

# Expectation-Oriented Framework for Automating Approximate Programming

Jongse Park    Kangqi Ni    Xin Zhang    Hadi Esmaeilzadeh    Mayur Naik

Georgia Institute of Technology

jspark@gatech.edu    vincent.nkq@gatech.edu    xin.zhang@gatech.edu    hadi@cc.gatech.edu    naik@cc.gatech.edu

## Abstract

We describe ExpAX, a framework for automating approximate programming based on programmer-specified error expectations. Three components constitute ExpAX: (1) a programming model based on a new kind of program specification, which we refer to as *expectations*. Our programming model enables programmers to implicitly relax the accuracy constraints without explicitly marking operations approximate; (2) a novel approximation safety analysis that automatically identifies a safe-to-approximate subset of the program operations; and (3) an optimization that automatically marks a subset of the safe-to-approximate operations as approximate while considering the error expectation. Further, we formulate the process of automatically marking operations as approximate as an optimization problem and provide a genetic algorithm to solve it. We evaluate ExpAX using a diverse set of applications and show that it can provide significant energy savings while improving the quality-of-result degradation.

## 1. Introduction

As per-transistor speed and efficiency improvements diminish [8], radical departures from conventional approaches are necessary to improve the performance and efficiency of general-purpose computing. One such departure is general-purpose approximate computing. Recent research [1, 2, 4–7, 10–20] has shown that many emerging applications in both cloud and mobile services are amenable to approximation. For these diverse domains of applications, providing programming models and compiler optimizations for approximation can provide significant opportunities to improve performance and energy efficiency at the architecture and circuit level [5–7, 17]. State-of-the-art systems for approximate computing such as EnerJ [18] and Rely [3] require programmers to *manually and explicitly* declare low-level details, such as the specific variables and operations to be approximated, and provide safety [18] or quality of result guarantees [3]. In contrast, we focus on providing an automated framework that allows programmers to express concise, high-level, and intuitive *error expectation specifications*, and automatically finds the approximable subset of operations in the program. In doing so, our framework enables programmers to implicitly declare *which* parts of the program are safely approximable while explicitly expressing *how much* approximation is preferable.

Figure 1 shows our automated expectation-oriented framework, ExpAX, which constitutes (1) programming, (2) analysis, and (3) optimization. First, ExpAX allows programmers to *implicitly* relax the accuracy constraints on low-level program data and operations by *explicitly* specifying error expectations on program variables. Our programming model provides the syntax and semantics for specifying such high-level error expectations. The program itself without the specific expectation carries the most strict 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 a novel *approximation safety analysis* that automatically finds a the safe-to-approximate subset of program operations. The program outputs on which the programmer

<i>(constant)</i> $r \in \mathbb{R}$	<i>(assignment label)</i> $l \in \mathbb{L}$
<i>(variable)</i> $v \in \mathbb{V}$	<i>(expectation label)</i> $k \in \mathbb{K}$
<i>(operation)</i> $o \in \mathbb{O}$	<i>(expression)</i> $e ::= v \mid r$

<i>(program)</i> $s ::= v :=^l o(e_1, e_2)$
$\phi^k$
$s_1 ; s_2$
<b>if</b> $(v > 0)$ <b>then</b> $s_1$ <b>else</b> $s_2$
<b>while</b> $(v > 0)$ <b>do</b> $s$

Figure 2: Language syntax.

<i>(error)</i> $c \in \mathbb{R}_{0,1} = [0, 1]$
<i>(error magnitude)</i> $f \in (\mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{R}_{0,1}$
<i>(expectation)</i> $\phi ::= \text{rate}(v) < c$
$\text{magnitude}(v) < c$ using $f$
$\text{magnitude}(v) > c$ using $f$ with $\text{rate} < c'$

Figure 3: Expectation syntax.

has specified error expectations are inputs to this analysis. Third, ExpAX includes an optimization framework that selectively marks a number of the candidate operations as approximate while considering the specified error expectations. We formulate the problem of selecting these operations as a general optimization procedure that minimizes error and energy with respect to the specified expectations. Thus, in our formulation, the error expectations guide the optimization procedure to strike a balance between quality-of-result degradation and energy savings.

The optimization procedure is parameterized by a system specification that models error and energy characteristics of the underlying hardware on which the program will execute. While the programmer specifies the error expectations in a high-level, hardware-independent manner, our optimization formulation automatically considers the low-level hardware parameters without exposing them to the programmer. ExpAX thereby enhances the portability of the approximate programs. We implement an instance of the optimization procedure using genetic algorithms and evaluate the effectiveness of the resulting framework using a diverse set of benchmarks. Our empirical results show that for many applications, there is a subset of safe-to-approximate operations that if approximated will result in significant quality-of-result degradation. This insight and the results confirms that automating approximate programming is of significant value. Further, our optimization formulation and its genetic implementation can automatically find and filter this subset while providing significant energy savings.

## 2. Expectation-Oriented Programming Model

### 2.1 Language Syntax

We assume a simple imperative language shown in Figure 2. A program  $s$  in the language is a single procedure with the usual control-flow constructs (sequential composition, branching, and loops). The language supports only real-valued data, only binary operations  $o$  on them, and only expressions of the form  $v > 0$  in

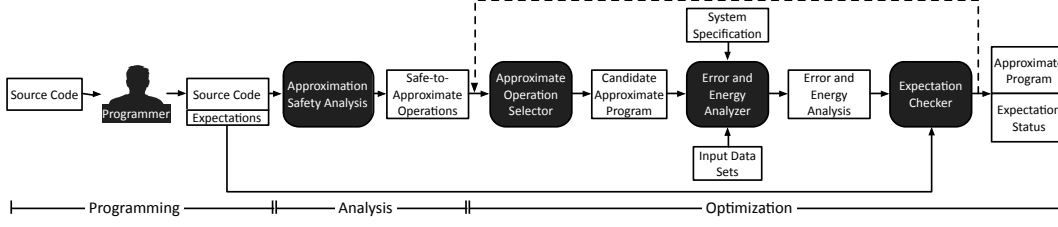


Figure 1: Overview of our expectation-oriented framework, ExpAX, that automates approximate programming.

$$\begin{aligned}
 (\text{error model}) \quad \epsilon &\in \mathbb{E} = (\mathbb{R}_{0,1} \times \mathbb{R}) \\
 (\text{energy model}) \quad j &\in \mathbb{J} = (\mathbb{R}_{0,1} \times \mathbb{R}) \\
 (\text{system spec}) \quad \psi &\in \mathbb{O} \rightarrow (\mathbb{E} \times \mathbb{J})
 \end{aligned}$$

Figure 4: System specification.

conditionals. We limit our formalism to this simplified setting for clarity of exposition. It is straightforward to incorporate procedure calls and structured data types such as arrays, records, and pointers. These extensions are supported in our implementation for the full Java language and we describe their handling in Section 5.

Our simple language supports two kinds of primitive statements, both of which play a special role in our approximation framework: *assignments* and *expectations*. Assignments 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 that each assignment has a unique label  $l \in \mathbb{L}$ . Conversely, expectations  $\phi$  are the only places in the program where the programmer may specify acceptable bounds on such errors. We assume that each expectation has a unique label  $k \in \mathbb{K}$ .

The syntax of expectations is presented in Figure 3. 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. We first informally explain the meaning of these expectations and then give real-world examples illustrating their usefulness. Section 2.3 provides the formal semantics of these expectations.

- Expectation  $\text{rate}(v) < c$  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 exact value  $n_1$  times. Then, the ratio  $n_1/n_2$  should be bounded by  $c$ .
- Expectation  $\text{magnitude}(v) < c$  using  $f$  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  $f$  that takes two arguments—the potentially erroneous value of  $v$  and the exact value of  $v$ —and returns the normalized magnitude of that error.
- Expectation  $\text{magnitude}(v) > c$  using  $f$  with  $\text{rate} < c'$  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  $c'$ .

We illustrate the above three kinds of expectations on a real-world program shown in Figure 5 that performs edge detection on an image. The edge detection algorithm first converts the image to grayscale (lines 6–8). Then, it slides a  $3 \times 3$  window over the pixels

```

1 void detect_edges(Pixel[][] input_image,
2                  Pixel[][] output_image) {
3     float[WIDTH][HEIGHT] gray_image;
4     float[3][3] window;
5     float gradient;
6     for(int y = 0; y < HEIGHT; ++y)
7         for(int x = 0; x < WIDTH; ++x)
8             gray_image[x][y] = to_grayscale(input_image[x][y]);
9     for(int y = 0; y < HEIGHT; ++y)
10        for(int x = 0; x < WIDTH; ++x) {
11            build_window(gray_image, x, y, window);
12            gradient = sobel(p);
13            output_image[x][y].r = gradient;
14            output_image[x][y].g = gradient;
15            output_image[x][y].b = gradient;
16        }
17        accept magnitude[output_image] < 0.1;
18    }
19    float to_grayscale(Pixel p) {
20        float luminance;
21        luminance = p.r * 0.30 + p.g * 0.59 + p.b * 0.11;
22        accept magnitude[luminance] > 0.9 with rate < 0.35;
23        return luminance;
24    }
25    float sobel(float[3][3] p) {
26        float x, y, gradient;
27        x = (p[0][0] + 2 * p[0][1] + p[0][2]);
28        x += (p[2][0] + 2 * p[2][1] + p[2][2]);
29        y = (p[0][2] + 2 * p[1][2] + p[2][2]);
30        y += (p[0][0] + 2 * p[1][1] + p[2][0]);
31        gradient = sqrt(x * x + y * y);
32        gradient = (gradient > 0.7070) ? 0.7070 : gradient;
33        accept rate[gradient] < 0.25;
34        return gradient;
35    }

```

Figure 5: Program illustrating expectation-oriented programming.

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 making Sobel inherently approximate.

The programmer specifies 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 exact 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 upto 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 expectations of approximations. Further, the specified expectations serve to implicitly identify any operations contributing to the computation of data that can be potentially approximated.

(approximated assignments)	$L$	$\subseteq$	$\mathbb{L}$
(program state)	$\rho, \rho^*$	$\in$	$\mathbb{V} \rightarrow \mathbb{R}$
(error expectation values)	$\theta$	$\in$	$\mathbb{K} \rightarrow (\mathbb{Z}^2 \cup \mathbb{R})$

Figure 6: Semantic domains of instrumented program.

## 2.2 System Specification

The system specification provides the error model and the energy model of the system on which our programs execute.

We adopt system specifications of the form shown in Figure 4. Such a specification  $\psi$  specifies an error model  $\epsilon$  and an energy model  $j$  for each approximable operation. In our formalism, this is every operation  $o \in \mathbb{O}$  on real-valued data. Error models and energy models are specified in our framework as follows:

- An error model  $\epsilon$  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 exactly, while  $c$  is the fraction of energy that is saved if the same operation is run approximately.

## 2.3 Instrumented Program Semantics

We now provide 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 assignments in the program.

Figure 6 shows the domains of the instrumented semantics. We use  $L$  to denote the set of labels of assignments in the program that must be approximated. We use  $\rho$ , 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 of execution. Finally, we use  $\theta$  to denote a valuation to all expectations in the program at any instant of execution. 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 of execution, 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. The rules of the semantics are provided in [9]. Each rule is of the form:

$$L \models_{\psi} \langle s, \rho_1, \rho_1^*, \theta_1 \rangle \xrightarrow{\tau} \langle \rho_2, \rho_2^*, \theta_2 \rangle$$

and describes a possible execution of program  $s$  under the assumption that the set of approximated assignments in the program is  $L$ , the start state is  $\rho_1$  with expectation valuation  $\theta_1$ , and the system is specified by  $\psi$ . The execution ends in state  $\rho_2$  with expectation valuation  $\theta_2$ , and the energy cost of executing all assignments (approximated as well as exact) in the execution is  $r$ . Note that  $\rho_1$  and  $\rho_2$  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*  $\rho_1^*$  and  $\rho_2^*$  that track the exact values of variables. We require shadow states to compute expectation valuations  $\theta$ . For instance, to determine the valuation of expectation  $\text{rate}(v) < c$  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 an exact value each time the expectation was reached in the execution.

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

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

## 3. Approximation Safety Analysis

In this section, we present an approximation safety analysis that determines a subset of assignments  $\mathcal{L} \subseteq \mathbb{L}$  that is safe to approximate. In practice, there are two kinds of assignments that are unsafe to approximate. The first kind is those that might violate memory safety, e.g., cause null pointer dereferences or index arrays out of bounds. The second kind are those that might violate functional correctness (i.e., application-specific) properties of the program.

To allow us to reason about approximation safety, we extend our simple language with two kinds of assertions,  $\text{assert}(v)$  and  $\text{assert\_all}$ . The former checks that  $v$  has the exact value, while the latter checks that all *live* variables that are not *endorsed* by some expectation have exact value. We explain the meaning of these assertions by an example before providing their formal semantics.

```
(float, float) m(p : { float x, float y }) {
  assert(p);
  v1 := p.x * 0.5; // 11
  v2 := p.y * 0.7; // 12
  v3 := o(v1, v2); // 13
  v4 := o(v2, v2); // 14
  rate(v3) < 0.1;
  assert_all;
  return(v3, v4);
}
```

Consider function  $m$  above that takes a pointer  $p$  to a pair of real numbers and returns another pair of real numbers. Our framework generates an assertion  $\text{assert}(p)$  at every dereference of variable  $p$ . For brevity, we only show it once, at the dominating dereference. This assertion says that it is unsafe to approximate any assignment whose result propagates to  $p$  at the entry of  $m$ ; otherwise, memory safety could be violated at the dereference of  $p$  in the body of  $m$ .

Our framework also generates assertion  $\text{assert\_all}$  at the end of  $m$ . This assertion says that it is unsafe to approximate any assignment whose result propagates to variable  $v4$  at the end of  $m$ . The reason is that  $v4$  is live at the end of  $m$  and the programmer has not specified any expectation on the error in the value of  $v4$ . Our framework therefore assumes that the programmer expects the value of  $v4$  to be exact (e.g., in order to satisfy an implicit functional correctness property). This in turn prevents our framework from approximating assignments 12 and 14, since they both contribute to the value of  $v4$  at the end of  $m$  (note that 12 assigns to  $v2$ , which is used at 14 to compute  $v4$ ).

On the other hand, it is safe to approximate any remaining assignment whose result propagates to variable  $v3$  at the end of  $m$ . The reason is that the programmer expects errors in the value of  $v3$ , via the expectation  $\text{rate}(v3) < 0.1$  just before the end of  $m$ . This in turn leads our framework to conclude that it is safe to approximate assignments 13 and 11 (note that 12 also contributes to the value of  $v3$  but it was deemed unsafe to approximate above because it also contributes to the value of  $v4$ ). Thus, for this example, only assignments in  $\{ 11, 13 \}$  are safe to approximate.

$$\begin{aligned}
L \models \langle v :=^l o(e_1, e_2), \rho, T, E \rangle &\rightsquigarrow \langle \rho[v \mapsto \llbracket o(e_1, e_2) \rrbracket(\rho)], (T \setminus \{v\}) \cup T', E \setminus \{v\} \rangle \left[ T' = \begin{cases} \{v\} & \text{if } l \in L \text{ or } \textit{uses}(e_1, e_2) \cap T \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \right] \quad (\text{ASGN}) \\
L \models \langle \textit{rate}(v) < c, \rho, T, E \rangle &\rightsquigarrow \langle T, E \cup \{v\} \rangle \quad (\text{EXP-RATE}) \\
L \models \langle \textit{assert}(v), \rho, T, E \rangle &\rightsquigarrow \langle \rho, T, E \rangle \quad [\text{if } v \notin T] \quad (\text{VAR-PASS}) & L \models \langle \textit{assert\_all}, \rho, T, E \rangle &\rightsquigarrow \langle \rho, T, E \rangle \quad [\text{if } T \subseteq E] \quad (\text{ALL-PASS}) \\
L \models \langle \textit{assert}(v), \rho, T, E \rangle &\rightsquigarrow \mathbf{error} \quad [\text{if } v \in T] \quad (\text{VAR-FAIL}) & L \models \langle \textit{assert\_all}, \rho, T, E \rangle &\rightsquigarrow \mathbf{error} \quad [\text{if } T \not\subseteq E] \quad (\text{ALL-FALL}) \\
\frac{L \models \langle s_1, \rho_1, T_1, E_1 \rangle &\rightsquigarrow \langle \rho_2, T_2, E_2 \rangle}{L \models \langle s_1 ; s_2, \rho_1, T_1, E_1 \rangle &\rightsquigarrow \langle \rho_3, T_3, E_3 \rangle} & L \models \langle s_2, \rho_2, T_2, E_2 \rangle &\rightsquigarrow \langle \rho_3, T_3, E_3 \rangle & \quad (\text{SEQ}) \\
L \models \langle \textit{if } (v > 0) \textit{ then } s_1 &\textit{ else } s_2, \rho, T, E \rangle &\rightsquigarrow \mathbf{error} & \quad [\text{if } v \in T] \quad (\text{IF-FAIL}) \\
\frac{L \models \langle s_1, \rho_1, T_1, E_1 \rangle &\rightsquigarrow \langle \rho_2, T_2, E_2 \rangle}{L \models \langle \textit{if } (v > 0) \textit{ then } s_1 &\textit{ else } s_2, \rho_1, T_1, E_1 \rangle &\rightsquigarrow \langle \rho_2, T_2, E_2 \rangle} & \quad [\text{if } v \notin T_1 \wedge \rho_1(v) > 0] \quad (\text{IF-TRUE}) \\
\frac{L \models \langle s_2, \rho_1, T_1, E_1 \rangle &\rightsquigarrow \langle \rho_2, T_2, E_2 \rangle}{L \models \langle \textit{if } (v > 0) \textit{ then } s_1 &\textit{ else } s_2, \rho_1, T_1, E_1 \rangle &\rightsquigarrow \langle \rho_2, T_2, E_2 \rangle} & \quad [\text{if } v \notin T_1 \wedge \rho_1(v) \leq 0] \quad (\text{IF-FALSE})
\end{aligned}$$

Figure 7: Semantics of approximation safety. Rules for other kinds of expectations, loops, and rules propagating **error** are elided for brevity.

$$\begin{aligned}
L, \langle T, E \rangle \vdash v :=^l o(e_1, e_2) \triangleright \langle (T \setminus \{v\}) \cup T', E \setminus \{v\} \rangle & \quad (\text{ASGN}) \\
\text{where } T' = \begin{cases} \{v\} & \text{if } l \in L \text{ or } \textit{uses}(e_1, e_2) \cap T \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
L, \langle T, E \rangle \vdash \textit{rate}(v) < c \triangleright \langle T, E \cup \{v\} \rangle & \quad (\text{EXP-RATE}) \\
L, \langle T, E \rangle \vdash \textit{assert}(v) \triangleright \langle T, E \rangle & \quad [\text{if } v \notin T] \quad (\text{ASSERT-VAR}) \\
L, \langle T, E \rangle \vdash \textit{assert\_all} \triangleright \langle T, E \rangle & \quad [\text{if } T \subseteq E] \quad (\text{ASSERT-ALL}) \\
\frac{L, \langle T_1, E_1 \rangle \vdash s_1 \triangleright \langle T_2, E_2 \rangle \quad L, \langle T_2, E_2 \rangle \vdash s_2 \triangleright \langle T_3, E_3 \rangle}{L, \langle T_1, E_1 \rangle \vdash s_1 ; s_2 \triangleright \langle T_3, E_3 \rangle} & \quad (\text{SEQ}) \\
\frac{L, \langle T, E \rangle \vdash s_1 \triangleright \langle T_1, E_1 \rangle \quad L, \langle T, E \rangle \vdash s_2 \triangleright \langle T_2, E_2 \rangle}{L, \langle T, E \rangle \vdash \textit{if } (v > 0) \textit{ then } s_1 \textit{ else } s_2 \triangleright \langle T_1 \cup T_2, E_1 \cap E_2 \rangle} & \quad (\text{IF}) \\
\frac{L, \langle T, E \rangle \vdash s \triangleright \langle T, E \rangle}{L, \langle T, E \rangle \vdash \textit{while } (v > 0) \textit{ do } s \triangleright \langle T, E \rangle} & \quad (\text{WHILE})
\end{aligned}$$

Figure 8: Approximation safety analysis. Rules for the two kinds of expectations not shown for brevity are similar to (EXP-RATE).

A semantics formalizing the meaning of our assertions is shown in Figure 7. Each rule is of the form:

$$L \models \langle s, \rho_1, T_1, E_1 \rangle \rightsquigarrow \langle \rho_2, T_2, E_2 \rangle \mid \mathbf{error}$$

It describes whether an execution of the program  $s$  starting with input state  $\rho_1$  will violate any assertions in  $s$  if the set of assignments  $L$  is approximated. If any assertion is violated, the final state is designated **error**; otherwise, it is the normal output state  $\rho_2$ .

To determine whether any assertion will be violated, the semantics tracks two sets of variables  $T$  and  $E$ , which we call the *tainted* and *endorsed* sets, respectively. Intuitively, a variable gets tainted if its value is affected by some assignment in the set  $L$ , and a variable gets endorsed if an expectation on that variable is executed.

We explain the most interesting rules in Figure 7. Rule (ASGN) states that the target variable  $v$  of an assignment  $v :=^l o(e_1, e_2)$  gets tainted if  $l$  is an approximated assignment (i.e.,  $l \in L$ ) or if a variable used in  $e_1$  or  $e_2$  is already tainted. Rule (VAR-FAIL) states that  $\textit{assert}(v)$  fails if  $v$  is tainted (i.e.,  $v \in T$ ). Rule (ALL-FAIL) states that  $\textit{assert\_all}$  fails if any tainted variable is not endorsed (i.e.,  $T \not\subseteq E$ ). Strictly, we require only *live* tainted variables to be endorsed, but for simplicity the semantics assumes all variables are live, though our implementation does a liveness analysis.

Given the semantics of assertions, we can characterize the sets of assignments that are safe to approximate, as follows:

**DEFINITION 3.1. (Approximation Safety Problem)** *A set of assignments  $L$  is safe to approximate if for every  $\rho$ , there exists  $\rho', T, E$  such that  $L \models \langle s, \rho, \emptyset, \emptyset \rangle \rightsquigarrow \langle \rho', T, E \rangle$ .*

We now present a static analysis that conservatively estimates a set of assignments that are safe to approximate. The analysis is shown in Figure 8 using type rules. It states that a set of assignments  $L$  is safe to approximate in a program  $s$  if there exists tainted sets of variables  $T_1, T_2$  and endorsed sets of variables  $E_1, E_2$  such that the type judgment  $L, \langle T_1, E_1 \rangle \vdash s \triangleright \langle T_2, E_2 \rangle$  holds. The rules are self-explanatory and mirror those of the semantics of assertions in Figure 7. The most interesting one is rule (IF) which states that outgoing tainted sets  $T_1$  and  $T_2$  of the true and false branches of a conditional statement are unioned, whereas the outgoing endorsed sets  $E_1$  and  $E_2$  are intersected. The reason is that it is safe to grow tainted sets and shrink endorsed sets, as evident in the checks of rules (ASSERT-VAR) and (ASSERT-ALL).

The following theorem states that our static analysis is sound:

**THEOREM 3.2. (Soundness)** *If  $L, \langle T_1, E_1 \rangle \vdash s \triangleright \langle T'_1, E'_1 \rangle$  and  $T_2 \subseteq T_1$  and  $E_1 \subseteq E_2$  then  $\exists \rho'$  such that  $L \models \langle s, T_2, E_2 \rangle \rightsquigarrow \langle \rho', T'_2, E'_2 \rangle$  and  $T'_2 \subseteq T'_1$  and  $E'_1 \subseteq E'_2$ .*

We seek to find as large a set of assignments as possible that is safe to approximate, so as to maximize the opportunities for energy savings. One can use techniques for finding optimal abstractions (e.g., [21]) to efficiently identify the largest set of safe approximable assignments. Hereafter, we denote the set of safe approximable assignments as  $\mathcal{L}$ .

## 4. Optimization Framework

### 4.1 Optimization Problem Formulation

Figure 9 presents the formulation of our optimization problem to automatically select a subset of the safe-to-approximate assignments. The objective is to approximate as much of the program to reduce energy while minimizing error, considering the specified error expectations. To this end, the optimization procedure aims to minimize the weighted normalized sum of three components:

1. Number of static assignments in the program that are not approximate  $(1 - |\mathbf{L}|/|\mathcal{L}|)$ .
  2. Excess error (rates and magnitudes) incurred due to approximations, over and above the bounds specified in expectations by programmers ( $\textit{error}(\rho)$ ).
  3. Energy cost incurred due to lack of approximation ( $\textit{energy}(\rho)$ ).
- In our optimization framework, we assume that error and energy are computed with respect to given program input datasets  $(\rho_1, \dots, \rho_n)$ .

**Inputs:**

- (1) Program  $\Pi = \langle s, \mathcal{L}, \mathbb{K} \rangle$
- (2) Program input datasets  $\Delta = \{\rho_1, \dots, \rho_n\}$
- (3) System specification  $\psi$

**Output:**

Best set of approximable assignments  $\mathbf{L} \subseteq \mathcal{L}$ .

**Minimize:**

$$\sum_{\rho \in \Delta} \left( \alpha \left(1 - \frac{|\mathbf{L}|}{|\mathcal{L}|}\right) + \beta \text{error}(\rho) + \gamma \text{energy}(\rho) \right)$$

where  $\alpha, \beta, \gamma \in \mathbb{R}_{0,1}$  and  $\alpha + \beta + \gamma = 1$  and:

- (1)  $\text{error}(\rho) = \frac{\sum_{k \in \mathbb{K}} \text{error}(\theta(k), \text{bound}(k))}{|\mathbb{K}|}$  where:
  - $\theta$  s.t.  $\exists \rho_1, \rho_2, r : \mathbf{L} \models_{\psi} \langle s, \rho, \rho, \lambda k.0 \rangle \xrightarrow{r} \langle \rho_1, \rho_2, \theta \rangle$
  - $\text{bound}(k) = c$  where expectation labeled  $k$  is of form:
    - $\text{rate}(v) < c$  or
    - $\text{magnitude}(v) < c$  using  $f$  or
    - $\text{magnitude}(v) > c'$  using  $f$  with  $\text{rate} < c$
  - $\text{error}(c_1, c_2) = \begin{cases} c_1 - c_2 & \text{if } c_1 > c_2 \\ 0 & \text{otherwise} \end{cases}$
- (2)  $\text{energy}(\rho) = \frac{r}{r^*}$  where:
  - $r$  s.t.  $\exists \rho_1, \rho_2, \theta : \mathbf{L} \models_{\psi} \langle s, \rho, \rho, \lambda k.0 \rangle \xrightarrow{r} \langle \rho_1, \rho_2, \theta \rangle$
  - $r^*$  s.t.  $\exists \rho_1, \rho_2, \theta : \emptyset \models_{\psi} \langle s, \rho, \rho, \lambda k.0 \rangle \xrightarrow{r^*} \langle \rho_1, \rho_2, \theta \rangle$

Figure 9: Optimization problem formulation.

Minimizing  $\text{energy}(\rho)$  and the number of non-approximate static assignments reduces energy but can lead to significant quality-of-result degradation. The  $\text{error}(\rho)$  term in the minimization objective functions strikes a balance between saving energy and preserving the quality-of-results, while considering the programmer expectations. The  $\text{energy}(\rho)$  term in the formulation is the energy dissipation when a subset of the approximable assignments are approximated, normalized to the energy dissipation of the program when executed on the same input dataset with no approximation.

**Incorporating expectation in the optimization.** The  $\text{error}(\rho)$  term is the normalized sum of error (rate or magnitude) at the site of all expectations when the program runs on the  $\rho$  dataset. We incorporate the expectations in the optimization through  $\text{error}(\rho)$ . We assume error on an expectation site is 0, if the observed error is less than the expectation bound. Otherwise, we assume 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 is to push the error below the specified bound since the programmer has specified that error below the expectation bound is acceptable.

## 4.2 Genetic Optimization Framework

We develop a genetic algorithm that explores the space of possible solutions for the formulated optimization problem and finds a best-effort answer. However, there may be other algorithms for solving the formulated optimization problem.

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

levels to each element, allowing multiple levels of approximation for each assignment. 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 assignments, 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 based on the objective function in Figure 9:

$$f(\text{phenotype}) = \left( \alpha \frac{\sum_j \text{phenotype}_j}{|\mathcal{L}|} + \beta \text{error} + \gamma \text{energy} \right)^{-1}$$

We use the inverse of minimization object in Figure 9 to calculate fitness scores. That is, a phenotype with more approximate instructions ( $\frac{\sum_j \text{phenotype}_j}{|\mathcal{L}|}$ ), less *error*, and less *energy* has a higher score and higher chance of reproducing better solutions. Since we use '0' to mark an assignment as approximate, the lower the  $\sum_j \text{phenotype}_j$ , the more instructions are approximated, which is in correspondence with the formulation in Figure 9. As discussed, the *error* incorporates the programmer-specified expectations. Since not all the static assignments contribute equally to the dynamic error and energy dissipation of the program, we set  $\alpha = 0.2$ ,  $\beta = 0.4$ , and  $\gamma = 0.4$ . This setting equally weighs the dynamic information captured in *error* and *energy* while giving less importance to the number of static assignments that are approximated. We generate approximate versions of the program based on the phenotypes and run them with the input datasets. We make each version of the program an independent thread and run them in parallel. Executing different input datasets can also be parallel.

**Roulette wheel gene selection.** We use the roulette wheel selection strategy for generating the next generation. The probability of selecting the  $i^{\text{th}}$  phenotype is  $\text{gl}[i] = \frac{f_i}{\sum f_i}$ . That is, phenotype with higher fitness have a better chance of producing offsprings. Nonetheless, the low score phenotypes will still have the chance to reproduce since they may contain part of a better solution that will manifest several generation later.

**Crossover and mutation.** The crossover operator takes two selected phenotypes and recombines them to generate a new solution. All the phenotypes are of the same size. Therefore, crossover randomly picks a location on the bit-vector, cuts the two bit-vectors, and swaps the cut parts. Since crossover does not improve the diversity of the solutions, with a small probability, we mutate the phenotypes after each crossover by randomly flips a bit.

**Elitism.** We record the best global solution across all the generations. This approach is referred to as elitism.

## 5. Evaluation

**Benchmarks.** We examine nine benchmarks written in Java, which are the same programs used in [18]. Table 1 summarizes the benchmarks and shows their quality metric.

**Application-specific quality metrics.** For *jmeint* and *zxing*, the quality metric is a rate. For *jmeint*, the quality metric is the rate of incorrect intersection decisions. Similarly, for the *zxing* bar code recognizer, the quality metric is the rate of unsuccessful decodings of a sample QR code image. 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.

Table 1: Benchmarks, quality metric (Mag=Magnitude), and result of approximation safety analysis: # approximable operations.

Description		Quality Metric	# Lines	# Approximable Operations
fft		Mag: Avg entry diff	168	123
sor	SciMark2	Mag: Avg entry diff	36	23
mc	benchmark:	Mag: Normalized diff	59	11
smm	scientific kernels	Mag: Avg normalized diff	38	7
lu		Mag: Avg entry diff	283	46
zxing	Bar code decoder for mobile phones	Rate of incorrect reads	26171	756
jmeint	jMonkeyEngine game: triangle intersection kernel	Rate of incorrect decisions	5962	1046
imagefill	ImageJ image processing: flood-filling kernel	Mag: Avg pixel diff	156	130
raytracer	3D image renderer	Mag: Avg pixel diff	174	199

Table 2: Summary of the (*error rate, energy saving*)’s with our four system specifications derived from [18].

Operation	Mild	Medium	High	Aggressive
Instruction	( $10^{-6}$ , 12%)	( $10^{-4}$ , 22%)	( $10^{-3}$ , 26%)	( $10^{-2}$ , 30%)
On-chip read	( $10^{-16.7}$ , 70%)	( $10^{-7.4}$ , 80%)	( $10^{-5.2}$ , 85%)	( $10^{-3}$ , 90%)
On-chip write	( $10^{-5.6}$ , 70%)	( $10^{-4.9}$ , 80%)	( $10^{-4}$ , 85%)	( $10^{-3}$ , 90%)
Off-chip	( $10^{-9}$ , 17%)	( $10^{-5}$ , 22%)	( $10^{-4}$ , 23%)	( $10^{-3}$ , 24%)

**Genetic optimization framework.** We implement the genetic framework described earlier to automatically find the approximate version of the program considering the specified expectations. 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.

**Simulator.** We modified the open-source simulator provided in [18] and built our genetic optimization framework on top of the modified simulator.

**System specifications.** We derive four system specifications from the energy models provided in [18] as shown in Table 2. We consider a simplified processor/memory system energy dissipation model that comprises three components: instruction execution, on-chip storage (SRAM), and off-chip storage (DRAM).

**Architecture model.** We assume an architecture similar to Truffle [7] that can execute interleaving of precise and approximate instructions and does not incur overhead when switching between approximate and precise instructions.

**Genetic optimization.** To understand the effectiveness of the genetic optimization, we take a close look at lu when it undergoes the genetic optimization under the Aggressive system specification with the output expectation set to 0.00. We will also present the energy savings and error levels for all the applications. Figure 10a depicts the distribution of error for lu in each generation of the genetic optimization. As shown, the application shows a wide spectrum of error, 18%–100%. *The result shows that there is a subset of safe-to-approximate operations that if approximated can lead to significant quality-of-result degradation.*

In contrast to error, as Figure 10b depicts, lu’s energy profile has a narrow spectrum. That is, carefully selecting which operation to approximate can lead to significant energy savings, while delivering significantly improved quality-of-results. Compared to approximating all safe-to-approximate operations, our automated genetic

framework improves the error level from 100% to 18%, while only reducing the energy savings from 21% to 16%. That is,  $5.5\times$  improvement in quality-of-results with only 31% reduction in energy benefits. Finally, Figure 10c shows the fraction of candidate static operations that are approximated across generations.

**Effect of error bounds on the genetic optimization.** Figure 11 shows (a) error, (b) energy savings, and (c) fraction of approximated safe-to-approximate static operations under the Medium system specification. The output expectation bounds are set to 0.00, 0.03, 0.05, 0.10, 0.50, and 0.75. The last bar in Figures 11(a-c) represents the case when all the safe-to-approximate operations are approximated. In general, the larger the expectation bounds, the more relaxed the error requirements. Almost consistently, as the error requirements become more strict, the genetic algorithm finds lower-error versions of the approximate programs, while maintaining significant energy benefits (Figure 11b). The genetic optimization also filters out the subset of the static operations that lead to significant quality-of-result degradation (Figure 11c). Since the genetic algorithm is a guided random algorithm, there are some expected minor fluctuations in the results.

As Figure 11a shows, the correlation of error reduction with tighter error bounds is evident in the case of fft. The automated genetic optimization has been able to find approximate versions of fft that exhibit significantly better quality-of-results while delivering significant energy savings. This case shows that the bounds guide the genetic optimization to find and filter out safe-to-approximate operations that lead to significant quality degradation, while maintaining significant energy savings (Figure 11b). For sor, the genetic algorithm has been able to satisfy all the expectations and reduce the error to negligible levels when the bounds are strict. For mc, smm, lu, and zxing, imagefill, raytracer, the genetic algorithm has been able to find a near-to-optimal solution. However, in case of jmeint the genetic algorithm has failed to find the optimal solution, which is approximating all the operations. We can always consider the fully approximated solution as a candidate solution and bypass genetic optimization if it satisfies the expectations. As Figure 11c shows, for fft, sor, lu, zxing, and raytracer the genetic optimization filters out the subset of safe-to-approximate operations that when approximated lead to quality degradations. We performed similar experiments with the Aggressive system specification and found similar trends. However, due to space limitations we did not include those results. *These results confirm that automating the process of selecting approximate operations is of significant value and confirms the effectiveness of our optimization formulation and our concrete genetic algorithm solution.*

## 6. Conclusions

We described ExpAX, an expectation-oriented framework for automating approximate programming, and its three components: programming, analysis, and optimization. We developed a programming model and a new program specification, referred to as 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. Further, the expectations allow programmers to quantitatively express their error preferences. Then, we developed an approximation safety analysis that using the high-level expectation specifications automatically finds the candidate safe-to-approximate subset of the operations. Then, we described a system-independent optimization formulation for selectively marking a number of the candidate operations as approximate. Further, we provided a concrete instantiation of the optimization framework using a genetic algorithm and evaluated its effectiveness using a diverse set of applications. The results show that in many cases, there is a subset of the safe-to-approximate operations that if approximated can lead

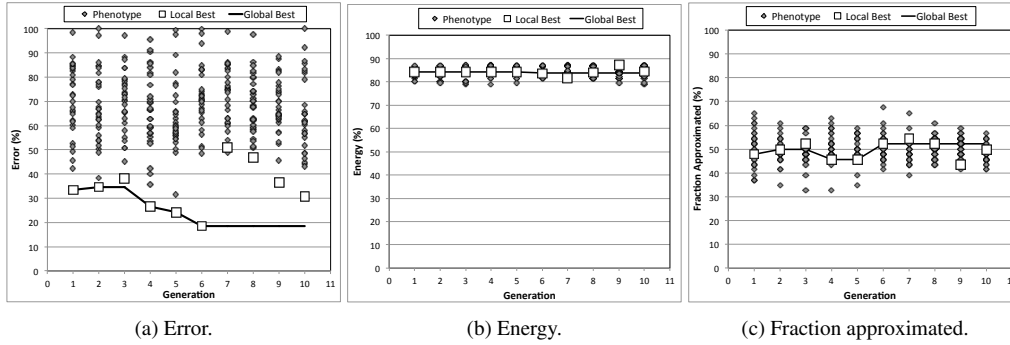


Figure 10: The lu benchmark under the genetic optimization with the Aggressive system specification and the output expectation set to 0.00.

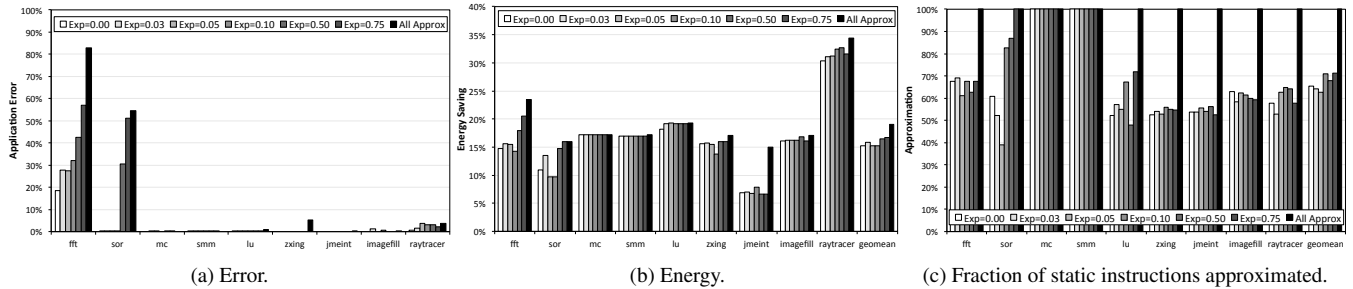


Figure 11: Results of the genetic optimization under the Medium system specification with expectation bounds set to 0.00, 0.03, 0.05, 0.10, 0.50, and 0.75 along with the results when all the safe-to-approximate operations are approximated.

to significant quality-of-result degradations. We show that our genetic optimization framework effectively excludes these operations from approximation. The results confirm that automating this process is of significant value and can in many cases enable effective approximation even on highly unreliable hardware. Our approach is a best-effort solution to automate approximate programming, improve their portability, automatically balance quality-of-result degradation and efficiency, and guide approximation based on programmers' expectations.

## References

- [1] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54(7), 2005.
- [2] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.
- [3] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*, 2013.
- [4] M. de Kruijf and K. Sankaralingam. Exploring the synergy of emerging workloads and silicon reliability trends. In *SELSE*, 2009.
- [5] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *ISCA*, 2010.
- [6] H. Esmailzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.
- [7] H. Esmailzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, 2012.
- [8] H. Esmailzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2), 2013.
- [9] H. Esmailzadeh, V. Ni, and M. Naik. Expectation-oriented framework for automating approximate programming. Technical report, Georgia Tech, 2013. Available at <https://smartech.gatech.edu/handle/1853/49755>.
- [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. R. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In *ISLPED*, 1999.
- [12] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra. ERSAs: 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. G. Zorn. Flicker: 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] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones. Scalable stochastic processors. In *DATE*, 2010.
- [18] 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.
- [19] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy tradeoffs with loop perforation. In *FSE*, 2011.
- [20] V. Wong and M. Horowitz. Soft error resilience of probabilistic inference applications. In *SELSE*, 2006.
- [21] X. Zhang, M. Naik, and H. Yang. Finding optimum abstractions in parametric dataflow analysis. In *PLDI*, 2013.