%	80%	85%	90%
DRAM (Memory)	Reduced Refresh Rate	Per-Second Bit Flip Probability	10^{-9}	10^{-5}	10^{-4}	10^{-3}
		Memory Power Saving	17%	22%	23%	24%

T that consists of real-valued memory locations, including variables v and heap locations (o, f) denoting field f of runtime object o. A location gets tainted if its value is affected by some operation to be approximated, and it gets untainted if an expectation is executed on it (recall that we regard such a location as holding a precise value thereafter). Each rule of the semantics is of the form:

$$L \models \langle s, \rho_1, \sigma_1, T_1 \rangle \rightsquigarrow \langle \rho_2, \sigma_2, T_2 \rangle \mid \mathsf{halt} \mid \mathsf{error}$$

It describes an execution of program s with set L of operations to be approximated, starting with environment (i.e., valuation to variables) ρ_1 and heap σ_1. The rules are similar to those in information flow tracking: approximated operations in L are *sources* (rule ASGN), $\mathsf{precise}(v)$ constructs are *sinks* (rules VARPASS and VARFAIL), and $\mathsf{accept}(v)$ constructs are *sanitizers* (rule ACCEPT). The execution ends in state error if some $\mathsf{precise}(v)$ construct is executed when the tainted set contains v, as captured by rule VARFAIL. The execution may also end in state halt, which is normal and occurs when $\mathsf{assume}(v)$ fails (i.e., $v \leq 0$), as described by rules ASMPASS and ASMFAIL. Lastly, rules FIELDRD and FIELDWR describe how the tainted set is manipulated by reads and writes of real-valued fields. We omit the rules for pointer-manipulating operations (e.g., reads and writes of object-valued fields), those for compound statements, and those that propagate error and halt, as they are relatively standard and do not affect the tainted set.

We now state the approximation safety property formally:

DEFINITION 3.1. ***(Approximation Safety Property)*** *A set of operations L in a program s is safe to approximate if for every ρ and σ, we have $L \models \langle s, \rho, \sigma, \emptyset \rangle \not\rightsquigarrow \mathsf{error}$.*

To maximize the opportunities for energy savings, we seek to find as large a set of operations that is safe to approximate. In fact, there exists a unique largest set of such operations:

PROPOSITION 3.2. *If sets of operations L_1 and L_2 in a program are safe to approximate, then set of operations $L_1 \cup L_2$ is also safe to approximate.*

Example. For our running example, the largest set of safe-to-approximate operations comprises those on lines 2 and 6. The column headed `L={2,6}` on the right of the example shows the tainted set computed by our semantics after each statement under this set of approximated operations. The error state is not reachable in this execution as the tainted set just before executing each $\mathsf{precise}(v)$ does not contain v. Hence, this set of operations is safe to approximate. □

$$\begin{array}{rrcl}
(\textit{object}) & o & \in & \mathbb{O} \\
(\textit{environment}) & \rho & \in & (\mathbb{V} \to \mathbb{R}) \cup (\mathbb{P} \to \mathbb{O}) \\
(\textit{heap}) & \sigma & \in & (\mathbb{O} \times \mathbb{F}) \to (\mathbb{O} \cup \mathbb{R}) \\
(\textit{tainted set}) & T & \subseteq & \mathbb{V} \cup (\mathbb{O} \times \mathbb{F}) \\
(\textit{program state}) & \omega & ::= & \langle s, \rho, \sigma, T \rangle \mid \langle \rho, \sigma, T \rangle \mid \mathsf{error} \mid \mathsf{halt}
\end{array}$$

Figure 6: Semantic domains.

$$\begin{array}{rrcl}
(\textit{abstract tainted set}) & \pi & \in & \Pi = 2^{\mathbb{V} \cup (\mathbb{H} \times \mathbb{F})} \\
(\textit{abstract state}) & D & \subseteq & \mathbb{D} = \Pi \cup \{\top\} \\
(\textit{points-to analysis}) & \mathsf{pts} & \in & \mathbb{P} \to 2^{\mathbb{H}}
\end{array}$$

$$\begin{array}{rcl}
F_L[s] & : & 2^{\mathbb{D}} \to 2^{\mathbb{D}} \\
F_L[s_1 \,;\, s_2](D) & = & (F_L[s_1] \circ F_L[s_2])(D) \\
F_L[s_1 + s_2](D) & = & F_L[s_1](D) \cup F_L[s_2](D) \\
F_L[s^*](D) & = & \mathsf{leastFix}\ \lambda D'.(D \cup F_L[s](D')) \\
F_L[t](D) & = & \{\, \mathsf{trans}_L[t](d) \mid d \in D \,\}
\end{array}$$

for atomic statement t, where:

$$\mathsf{trans}_L[t](\top) = \top$$

$$\mathsf{trans}_L[v :=^l \delta(e_1, e_2)](\pi) = \begin{cases} \pi \cup \{v\} & \text{if } l \in L \lor uses(e_1, e_2) \cap \pi \neq \emptyset \\ \pi \setminus \{v\} & \text{otherwise} \end{cases}$$

$$\mathsf{trans}_L[\mathsf{precise}(v)](\pi) = \begin{cases} \pi & \text{if } v \notin \pi \\ \top & \text{otherwise} \end{cases}$$

$$\mathsf{trans}_L[\mathsf{accept}(v)](\pi) = \pi \setminus \{v\}$$

$$\mathsf{trans}_L[v := p.f](\pi) = \begin{cases} \pi \cup \{v\} & \text{if } \exists h \in \mathsf{pts}(p) : (h, f) \in \pi \\ \pi \setminus \{v\} & \text{otherwise} \end{cases}$$

$$\mathsf{trans}_L[p.f := v](\pi) = \begin{cases} \pi \cup \{(h, f) \mid h \in \mathsf{pts}(p)\} & \text{if } v \in \pi \\ \pi & \text{otherwise} \end{cases}$$

Figure 8: Approximation safety analysis. Transfer functions of pointer operations are elided as they are no-ops; the effect of these operations is instead captured by points-to analysis pts.

3.2 Approximation Safety Checker

The approximation safety property is undecidable. In this section, we present a static analysis that conservatively determines whether a given set of operations is safe to approximate. The analysis is defined in Figure 8. It over-approximates the tainted sets that may arise at a program point by an abstract state D, which is a set each of whose

$$L \models \langle v :=^{l} \delta(e_1,e_2), \rho, \sigma, T\rangle \rightsquigarrow \langle \rho[v \mapsto [\![\delta(e_1,e_2)]\!](\rho)], \sigma, T'\rangle \text{ where } T' = \begin{cases} T \cup \{v\} & \text{if } l \in L \text{ or } uses(e_1,e_2) \cap T \neq \emptyset \\ T \setminus \{v\} & \text{otherwise} \end{cases} \quad \text{(ASGN)}$$

$$L \models \langle \mathsf{accept}(v), \rho, \sigma, T\rangle \rightsquigarrow \langle \rho, \sigma, T \setminus \{v\}\rangle \quad \text{(ACCEPT)}$$

$$L \models \langle \mathsf{precise}(v), \rho, \sigma, T\rangle \rightsquigarrow \langle \rho, \sigma, T\rangle \ [\text{if } v \notin T] \ \text{(VARPASS)} \qquad L \models \langle \mathsf{assume}(v), \rho, \sigma, T\rangle \rightsquigarrow \langle \rho, \sigma, T\rangle \ [\text{if } \rho(v) > 0] \ \text{(ASMPASS)}$$

$$L \models \langle \mathsf{precise}(v), \rho, \sigma, T\rangle \rightsquigarrow \mathsf{error} \quad [\text{if } v \in T] \ \text{(VARFAIL)} \qquad L \models \langle \mathsf{assume}(v), \rho, \sigma, T\rangle \rightsquigarrow \mathsf{halt} \quad [\text{if } \rho(v) \leq 0] \ \text{(ASMFAIL)}$$

$$L \models \langle v := p.f, \rho, \sigma, T\rangle \rightsquigarrow \langle \rho[v \mapsto r], \sigma, T'\rangle \ [\text{if } \rho(p) = o \wedge \sigma(o,f) = r] \text{ where } T' = \begin{cases} T \cup \{v\} & \text{if } (o,f) \in T \\ T \setminus \{v\} & \text{otherwise} \end{cases} \quad \text{(FIELDRD)}$$

$$L \models \langle p.f := v, \rho, \sigma, T\rangle \rightsquigarrow \langle \rho, \sigma[(o,f) \mapsto r], T'\rangle \ [\text{if } \rho(p) = o \wedge \rho(v) = r] \text{ where } T' = \begin{cases} T \cup \{(o,f)\} & \text{if } v \in T \\ T \setminus \{(o,f)\} & \text{otherwise} \end{cases} \quad \text{(FIELDWR)}$$

Figure 7: Semantics of approximation safety. Rules for compound statements and pointer operations are not shown for brevity.

elements is either $\top$ or an *abstract tainted set* π containing abstract locations. Each abstract location is either a variable v or an abstract heap location (h, f), denoting field f of an object allocated at the site labeled h. For precision, the analysis is disjunctive in that it does not merge different abstract tainted sets that arise at a program point along different program paths. Finally, it uses an off-the-shelf points-to analysis, denoted by function pts, to conservatively capture the effect of heap reads and writes on the abstract state.

Each transfer function of the analysis is of the form $F_L[s](D) = D'$, denoting that under set L of operations to be approximated, the program s transforms abstract state D into abstract state D'. The $\top$ element arises in D' either if it already occurs in D or if s contains a $\mathsf{precise}(v)$ statement and an abstract tainted set incoming into that statement contains the variable v. The $\top$ element thus indicates a potential violation of the approximation safety property. In particular, a program does not violate the property if the analysis determines that, starting from input abstract state $\{\emptyset\}$, the output abstract state does not contain $\top$:

THEOREM 3.3. ***(Soundness)*** *If* $\top \notin F_L[s](\{\emptyset\})$ *then* $L \models \langle s, \rho, \sigma, \emptyset\rangle \not\rightsquigarrow \mathsf{error}$.

Example. For our running example, the columns on the right show the abstract state computed by the analysis after each statement, using initial abstract state $\{\emptyset\}$, under the set of approximated operations indicated by the column header. For `L={1,2,5,6}`, the final abstract state contains $\top$, and indeed it is unsafe to approximate the operations on lines 1 and 5. On the other hand, for `L={2,6}`, the final abstract state does not contain $\top$, proving that it is safe to approximate the operations on lines 2 and 6. □

3.3 Inferring Largest Set of Approximable Operations

The analysis in the preceding section merely checks whether a *given* set of operations is safe to approximate; it does not infer the largest set of such operations that we seek. We employ a technique by Zhang et al. [28] to infer this set. Given a parameterized dataflow analysis $\mathcal{A}$ and a set of possible parameter values with an arbitrary preorder $\langle \Lambda, \preceq\rangle$ such that $\mathcal{A}_\lambda[s] \in \{0, 1\}$ for each $\lambda \in \Lambda$ and a given program s, the technique efficiently finds an optimum (largest) parameter value, i.e., a $\lambda \in \Lambda$ such that $\mathcal{A}_\lambda[s] = 1$ and $\lambda \preceq \lambda' \Rightarrow \mathcal{A}(\lambda') = 0$.

In our setting, analysis $\mathcal{A}$ is the function F defined in the preceding section (Figure 8). The parameter value is the set of operations $L \subseteq \mathbb{L}$ to be approximated, the space of possible parameter values is $\Lambda = 2^{\mathbb{L}}$, and the partial order $\preceq$ is defined as $L_1 \preceq L_2$ whenever $|L_1| \leq |L_2|$. Finally, $\mathcal{A}_L[s] = 1$ iff $\top \notin F_L[s](\{\emptyset\})$. Thus, the optimum parameter value corresponds to the largest set of operations that the analysis considers as safe to approximate.

The technique in [28] is based on counterexample-guided abstraction refinement (CEGAR). It runs analysis $\mathcal{A}$ on the given program s with a different parameter value λ in each iteration, starting with the largest one. When the analysis fails in a given iteration, that is, $\mathcal{A}_\lambda[s] = 0$, the technique applies a meta-analysis on an abstract counterexample trace to generalize the failure of the current parameter value λ and eliminate a set of parameter values under which the analysis is guaranteed to suffer a similar failure; then, in the subsequent iteration, the technique picks the largest of the remaining (i.e., so far uneliminated) parameter values according to the $\preceq$ preorder, and repeats the process. The process is guaranteed to terminate in our setting since the empty set of operations is always safe to approximate.

To analyze an abstract counterexample trace and generalize its failure to other parameter values, the technique requires a backward meta-analysis that is sound with respect to the abstract semantics of the forward analysis. We provide such a meta-analysis in Appendix B.

Example. We illustrate the above technique on our running example. The CEGAR framework starts by treating all operations approximable (L = `{1,2,5,6}`) and runs the forward analysis. The abstract state of the forward analysis becomes $\{\top\}$ after statement $\mathsf{precise}($`v1`$)$ on line 3. The analysis thus fails to prove that this set of operations is safe to approximate, and generates a counterexample comprising abstract states along the trace $\langle$`1,2,3,4,8,9,10,11`$\rangle$.

The backward meta-analysis starts from the end of the counterexample trace and generalizes the reason of failure to be `1` $\in L$, that is, the forward analysis will fail as long as the operation on line 1 is treated as approximable. Therefore, in the subsequent iteration, the CEGAR framework chooses L = `{2,5,6}` as the set of approximable operations, and reruns the forward analysis. The analysis fails again but this time generates a counterexample comprising abstract states along the trace $\langle$`1,2,3,4,5,6,7,8,9,10,11`$\rangle$. This time, the backward meta-analysis generalizes the reason of failure to be `5` $\in L$. In the third iteration, the framework chooses L = `{2,6}`, and the forward analysis succeeds, thus concluding that the largest set of safe-to-approximate operations comprises those on lines 2 and 6, as desired. □

4. Selecting Approximate Operations

After finding the safe-to-approximate operations, the next step is to automatically identify a subset of these operations that statistically satisfy the programmer-specified expectations. We take a pragmatic approach and develop optimization algorithms that perform the selection with respect to given input datasets. We randomly divide the inputs into two equally-sized subsets, which are training and validation inputs. The selection algorithm approximates as many of the operations as possible while minimizing the energy consumption and satisfying all the programmer-specified expectations on the training data. The validation subset is used to provide a confidence level on meeting the expectations when running on unseen data. We will provide 95% confidence intervals on the statistical guarantees in Section 5.

Our heuristic selection algorithm constitutes two phases, (1) genetic filtering and (2) greedy refining. In the genetic filtering phase, we develop a genetic algorithm to filter out the safe-to-approximate operations that if approximated will lead to significant quality degradation. Since genetic algorithms are *guided* random algorithms, the genetic filtering phase enables the selection procedure to explore the space of approximate versions of the program and significantly improve the quality of results. However, the genetic algorithm can only provide acceptable statistical guarantees if run for a large number of generations, which is intractable. We only run the genetic algorithm for a few generations and then switch to the greedy refining phase. The greedy refining phase starts from the result of the genetic filtering and greedily refines the solution until it satisfies the programmer expectations. Below we describe the two phases.

4.1 Genetic Filtering

Genetic algorithms iteratively explore the solution space of a problem and find a best-fitting solution that maximizes an objective. The first step in developing a genetic algorithm is defining an encoding for the possible solutions, namely phenotypes. Then, the algorithm randomly populates the first generation of the candidate solutions. The algorithm assigns scores to each individual solution based on a fitness function.

Algorithm 1 Genetic filtering.

```
1:  INPUT: Program Π = ⟨s, 𝓛, 𝕂⟩:
        - s is program body
        - 𝓛 is set of safe-to-approximate operations in s
        - 𝕂 is set of all expectations in s
2:  INPUT: Program input datasets 𝒟 = {ρ_1, ..., ρ_n}
3:  INPUT: System specification 𝒮
4:  INPUT: Genetic algorithm parameters ⟨N, M, P, α, β⟩:
        - N is population size           - M is number of generations
        - P is probability of mutation   - α := 0.5, β := 0.5
5:  OUTPUT: elite_phenotype Θ: Bit-vector with the max global fitness
6:  OUTPUT: contribution Γ: Vector with estimated contribution of each
    operation to error and energy
7:  for i in 1..N do
8:      phenotype[i] := random bit vector of size |𝓛|
9:  end for
10: t = Vector of size |𝓛| with values set to 0
11: contribution = Vector of size |𝓛| with values set to 0.0
12: for m in 1..M do
13:     var f, g of type ℝ_{0,1}[N]
14:     for [parallel] i in 1..N do
15:         error[i], energy[i] := execute(Π, phenotype[i], 𝒟, 𝒮)
16:         f[i] := (α × error[i] + β × energy[i])^{-1}
17:         for j in 1..|phenotype[i]| do
18:             c[j] := (1/f[i]) / (Σ_k phenotype[i]_k)
19:             if phenotype[i]_j == 1 then
20:                 contribution[j] += 1/(t[j]+1) (c[j] - contribution[j])
21:                 t[j] := t[j] + 1
22:             end if
23:         end for
24:     end for
25:     for i in 1..N do
26:         g[i] := f[i]/ Σ f[i]
27:         if f[i] > f_elite_phenotype then
28:             f_elite_phenotype := f[i]
29:             elite_phenotype := phenotype[i]
30:         end if
31:     end for
32:     for i in 1..N do
33:         x, y := roulette_wheel(g)
34:         phenotype[x], phenotype[y] :=
35:             crossover(phenotype[x], phenotype[y])
36:         phenotype[x] := mutate(phenotype[x], P)
37:         phenotype[y] := mutate(phenotype[y], P)
38:     end for
39: end for
```

The fitness function is derived from the optimization objective. Genetic algorithms includes two operators, crossover and mutation. These operators pseudo randomly generate the next generation of the solutions based on the scores of the current generation. We describe all the components of our genetic algorithm, presented in Algorithm 1.

Phenotype encoding. In our framework, a solution is a program in which a subset of the operations is marked approximate. The approximation safety analysis provides a candidate subset of operations that are safe to approximate. We assign a vector to this subset whose elements indicate whether the corresponding operation is approximate (`'1'`) or precise (`'0'`). For simplicity, in this paper, we only assume a binary state for each operation. However, a generalization of our approach can assign more than two levels to each element, allowing multiple levels of approximation for

each operation. The bit-vector is the template for encoding phenotypes and its bit pattern is used to generate an approximate version of the program. We associate the bit-vector with the subset resulting from the approximation safety analysis and not the entire set of operations, to avoid generating unsafe solutions.

Fitness function. The fitness function assigns scores to each phenotype. The higher the score, the higher the chance that the phenotype is selected for producing the next generation. The fitness function encapsulates the optimization objective and guides the genetic algorithm in producing better solutions. Our fitness function is defined as follows:

$$f(phenotype) = (\alpha \; error + \beta \; energy)^{-1} \tag{1}$$

Based on (1), a phenotype with less $error$ and less $energy$ has a higher score and higher chance of reproduction. In our optimization framework, error and energy are computed with respect to given program input datasets ($\mathcal{D} = \{\rho_1, ..., \rho_n\}$). The $energy$ term in the formulation is the energy dissipation when a subset of the approximable operations are approximated, normalized to the energy dissipation of the program when executed on the same input dataset with no approximation. The combination of $energy$ and $error$ in the fitness function strikes a balance between saving energy and preserving the quality-of-results, while considering the programmer expectations. The $error$ term is the normalized sum of error (rate or magnitude) at the site of all expectations when the program runs on the ρ_i dataset. We incorporate the expectations in the optimization through $error$. If the observed error is less than the expectation bound, the genetic algorithm assumes error on an expectation site is 0. Otherwise, the genetic algorithm assumes the error is the difference of the observed error and the expectation bound. Intuitively, when the observed error is less than the specified bound, the output is acceptable, same as the case where error is zero. The optimization objective in the genetic algorithm is to push the error below the specified expectation bound. The genetic algorithm equally weights the error and energy in the fitness function ($\alpha = \beta = 0.5$). The genetic algorithm generates approximate versions of the program based on the phenotypes and runs them with the input datasets. The algorithm makes each version of the program an independent thread and runs them in parallel. Executing different input datasets can also be parallel.

Selection strategy and operators. We use the common roulette wheel selection strategy and crossover and mutation operators for generating the next generations[1]. The algorithm records the best global solution across all the generations, a technique called elitism.

Estimating contributions. The genetic filtering phase executes the program several times and measures its energy and error under different selection patterns. We use this information to estimate a contribution score for each of the operations based on the final error and energy of the program.

[1] See http://en.wikipedia.org/wiki/Genetic_algorithm.

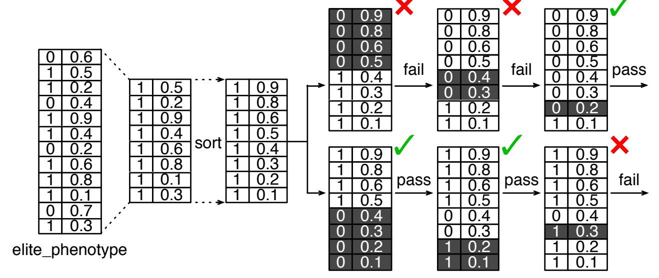


Figure 9: An example of the greedy refining algorithm.

For simplicity, we assume that all the operations contribute equally to the final error and energy.

More sophisticated estimations that require profiling may include the number of times an operation is executed. As (2) shows, if an operation is approximate, its contribution score is ($\alpha \; error + \beta \; energy$) divided by the number of approximated operations. The final contribution score is calculated as the average across all the phenotypes generated during the genetic algorithm. The higher the score, the higher the final error and final energy consumption.

$$c(operation_i) = \begin{cases} 0 & \text{if } phenotype_i = 0 \\ \frac{(\alpha \; error + \beta \; energy)}{\sum_k phenotype_k} & \text{otherwise} \end{cases} \tag{2}$$

Estimating the real contribution of the operations requires an exponential number of runs, on the order of $2^{|phenotype|}$. Our simple estimation score tries to reuse the information attained during the genetic filtering to guide the next phase of selection algorithm, greedy refining.

4.2 Greedy Refining

To provide statistical guarantees that the approximate version of the program satisfies the expectations, we develop a greedy algorithm that refines the output of the genetic filtering (see Algorithm 2). As Figure 9 illustrates, the greedy refining algorithm extracts the list of approximated operations from the output of the genetic filtering and sorts those operations based on their contribution score. The algorithm then divides the sorted operation into two (high-score, low-score) subsets. A higher score for an operation indicates lower possibility of energy benefits and more error when the operation is approximated. The algorithm iteratively explores two paths. The first path first excludes the high-score subset and checks whether the resulting approximated version of the program satisfies the expectations. On the other hand, the second path starts by excluding the low-score subset. The rest of the algorithm is similar for both paths.

If the resulting smaller subset of the operations satisfies the expectations, the algorithm adds back the low-score half of the excluded operations in the next iteration. If approximating the smaller subset fails to satisfy the expectations, in the next iteration, the algorithm removes the high-score half of the approximated operations. The algorithm repeats these steps until it either cannot generate a different version or it reaches a predetermined termination depth for exploring new versions. The algorithm records all the subsets that

Algorithm 2 Greedy refining.

1: **INPUT:** Program $\Pi = \langle s, \mathcal{L}, \mathbb{K} \rangle$:
- s is program body
- $\mathcal{L}$ is set of safe-to-approximate operations in s
- $\mathbb{K}$ is set of all expectations in s

2: **INPUT:** Program input datasets $\mathcal{D} = \{\rho_1, ..., \rho_n\}$
3: **INPUT:** System specification $\mathcal{S}$
4: **INPUT:** Greedy refining parameters $\langle \Theta, \Gamma, d, t \rangle$:
- elite_phenotype Θ
- Contributions, estimated during genetic filtering $\Gamma = \{\gamma_1, ..., \gamma_{|\Theta|}\}$
- d is current depth of greedy refining - t is termination depth

5: **OUTPUT:** Bit-vector satisfying the error expectation with maximum energy saving $<\Omega, \text{energy}_\Omega>$
6: **if** $d = 0$ **then**
7: error$_\Theta$, energy$_\Theta$:= execute(Π, Θ, $\mathcal{D}$, $\mathcal{S}$)
8: **if** satisfy(error$_\Theta$, $\mathbb{K}$) **then**
9: **return** $<\Theta, \text{energy}_\Theta>$
10: **else**
11: sort(Θ, Γ)
12: Θ_{high}, Θ_{low} := split(Θ, Γ)
13: Ω_{high}, energy$_{\Omega_{high}}$:= greedy_refining($\Theta_{high}, \Gamma, 1, t$)
14: Ω_{low}, energy$_{\Omega_{low}}$:= greedy_refining($\Theta_{low}, \Gamma, 1, t$)
15: **if** energy$_{\Omega_{high}}$ > energy$_{\Omega_{low}}$ **then**
16: **return** $<\Omega_{high}, \text{energy}_{\Omega_{high}}>$
17: **else**
18: **return** $<\Omega_{low}, \text{energy}_{\Omega_{low}}>$
19: **end if**
20: **end if**
21: **else if** $d \neq t$ **then**
22: error$_\Theta$, energy$_\Theta$:= execute(Π, Θ, $\mathcal{D}$, $\mathcal{S}$)
23: **if** satisfy(error$_\Theta$, $\mathbb{K}$) **then**
24: Θ_{d+1} := include_the_excluded_low_half(Θ, Γ)
25: Ω_{d+1}, energy$_{\Omega_{d+1}}$:= greedy_refining($\Theta_{d+1}, \Gamma, d+1, t$)
26: **if** energy$_\Theta$ > energy$_{\Omega_{d+1}}$ **then**
27: **return** $<\Theta, \text{energy}_\Theta>$
28: **else**
29: **return** $<\Omega_{d+1}, \text{energy}_{\Omega_{d+1}}>$
30: **end if**
31: **else**
32: Θ_{d+1} := exclude_the_included_high_half(Θ, Γ)
33: Ω_{d+1}, energy$_{\Omega_{d+1}}$:= greedy_refining($\Theta_{d+1}, \Gamma, d+1, t$)
34: **return** $<\Omega_{d+1}, \text{energy}_{\Omega_{d+1}}>$
35: **end if**
36: **else**
37: error$_\Theta$, energy$_\Theta$:= execute(Π, Θ, $\mathcal{D}$, $\mathcal{S}$)
38: **if** satisfy(error$_\Theta$, $\mathbb{K}$) **then**
39: **return** $<\Theta, \text{energy}_\Theta>$
40: **else**
41: **return** $<\phi, 0>$
42: **end if**
43: **end if**

satisfy the expectations and at the end returns the one that provides the highest energy savings. We emphasize that the program only satisfies the expectation if the error is below the specified expectations while running on all the training input datasets. We will use the validation datasets to provide confidence intervals on the statistical guarantees.

5. Evaluation

We have implemented ExpAX for Java bytecode programs.

5.1 Methodology

Benchmarks. We evaluate ExpAX on eight Java programs from the EnerJ benchmark suite [21] plus sobel (Table 2). Five of these benchmarks come from the SciMark2 suite. The rest include zxing, a multi-format bar code recognizer developed for Android phones; jmeint, algorithm for detecting 3D triangles intersection (part of jMonkeyEngine game development kit); sobel, an edge detection application based on the Sobel operator; and raytracer, a simple 3D ray tracer.

Quality metrics. Table 2 also shows the application-specific quality metrics. For jmeint and zxing, the quality metric is a rate. For jmeint, the quality metric is the rate of correct intersection detections. Similarly, for the zxing bar code recognizer, the quality metric is the rate of successful decodings of QR code images. For the rest of the benchmarks, the quality metric is defined on the magnitude of error, which is calculated based on an *application-specific* quality-of-result metric. For most of the applications, the metric is the root-mean-squared difference of the output vector, matrix, or pixel array from the precise output.

Input datasets. We use 20 distinct input data sets that are either typical inputs (e.g., the baboon image) or randomly generated. Ten of these inputs are used as training data for the approximate operation selection heuristic. The other ten are solely used for validation and calculating the confidence level on the statistical quality-of-result guarantees.

Genetic filtering. As mentioned, we use the genetic filtering to exclude the safe-to-approximate operations that if approximated lead to significant quality-of-result degradation. We use a fixed-size population of 30 phenotypes across all the generations. We run the genetic algorithm for 10 generations. We use a low probability of 0.02 for mutation. Using low probability for mutation is a common practice in using genetic algorithms and prevents the genetic optimization from random oscillations. Running the genetic phase for more generations and/or with larger populations can only lead to potentially better solutions. However, our current setup provides a good starting point for the greedy refining phase of the selection algorithm. We have made our genetic algorithm parallel. The programs generated based on phenotypes of a generation run as independent threads in parallel.

Tools. We use Chord [17] for the approximation safety analysis that finds the safe-to-approximate operations. To select the subset of the safe-to-approximate operations that statistically satisfy the programmer expectation, we modified the EnerJ open-source simulator [21] for energy and error measurements and built our approximate operation selection algorithm on top of it. The simulator allows the instrumentation of method calls, object creation and destruction, arithmetic operations, and memory accesses to collect statistics and inject errors based on the system specifications. The runtime system of the simulator, which is a Java library and is invoked by the instrumentation, records memory-footprint and arithmetic-operation statistics while simultaneously injecting error to emulate approximate execution. The modified simulator at the end records the error rate or error magnitude at the expectation sites. The simulator also calculates

Table 2: Benchmarks, quality metric (Mag=Magnitude), programmer effort, and result of approximation safety analysis.

	Description	Quality Metric	# Lines	# of Annotations (Programmer Effort)				# of Safe-to-Approximate Operations
				EnerJ	ExpAX			
					Intra-Proc Analysis	After If-Conversion	Inter-Proc Analysis	
fft	SciMark2 benchmark: scientific kernels	Mag: Avg entry diff	168	20	6	6	1	133
sor		Mag: Avg entry diff	36	9	3	3	1	23
mc		Mag: Normalized diff	59	3	3	2	1	11
smm		Mag: Avg normalized diff	38	8	3	3	1	8
lu		Mag: Avg entry diff	283	27	11	8	1	53
zxing	Bar code decoder for mobile phones	Rate of incorrect reads	26171	192	109	95	15	902
jmeint	jMonkeyEngine game: triangle intersection kernel	Rate of incorrect decisions	5962	113	40	26	1	1513
sobel	Image edge detection	Mag: Avg pixel diff	161	23	9	9	1	127
raytracer	3D image renderer	Mag: Avg pixel diff	174	27	8	5	1	321

the energy savings associated with the approximate version of the program, which is generated based on phenotypes of the genetic filtering or the bit-vectors of the greedy refining. We run each application ten times to compensate for the randomness of the error injection and average the results. The results from the simulation are fed back to the selection procedure as it searches for the approximate version of the program that statistically satisfies the error expectations.

5.2 Experimental Results

Programmer effort and approximation safety analysis. To assess the reduced programmer effort with ExpAX, we compare the number of EnerJ [21] annotations with the ExpAX annotations that lead to the same number of safe-to-approximate operations. EnerJ requires the programmer to explicitly annotate approximate variables using approximate type qualifiers. In Table 2, the *EnerJ* column shows the required number of annotations with the EnerJ type qualifiers. We present the number of ExpAX annotations in three steps:

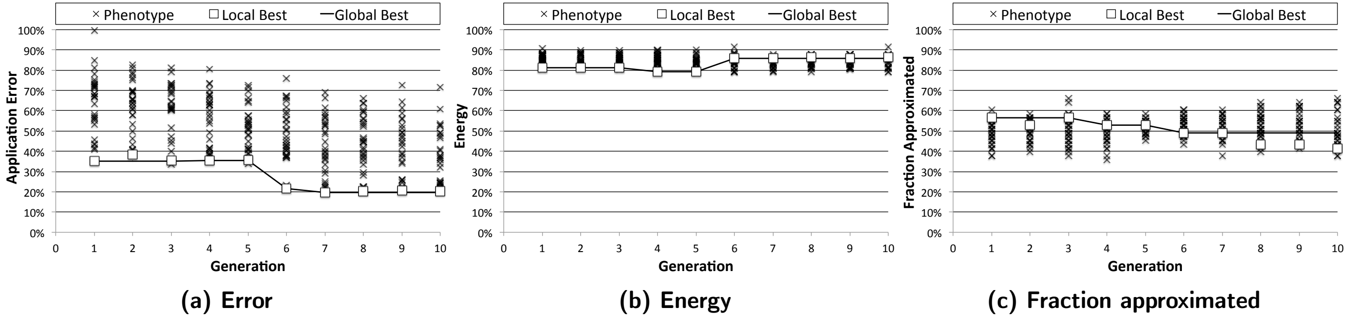
1. Adding annotations to each function separately (in Table 2, the *Intra-Proc Analysis* column). In this step, we add ExpAX annotations to each function while assuming no inter-procedural analysis. Even in this case, ExpAX requires significantly less annotations than EnerJ.
2. Using if-conversion to reduce the number of conditionals and consequently reducing annotations (in Table 2, the *If-Conversion* column). EnerJ and ExpAX do not allow conditionals to be approximate unless the programmer explicitly endorses approximating them. We use if-conversion to replace control dependence with data dependence when the conditional does not impact the program execution path. If-conversion removes the need for explicitly annotating many conditionals. For example, jmeint is a control-heavy benchmark with many if-then-else statements that only assign different values to variables. Applying if-conversion to those if-then-else statements reduces the number of annotations from 40 to 26.
3. Performing the complete inter-procedural approximation safety analysis and reducing the number of ExpAX annotations to minimum (in Table 2, the *Inter-Proc Analysis* column). In most cases, just one annotation coupled with our inter-procedural analysis results in the same number of safe-to-approximate operations that the programmer identifies using manual EnerJ annotations. Below, we discuss why zxing requires more annotations.

The last column shows the number of safe-to-approximate operations that are identified by our safety analysis. These operations match the operations that are identified by manual annotations with EnerJ. As presented, our safety analysis requires only one annotation on the final approximate output of most of the applications (the *Inter-Proc Analysis* column), while EnerJ requires far more annotations. *The results in Table 2 show between 3× (for mc) to 113× (for jmeint) reduction in programmer effort when comparing our automatic safety analysis to EnerJ's manual annotations.*

The largest benchmark in our suite, zxing, requires 15 annotations for the following two reasons:

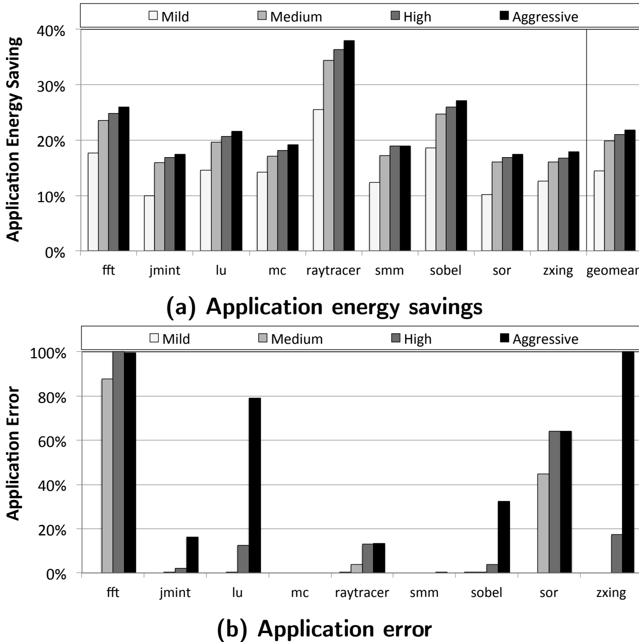
1. **Approximating conditionals that affect program path.** Several conditionals that control the break or continuations of loops are endorsed for approximation with EnerJ annotation. These conditionals cannot be if-converted, therefore, we add ExpAX annotations to match EnerJ's safe-to-approximate operations.
2. **Reading and writing intermediate results to files.** The zxing benchmark decodes QR barcodes which is implemented in several stages. In many cases these stages communicate by writing and reading to files instead of using program variables to pass values. If only one expectation is specified on the final output of this benchmark, the approximation safety analysis will not be able to track the dependence between the stages that communicate through files. Thus, ExpAX needs more annotations to identify the safe-to-approximate operations in each stage.

Approximating all safe-to-approximate operations. Figure 10 shows the energy and error when approximating all the safe-to-approximate operations. The error is averaged across the ten training inputs. As depicted, the geometric mean of energy savings ranges from 14% with the Mild system specification to 22% with the Aggressive system specification. The sor shows the least energy savings (10% with Mild) while raytracer shows the highest energy savings (38% with Aggressive). All the applications show acceptable error levels with the Mild system. However, in most cases, there is a jump in error when the system specification changes to High. These results show the upper bound on en-



(a) Error (b) Energy (c) Fraction approximated

Figure 11: The lu benchmark under the genetic filtering with the Aggressive system specification.



(a) Application energy savings

(b) Application error

Figure 10: (a) Energy savings and (b) error when approximating all the identified safe-to-approximate operations.

ergy savings. The results in Figure 10 show that our approximation safety analysis identifies the approximable subset of the program with few high-level annotations, which leads to significant energy savings. *Compared to the state-of-the art approximate programming models, EnerJ [21] and Rely [4], that require programmers to annotate all the data and/or operation as approximate, ExpAX offers large reduction in programmer effort while providing significant energy savings.*

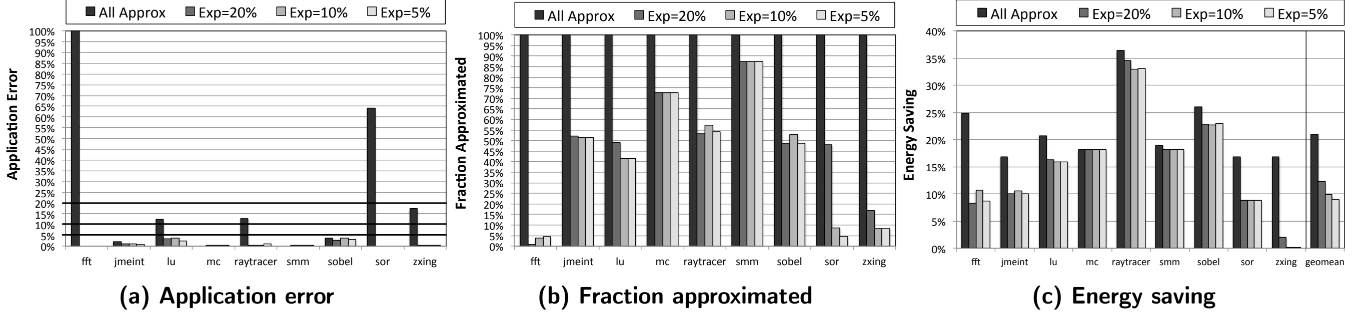
Genetic filtering. To understand the effectiveness of the genetic filtering stage in the selection algorithm, we take a close look at lu when it undergoes the genetic filtering with the Aggressive system specification. Figure 11a depicts the distribution of error for lu in each generation of the genetic filtering. As shown, the application shows a wide spectrum of error, 19%–100%. When all the safe-to-approximate operations are approximated, the error for lu is 100%.

In contrast to error, as Figure 11b depicts, lu's energy profile has a narrow spectrum. That is, carefully selecting which operations to approximate can lead to significant energy savings, while offering significantly improved quality-of-results. Compared to approximating all safe-to-approximate operations, our automated genetic filtering improves the error level from 80% to 20%, while only reducing the energy savings from 22% to 14%; a 4× improvement in quality-of-results with only 36% reduction in energy benefits. Finally, Figure 11c shows the fraction of candidate "static" operations that are approximated across generations. As shown, if we only approximate about half of the candidate operations, significant energy saving is achievable with significantly improved quality-of-results. *The results suggest that there is a subset of safe-to-approximate operations that if approximated will significantly degrade quality-of-results.* The genetic filtering algorithm effectively strikes a balance between energy efficiency and error. However, genetic filtering on its own does not provide statistical quality-of-results guarantees, which we address next.

Providing statistical guarantees. Figure 12 shows (a) error, (b) fraction of safe-to-approximate static operations, and (c) energy savings under the High system specification after genetic filtering plus greedy refining. The output expectations are set to 20%, 10%, and 5%. The first bar in Figures 12(a-c) represents the case where all the safe-to-approximate operations are approximated. The errors are averaged across ten inputs. When approximating all operations, even though the average is lower than the expectation (e.g., sobel in Figure 12a), not all the ten inputs satisfy the requirements. Therefore, our selection algorithm refines the approximate subset such that expectations are satisfied on all the ten inputs. As Figure 12a shows, in all cases, the error requirements are satisfied by the selection algorithm and only the subset of safe operations are approximated that do not violate the expectations (Figure 12b) on all ten input datasets.

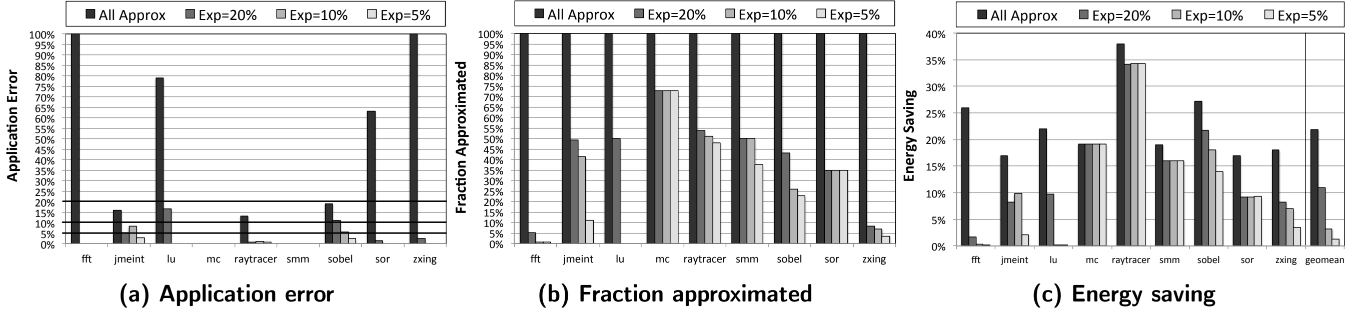
The genetic filtering first excludes the subset of operations that lead to significant quality-of-result degradation and then the greedy refining phase ensures that the expectations are satisfied, while providing significant energy savings. Figure 12c shows up to 35% energy savings for raytracer with 20% error bound and geometric mean of around 10%. Since genetic filtering is a guided random algorithm, some expected minor fluctuations manifest in the results.

Similarly, Figure 13(a-c) illustrates the error, fraction of approximated safe-to-approximate static operations, and energy savings under the Aggressive system specification. Under Aggressive settings, in some applications such as fft and



(a) Application error (b) Fraction approximated (c) Energy saving

Figure 12: Results of the genetic filtering plus greedy refining under High system specification with expectation set to 20%, 10%, and 5% along with the results when all the safe-to-approximate operations are approximated.



(a) Application error (b) Fraction approximated (c) Energy saving

Figure 13: Results of the genetic filtering plus greedy refining under Aggressive system specification with expectation set to 20%, 10%, and 5% along with the results when all the safe-to-approximate operations are approximated.

lu, approximating any operation will lead to the violation of the 10% and 5% expectations. Consequently, the selection algorithm does not approximate any of the operations. However, under the High setting, our selection algorithm was able to approximate a subset of the static operations while satisfying all the expectations. As Figure 12c and Figure 13c show, even though the Aggressive setting has a higher potential for energy savings than the High setting, because of the error expectations, the Aggressive system delivers significantly lower energy benefits with expectation =10% (3% geometric mean energy savings with Aggressive versus 10% with High) and =5% (1% versus 9% geometric mean energy savings). Conversely, as Figure 13c and Figure 12c show zxing gains higher energy savings with the Aggressive system rather than the High system. The selection algorithm finds similar numbers of operations for both settings (Figure 13b and Figure 12b). However, the Aggressive system enables more energy savings with the same operations. *In fact, our framework provides a guidance for the hardware systems to choose the level of approximation considering quality-of-result expectations and application behavior. For fft and lu, an adaptive hardware system would dial down approximation to the High setting to still benefit from approximation while providing statistical quality-of-result guarantees. For zxing, the adaptive system would conversely use the Aggressive setting.* We performed similar experiments with the Mild and Medium settings, which we did not include due to the space limitations. The trends are similar.

For programmers, selecting which operations to approximate to satisfy quality-of-result expectations is a tedious task. The automated ExpAX framework not only reduces the programmer effort in identifying safe-to-approximate operations but also eliminates the need for programmer involvement in providing statistical quality-of-result guarantees.

Confidence in statistical guarantees. To assess the confidence in the statistical guarantees, we measure the 95% central confidence intervals using the ten unseen validation datasets. These inputs are not used during the selection algorithm. Table 3 shows the 95% confidence intervals. To avoid bias when measuring the confidence intervals, we assume that the prior distribution is uniform. Consequently, the posterior will be a Beta distribution, $BETA(k+1, n+1-k)$, where n is the number of datasets (number of observed samples) and k is the number of datasets on which the approximated program satisfies the expectation [26]. In Table 3, the confidence interval (83%,99%) means that on arbitrary input datasets, with 95% confidence, the approximated version of the program will satisfy the programmer expectations with probability between 83% to 99%. The jmeint and lu show slightly lower confidence intervals with the Aggressive system setting. As discussed, an adaptive system can use the confidence information to run the program with other settings that provide higher confidence intervals. *As Table 3 shows, almost uniformly, ExpAX provides high-confidence statistical guarantees that the approximated program will satisfy the specified expectations. These results confirm the effectiveness of the ExpAX framework.*

5.3 Limitations and Considerations

Handling large programs. The framework can divide large programs into smaller kernels and run the optimiza-

Table 3: 95-percent central confidence intervals for satisfying the programmer-specified expectations.

System Specification	Expectation	fft	jmeint	lu	mc	raytracer	smm	sobel	sor	zxing
Mild, Medium, High	20% 10% 5%	(83%,99%)	(83%,99%)	(83%,99%)	(83%,99%)	(83%,99%)	(83%,99%)	(83%,99%)	(83%,99%)	(83%,99%)
Aggressive	20% 10% 5%	(83%,99%)	(83%,99%) (69%,97%) (76%,98%)	(52%,88%) (83%,99%) (83%,99%)	(83%,99%)	(83%,99%)	(83%,99%)	(83%,99%)	(83%,99%)	(83%,99%)

tion step by step. However, the order in which the kernels are optimized may affect the optimality of the results.

Online versus offline selection. This paper only focused on offline optimization. However, a JIT compiler may further optimize the selection during runtime. Nonetheless, there is a tradeoff between compute and energy resources allocated to the optimization and the benefits of online optimization. One option is to offload the online optimization to the cloud where the compute resources are abundant and the parallelism in our selection algorithm can be exploited.

Obtaining input datasets. In the current form, ExpAX requires programmers to provide the training and validation datasets. An alternative is to deploy a precise version of the application and then automatically collect the training and validation data as the application runs in the field. Then, the selection algorithm can update the application code.

Symbolic approximate execution. Our current system is a data-driven optimization framework which relies on profiling information. However, future research can incorporate probabilistic symbolic execution into our framework.

6. Related Work

There is a growing body of work on programming languages, reasoning, analysis, transformations, and synthesis for approximate computing. These works can be characterized based on (1) static vs. dynamic nature, (2) the granularity of approximation, (3) safety guarantees, (4) quality-of-result guarantees, and (5) automation and programmer effort. To this end, ExpAX is a static automated framework that works on the fine grain granularity of single instructions and provides "formal" safety and "statistical" quality-of-result guarantees, while minimizing the programmer effort to only providing high-level implicit annotations and automating the selection of the approximate operations through static analysis and a two-phase optimization algorithm.

EnerJ [21] is an imperative programming language that statically uses the programmer-specified type qualifiers for the program variables. EnerJ works at the granularity of instructions and provides safety but not quality-of-result guarantees. In EnerJ, the programmer must explicitly mark all the approximate variables. Rely [4] is another programming language that requires programmers to explicitly mark both variables and operations as approximate. Rely works at the granularity of instructions and symbolically verifies whether the quality-of-result requirements are satisfied for each function. To provide this guarantee, Rely requires the programmer to not only mark all variables and operations as approximate but also provide preconditions on the reliability and range of the data. Both EnerJ and Rely could be a backend for ExpAX when it automatically generates the approximate version of the program. Similarly, Carbin et al. [3] propose a relational Hoare-like logic for reasoning about the correctness of approximate programs. They provide stronger guarantees than us but require more programmer effort. Further, their framework does not provide solutions for automatically selecting which operations to approximate.

Several works have focused on approximation at the granularity of functions or loops. Loop perforation [16, 24, 25] provides an automatic static technique that periodically skips loop iterations. Even though loop perforation provides statistical quality-of-result guarantees, the technique is not safe and perforated programs may crash. Zhu et al. [29] provide an optimization procedure that uses randomized algorithms to select an alternative approximate implementation of functions at compile-time while providing statistical quality-of-result guarantees. The programmer needs to provide the alternative approximate implementation of the functions. The randomized algorithm in [29] operates at the granularity of functions which is not scalable to the fine-grained approximation model that we investigate. Green [1] provides a code-centric programming model for annotating loops for early termination and functions for approximate substitution. The programmer needs to provide the alternative implementation of the function. Green is also equipped with an online quality-of-result monitoring system that adjusts the level of approximation at runtime. Such runtime adjustments are feasible due to the coarse granularity of the approximation. Similarly, Sage [19] and Paraprox [20] provide a set of static approximation techniques for GPGPU kernels and operate at the granularity of GPU kernels. Both systems are equipped with online monitoring techniques; however, they do not provide statistical quality-of-result guarantees.

Uncertain<T> [2] is a type system for probabilistic programs that operate on uncertain data. It implements a Bayesian network semantics for computation and conditionals on probabilistic data. The work in [22] uses Bayesian network representation and symbolic execution to verify probabilistic assertions. We believe that ExpAX can use these probabilistic models to provide stronger quality-of-result guarantees. However, these techniques are not concerned with programming models or algorithms for selecting approximate operations. The work in [23] uses a genetic algorithm to directly manipulate program binaries for gains in energy. It does not provide safety or quality-of-result guaran-

tees. Alternatively, we use a combination of a genetic and a greedy algorithm to select which operations to approximate.

In contrast to the prior work, we provide a high-level implicit programming model, a static analysis, and a profile-driven optimization that automatically finds approximate operations that are in the granularity of single instructions and execute on unreliable hardware. ExpAX reduces programmer effort to a great extent while providing formal safety and statistical quality-of-result guarantees.

7. Conclusions

We described ExpAX, a comprehensive framework for automating approximate programming that constitutes programming, analysis, and optimization. We developed a programming model based on a new program specification in the form of error expectations. Our programming model enables programmers to implicitly relax the accuracy constraints on low-level program data and operations without explicitly marking them as approximate. The expectations also allow programmers to quantitatively express error constraints. We developed a static safety analysis that uses the high-level expectations and automatically infers a safe-to-approximate set of program operations. We described a system-independent optimization algorithm for selecting a subset of these operations to approximate while delivering significant energy savings and providing statistical guarantees that the error expectations will be satisfied. Selecting which operations to approximate while satisfying quality-of-result expectations is a tedious task for programmers. To this end, the automated ExpAX framework not only reduces the programmer effort in identifying safe-to-approximate operations ($3\times$ to $113\times$) but also eliminates the need for programmer involvement in providing quality-of-result guarantees.

References

[1] W. Baek and T. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

[2] J. Bornholt, T. Mytkowicz, and K. McKinley. Uncertain<T>: A first-order type for uncertain data. In *ASPLOS*, 2014.

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

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

[5] M. de Kruijf and K. Sankaralingam. Exploring the synergy of emerging workloads and silicon reliability trends. In *SELSE*, 2009.

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

[7] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

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

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

[10] Y. Fang, H. Li, and X. Li. A fault criticality evaluation framework of digital systems for error tolerant video applications. In *ATS*, 2011.

[11] R. Hegde and N. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In *ISLPED*, 1999.

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

[13] X. Li and D. Yeung. Exploiting soft computing for increased fault tolerance. In *ASGI*, 2006.

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

[15] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn. Flikker: Saving refresh-power in mobile devices through critical data partitioning. In *ASPLOS*, 2011.

[16] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard. Quality of service profiling. In *ICSE*, 2010.

[17] M. Naik. Chord: A program analysis platform for Java. `http://jchord.googlecode.com`.

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

[19] M. Samadi, J. Lee, D. Jamshidi, A. Hormati, and S. Mahlke. Sage: Self-tuning approximation for graphics engines. In *MICRO*, 2013.

[20] M. Samadi, D. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS*, 2014.

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

[22] A. Sampson, P. Panchekha, T. Mytkowicz, K. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *PLDI*, 2014.

[23] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. Post-compiler software optimization for reducing energy. In *ASPLOS*, 2014.

[24] S. Sidiroglou, S. Misailovic, H. Hoffman, and M. Rinard. Probabilistically accurate program transformations. In *SAS*, 2011.

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

[26] A. Tamhane and D. Dunlop. Statistics and data analysis. In *Prentice-Hall*, 2000.

[27] V. Wong and M. Horowitz. Soft error resilience of probabilistic inference applications. In *SELSE*, 2006.

[28] X. Zhang, M. Naik, and H. Yang. Finding optimum abstractions in parametric dataflow analysis. In *PLDI*, 2013.

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

A. Semantics of Expectations

This appendix gives an instrumented semantics of programs under a given system specification. The goal of this semantics is two-fold: first, it precisely specifies the meaning of expectations; and second, it specifies the runtime instrumentation that our optimization framework needs in order to measure the impact on accuracy and energy of approximating a given set of operations in the program.

Figure 14 shows the domains of the instrumented semantics. We use L to denote the set of labels of operations in the program that are approximated. We use ρ, a valuation of real-valued data to all program variables, to denote both the input to the program and the runtime state of the program at any instant. Finally, we use θ to denote a valuation to all expectations in the program at any instant. The value of expectation labeled k, denoted $\theta(k)$, is either a pair of integers (n_1, n_2) or a real value c, depending upon whether the expectation tracks the error rate or the error magnitude, respectively. In particular, n_1/n_2 denotes the error rate thus far in the execution, and c denotes the largest error magnitude witnessed thus far. Tracking these data suffices to determine, at any instant, whether or not each expectation in the program meets its specified error bound.

We define an instrumented semantics of programs using the above semantic domains. Figure 15 shows the rules of the semantics for the most interesting cases: operations and expectations. For brevity, we omit the rules for the remaining kinds of statements, as they are relatively straightforward. Each rule is of the form:

$$L \models_{\mathcal{S}} \langle s, \rho_1, \rho_1^*, \theta_1 \rangle \overset{r}{\rightsquigarrow} \langle \rho_2, \rho_2^*, \theta_2 \rangle$$

and describes a possible execution of program s under the assumption that the set of approximated operations in the program is L, the start state is ρ_1 with expectation valuation θ_1, and the system is specified by $\mathcal{S}$. The execution ends in state ρ_2 with expectation valuation θ_2, and the energy cost of executing all operations (approximated as well as precise) in the execution is r. Note that ρ_1 and ρ_2 track the actual (i.e., potentially erroneous) values of variables in the approximated program. We call these *actual states*, in contrast to corresponding *shadow states* ρ_1^* and ρ_2^* that track the precise values of variables. We require shadow states to compute expectation valuations θ. For instance, to determine the valuation of expectation $\mathsf{rate}(v) < c$ at the end of an execution, we need to know the fraction of times that this expectation was executed in which an error was incurred on v, which in turn needs determining whether or not v had a precise value each time the expectation was reached in the execution.

To summarize, an instrumented program execution tracks the following extra information at any instant:

- a shadow state ρ^*, a vector of real-valued data of length $|\mathbb{V}|$ that tracks the precise current value of each program variable;
- a real-valued data r tracking the cumulative energy cost of all operations executed thus far; and
- the expectation valuation θ, a vector of integer pairs or real values of length $|\mathbb{K}|$ that tracks the current error incurred at each expectation in the program.

(*approximated operations*)	L	$\subseteq$	$\mathbb{L}$
(*program state*)	ρ, ρ^*	$\in$	$\mathbb{V} \to \mathbb{R}$
(*error expectation values*)	θ	$\in$	$\mathbb{K} \to (\mathbb{Z}^2 \cup \mathbb{R})$

Figure 14: Semantic domains of instrumented program.

We now give the semantics of operations and expectations.

Semantics of Operations. Rules (ASGN-EXACT) and (ASGN-APPROX) in Figure 15 show the execution of a precise and an approximate operation, respectively. The operation $v :=^l \delta(e_1, e_2)$ is approximate if and only if its label l is contained in set L. We use $[\![\delta(e_1, e_2)]\!](\rho)$ to denote the result of expression $\delta(e_1, e_2)$ in state ρ. We need to determine i) the energy cost of this operation, and ii) the value of variable v after the operation executes, in the actual state (determining its value in the shadow state is straightforward). To determine these two quantities, we use the system specification $\mathcal{S}$ to get the error model of operation o as (c_ϵ, r_ϵ) and its energy model as (c_j, r_j). Then, in the precise case, the operation costs energy r_j, whereas in the approximate case, it costs lesser energy $r_j(1 - c_j)$. Moreover, in the approximate case, the operation executes erroneously with probability c_ϵ and precisely with probability $1 - c_\epsilon$; in the erroneous case, the operation yields a potentially erroneous value for variable v, namely $r \pm r_\epsilon$ instead of value r that would result if the operation executed precisely.

Semantics of Expectations. Rules (EXP-RATE), (EXP-MAG), and (EXP-BOTH) in Figure 15 show the execution of the three kinds of expectations. The only thing that these rules modify is the error expectation value of $\theta(k)$, where k is the label of the expectation. We explain each of these three rules separately.

Rule (EXP-RATE) handles the execution of the expectation $\mathsf{rate}(v) < c$, updating incoming value $\theta(k) = (n_1, n_2)$ to either $(n_1, n_2 + 1)$ or $(n_1 + 1, n_2 + 1)$, depending upon whether the actual and shadow values of variable v are equal or not equal, respectively. In both cases, we increment n_2—the number of times this expectation has been executed thus far. But we increment n_1—the number of times this expectation has incurred an error thus far—only in the latter case. At the end of the entire program's execution, we can determine whether or not the error rate of this expectation—over all instances it was reached during that execution—was under the programmer-defined bound c; dividing n_1 by n_2 in the final value of $\theta(k)$ provides this error rate.

Rule (EXP-MAG) handles the execution of the expectation $\mathsf{magnitude}(v) < c$ using ξ, updating incoming value $\theta(k)$ to the greater of $\theta(k)$ and the new magnitude of error incurred by this expectation, as determined by programmer-

defined function ξ. The reason it suffices to keep only the maximum value is, at the end of the entire program's execution, we only require knowing whether or not the maximum error magnitude of this expectation—over all instances it was reached during that execution—was under the programmer-specified bound c.

Finally, Rule (EXP-BOTH) handles the execution of the expectation $\mathsf{magnitude}(v) > c$ using ξ with rate $< c'$, updating incoming value $\theta(k) = (n_1, n_2)$ as in Rule (EXP-RATE), except that the condition under which n_1 is incremented is not that the most recent execution of this expectation incurred an error, but instead that it incurred an error whose magnitude exceeded the programmer-specified bound c (according to programmer-defined function ξ).

B. Meta-Analysis for Approximation Safety

This appendix provides a backward meta-analysis for the approximation safety analysis defined in Section 3.2. Using the terminology of Zhang et al. [28], the approximation safety analysis is called a *forward analysis*. Together, the forward and backward analysis pair form an instance of the framework by Zhang et al. [28], provided the transfer functions for atomic statements of the two analyses meet a requirement described below. The framework instantiated using this pair of analyses efficiently infers the largest set of operations that is safe to approximate in a given program.

The backward meta-analysis is specified by the following data, shown in Figure 16:

- A set Ψ and a function

$$\gamma \in \Psi \to 2^{2^{\mathbb{L}} \times \mathbb{D}}.$$

  Elements in Ψ are the main data structures used by the meta-analysis, and γ determines their meanings. We suggest to read elements in Ψ as predicates over $(2^L \times \mathbb{D})$. The meta-analysis uses such a predicate $\psi \in \Psi$ to express a sufficient condition for verification failure: for every $(L, d) \in \gamma(\psi)$, if we instantiate the forward analysis with set L of operations to be approximated, and run this instance from the abstract state d (over the part of a trace analyzed so far), we will obtain abstract state $\top$ (that is, the forward analysis will not be able to prove that the set of operations L is safe to approximate).

- A function

$$\mathsf{btrans}[t] \in \Psi \to \Psi$$

  for each atomic statement t. The input $\psi_1 \in \Psi$ represents a postcondition on $(2^{\mathbb{L}} \times \mathbb{D})$. Given such ψ_1, the function computes the weakest precondition ψ such that running t from any abstract state in $\gamma(\psi)$ has an outcome in $\gamma(\psi_1)$. This intuition is formalized by the following requirement on the backward analysis's transfer function $\mathsf{btrans}[t]$:

$$\forall \psi_1 \in \Psi : \ \gamma(\mathsf{btrans}[t](\psi_1)) = \{\ (L, d) \mid (L, \mathsf{trans}_L(d)[t]) \in \gamma(\psi_1)\ \}$$

  where trans is the forward analysis's transfer function defined in Figure 8.

$$\psi \in \Psi$$
$$\psi ::= \mathsf{approx}(l) \mid \mathsf{tainted}(v) \mid \mathsf{tainted}(h,f) \mid \mathsf{err} \mid \mathsf{true} \mid \mathsf{false} \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2$$

$$\gamma \in \Psi \to 2^{(2^{\mathbb{L}} \times \mathbb{D})}$$
$$\begin{aligned}
\gamma(\mathsf{approx}(l)) &= \{(L,d) \mid l \in L\}\\
\gamma(\mathsf{tainted}(v)) &= \{(L,\pi) \mid v \in \pi\}\\
\gamma(\mathsf{tainted}(h,f)) &= \{(L,\pi) \mid (h,f) \in \pi\}\\
\gamma(\mathsf{err}) &= \{(L,\top)\}\\
\gamma(\mathsf{true}) &= (2^{\mathbb{L}} \times \mathbb{D})\\
\gamma(\mathsf{false}) &= \emptyset\\
\gamma(\neg\psi) &= (2^{\mathbb{L}} \times \mathbb{D}) \setminus \gamma(\psi)\\
\gamma(\psi_1 \vee \psi_2) &= \gamma(\psi_1) \cup \gamma(\psi_2)\\
\gamma(\psi_1 \wedge \psi_2) &= \gamma(\psi_1) \cap \gamma(\psi_2)
\end{aligned}$$

$$\mathsf{btrans}[t] \in \Psi \to \Psi$$
$$\begin{aligned}
\mathsf{btrans}[v :=^l \delta(e_1,e_2)](\mathsf{err}) &= \mathsf{err}\\
\mathsf{btrans}[\mathsf{precise}(v)](\mathsf{err}) &= \mathsf{err} \vee \mathsf{tainted}(v)\\
\mathsf{btrans}[\mathsf{accept}(v)](\mathsf{err}) &= \mathsf{err}\\
\mathsf{btrans}[p.f := v](\mathsf{err}) &= \mathsf{err}\\
\mathsf{btrans}[v := p.f](\mathsf{err}) &= \mathsf{err}\\
\mathsf{btrans}[t](\mathsf{approx}(l)) &= \mathsf{approx}(l)\\
\mathsf{btrans}[v_2 :=^l \delta(e_1,e_2)](\mathsf{tainted}(v_1)) &= \begin{cases}\mathsf{tainted}(v_1) & \text{if } v_2 \neq v_1\\ \mathsf{approx}(l) \vee \bigvee_{v \in uses(e_1,e_2)} \mathsf{tainted}(v) & \text{if } v_2 = v_1\end{cases}\\
\mathsf{btrans}[\mathsf{precise}(v_2)](\mathsf{tainted}(v_1)) &= \begin{cases}\mathsf{tainted}(v_1) & \text{if } v_2 \neq v_1\\ \mathsf{false} & \text{if } v_2 = v_1\end{cases}\\
\mathsf{btrans}[\mathsf{accept}(v_2)](\mathsf{tainted}(v_1)) &= \begin{cases}\mathsf{tainted}(v_1) & \text{if } v_2 \neq v_1\\ \mathsf{false} & \text{if } v_2 = v_1\end{cases}\\
\mathsf{btrans}[p.f := v_2](\mathsf{tainted}(v_1)) &= \mathsf{tainted}(v_1)\\
\mathsf{btrans}[v_2 := p.f](\mathsf{tainted}(v_1)) &= \begin{cases}\mathsf{tainted}(v_1) & \text{if } v_2 \neq v_1\\ \bigvee_{h \in \mathsf{pts}(p)} \mathsf{tainted}(h) & \text{if } v_2 = v_1\end{cases}\\
\mathsf{btrans}[v :=^l \delta(e_1,e_2)](\mathsf{tainted}(h,f)) &= \mathsf{tainted}(h,f)\\
\mathsf{btrans}[\mathsf{precise}(v)](\mathsf{tainted}(h,f)) &= \mathsf{tainted}(h,f)\\
\mathsf{btrans}[\mathsf{accept}(v)](\mathsf{tainted}(h,f)) &= \mathsf{tainted}(h,f)\\
\mathsf{btrans}[p.f := v](\mathsf{tainted}(h,f)) &= \begin{cases}\mathsf{tainted}(h,f) \vee \mathsf{tainted}(v) & \text{if } h \in \mathsf{pts}(p)\\ \mathsf{tainted}(h,f) & \text{otherwise}\end{cases}\\
\mathsf{btrans}[v := p.f](\mathsf{tainted}(h,f)) &= \mathsf{tainted}(h,f)
\end{aligned}$$

Figure 16: Backward meta-analysis for approximation safety.

$$L \models_{\mathcal{S}} \langle v :=^{l} \delta(e_1, e_2), \rho, \rho^*, \theta\rangle \overset{r_j}{\rightsquigarrow} \langle \rho[v \mapsto r], \rho^*[v \mapsto r^*], \theta\rangle \qquad [\text{if } l \notin L] \qquad (\textsc{Asgn-Exact})$$

$$\text{where } \left[\ \mathcal{S}(\delta) = ((__,__),(__, r_j)) \text{ and } r = [\![\delta(e_1, e_2)]\!](\rho) \text{ and } r^* = [\![\delta(e_1, e_2)]\!](\rho^*)\ \right]$$

$$L \models_{\mathcal{S}} \langle v :=^{l} \delta(e_1, e_2), \rho, \rho^*, \theta\rangle \overset{r_1}{\rightsquigarrow} \langle \rho[v \mapsto r_2], \rho^*[v \mapsto r^*], \theta\rangle \qquad [\text{if } l \in L] \qquad (\textsc{Asgn-Approx})$$

$$\text{where } \left[\begin{array}{c} \mathcal{S}(\delta) = ((c_\epsilon, r_\epsilon),(c_j, r_j)) \text{ and } r = [\![\delta(e_1, e_2)]\!](\rho) \text{ and } r^* = [\![\delta(e_1, e_2)]\!](\rho^*) \\ \text{and } r_1 = r_j(1 - c_j) \text{ and } r_2 = \left\{ \begin{array}{ll} r & \text{with probability } 1 - c_\epsilon \\ r \pm r_\epsilon & \text{with probability } c_\epsilon \end{array} \right. \end{array} \right]$$

$$L \models_{\mathcal{S}} \langle (\mathsf{rate}(v) < c)^k, \rho, \rho^*, \theta\rangle \overset{0}{\rightsquigarrow} \langle \rho, \rho^*, \theta[k \mapsto (n_1', n_2 + 1)]\rangle \qquad (\textsc{Exp-Rate})$$

$$\text{where } \left[\theta(k) = (n_1, n_2) \text{ and } n_1' = \left\{ \begin{array}{ll} n_1 + 1 & \text{if } \rho(v) \neq \rho^*(v) \\ n_1 & \text{otherwise} \end{array} \right. \right]$$

$$L \models_{\mathcal{S}} \langle (\mathsf{magnitude}(v) < c \text{ using } \xi)^k, \rho, \rho^*, \theta\rangle \overset{0}{\rightsquigarrow} \langle \rho, \rho^*, \theta[k \mapsto max(\theta(k), f(\rho(v), \rho^*(v))]\rangle \qquad (\textsc{Exp-Mag})$$

$$L \models_{\mathcal{S}} \langle (\mathsf{magnitude}(v) > c \text{ using } \xi \text{ with rate} < c')^k, \rho, \rho^*, \theta\rangle \overset{0}{\rightsquigarrow} \langle \rho, \rho^*, \theta[k \mapsto (n_1', n_2 + 1)]\rangle \qquad (\textsc{Exp-Both})$$

$$\text{where } \left[\theta(k) = (n_1, n_2) \text{ and } n_1' = \left\{ \begin{array}{ll} n_1 + 1 & \text{if } f(\rho(v), \rho^*(v)) > c \\ n_1 & \text{otherwise} \end{array} \right. \right]$$

Figure 15: Instrumented program semantics. Rules for compound statements and heap operations are omitted for brevity.