FLEXJAVA: Language Support for Safe and Modular Approximate Programming

Jongse Park Hadi Esmaeilzadeh Xin Zhang Mayur Naik William Harris
Georgia Institute of Technology, GA, USA
jspark@gatech.edu hadi@cc.gatech.edu xin.zhang@gatech.edu
naik@cc.gatech.edu wharris@cc.gatech.edu

ABSTRACT

Energy efficiency is a primary constraint in modern systems. Approximate computing is a promising approach that trades quality of result for gains in efficiency and performance. State-of-the-art approximate programming models require extensive manual annotations on program data and operations to guarantee safe execution of approximate programs. The need for extensive manual annotations hinders the practical use of approximation techniques. This paper describes FLEXJAVA, a small set of language extensions, that significantly reduces the annotation effort, paving the way for practical approximate programming. These extensions enable programmers to annotate approximation-tolerant method outputs. The FLEXJAVA compiler, which is equipped with an approximation safety analysis, automatically infers the operations and data that affect these outputs and selectively marks them approximable while giving safety guarantees. The automation and the language–compiler codesign relieve programmers from manually and explicitly annotating data declarations or operations as safe to approximate. FLEXJAVA is designed to support safety, modularity, generality, and scalability in software development. We have implemented FLEXJAVA annotations as a Java library and we demonstrate its practicality using a wide range of Java applications and by conducting a user study. Compared to EnerJ, a recent approximate programming system, FLEXJAVA provides the same energy savings with significant reduction (from 2× to 17×) in the number of annotations. In our user study, programmers spend 6× to 12× less time annotating programs using FLEXJAVA than when using EnerJ.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Frameworks; D.2.4 [**Software/Program Verification**]: Reliability

Keywords

Language design, modular approximate programming

1. INTRODUCTION

Energy efficiency is a primary concern in modern systems. Mobile devices are limited by battery life and a significant fraction of the data center cost emanates from energy consumption. Furthermore, the dark silicon phenomenon limits the historical improvements in energy efficiency and performance [10]. Approximate computing is a promising approach that trades small and acceptable loss of output quality for energy efficiency and performance gains [6, 11, 13, 17, 27, 28, 30, 33]. This approach exploits the inherent tolerance of applications to occasional error to execute faster or use less energy. These applications span a wide range of domains including web search, big-data analytics, machine learning, multimedia, cyber-physical systems, speech and pattern recognition, and many more. For instance, a lossy video codec can tolerate imprecision and occasional errors when processing pixels of a frame. Practical programming models for approximation are vital to fully exploit this opportunity. Such models can provide significant improvements in performance and energy efficiency in the hardware by relaxing the abstraction of full accuracy [2, 8, 24, 39].

Safe execution of programs is crucial to the applicability of such techniques. That is, the programming model needs to guarantee that approximation will never lead to catastrophic failures such as array out-of-bound exceptions. Recent works on approximate programming languages [6, 30] enable these techniques to provide such safety guarantees. These guarantees, however, come at the expense of extensive programmer annotations: programmers need to manually annotate all approximate variable declarations [30] or even annotate the safe-to-approximate operations [6]. This need for extensive annotations hinders the practical use of approximation techniques.

In this paper, we propose a small set of language extensions that significantly lowers the annotation effort and paves the way for practical approximate programming. To achieve this goal, we identified the following challenges that need to be addressed. The extensions should enable programmers to annotate approximation-tolerant method outputs. The compiler then should automatically infer the operations and data that affect these outputs and selectively mark them approximable while providing safety guarantees. This process should be automatic and the language–compiler should be codesigned in order to relieve programmers from manually and explicitly annotating data declarations or operations. This paper addresses these challenges through the following contributions:

1. We introduce FLEXJAVA, a small set of extensions that enables safe, modular, general, and scalable object-oriented approximate programming. It provides these features by introducing only four intuitive annotations. FLEXJAVA supports modularity by defining a scope for the annotations based on the syntactic structure of the program. Scoping and adherence to program structure makes annotation a natural part of the software development process (Section 3.).

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786807

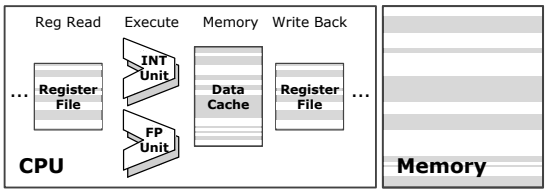

Figure 1: A processor that supports fine-grained approximation. The shaded units perform approximate operations or store data in approximate storage.

2. The FLEXJAVA annotations are designed to support both coarse-grained and fine-grained approximation, and enable programmers to specify a wide range of quality requirements, quality metrics, and recovery strategies (Section 3.).
3. The language is codesigned with a compiler that *automatically* infers the safe-to-approximate data and operations from limited annotations on program or function outputs. The compiler statically enforces safety using a scalable dataflow analysis that conservatively infers the maximal set of safe-to-approximate data and operations. This automated analysis significantly reduces the number of annotations and avoids the need for safety checks at runtime (Section 5.).
4. We implemented FLEXJAVA annotations as a library to make it compatible with Java programs and tools. We extensively evaluate FLEXJAVA using a diverse set of programs and by conducting a user study (Section 4. and Section 6.).

The results of our evaluation show that FLEXJAVA reduces the number of annotations (from 2× to 17×) compared to EnerJ, a recent approximate programming language. We also conduct a user study that shows from 6× to 12× reduction in annotation time compared to EnerJ. With fine-grained approximation and small losses in quality, FLEXJAVA provides the same level of energy savings (from 7% to 38%) compared to EnerJ. With coarse-grained approximation, FLEXJAVA achieves even higher benefits—2.2× average energy reduction and 1.8× average speedup—for less than 10% quality loss.

A growing body of work is proposing new approximation techniques that stand to deliver an order of magnitude benefits in both energy and performance [2, 12, 27, 28, 35]. Our results suggest that practical programming solutions, such as FLEXJAVA, are imperative for making these techniques widely applicable.

2. BACKGROUND

Approximation techniques are broadly divided into two types: (1) fine-grained techniques that apply approximation at the granularity of individual instructions and data elements, and (2) coarse-grained techniques that apply approximation at the granularity of entire code blocks. FLEXJAVA supports both types of techniques. We review the literature on these techniques before presenting the design of FLEXJAVA.

Fine-grained approximation. Architectures support fine-grained approximation by allowing to execute interleaved approximate and precise instructions [6, 11, 19, 30, 34]. As Figure 1 shows, such architectures support both approximate operations and approximate storage. A bit in the instruction opcode identifies whether the instruction is the approximate or the precise version. Current proposals for approximate instructions lack room for enough bits to encode multiple approximation levels. As a result, we assume the prevalent binary-level approximation [6, 11, 19, 30, 34], although our approach can take advantage of multi-level approximation.

In this model, an approximate instruction has probabilistic semantics: it returns an approximate value with probability p and the precise value with probability $1-p$. The approximate value may be arbitrary. The architecture also allows approximate storage, i.e., program data can be stored in approximate sections of the memory, cache, or registers. We use three such probabilistic architecture settings, shown in Table 1, that offer increasing energy savings with higher error probabilities. These models are similar to the ones that are used in recent works on approximate programming [6, 30].

Table 1: Error probabilities and energy savings for different operations in fine-grained approximation. We consider the three hardware settings of Mild, Medium, and Aggressive from [30].

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

Coarse-grained approximation. Coarse-grained approximation techniques concern approximating entire loop bodies or functions [3, 12, 33]. Loop perforation [33] is one such technique that transforms loops to skip a subset of their iterations. Green [3] substitutes functions with simpler approximate implementations or terminates loops early. NPUs [12] are a new class of accelerators that replace functions with hardware neural networks to approximately mimic the functions behavior. More generally, as the focus of the semiconductor industry shifts to programmable accelerators [15, 26, 36, 37], coarse-grained approximation can pave the way for new classes of approximate accelerators that can deliver significantly better performance and energy savings.

3. FLEXJAVA LANGUAGE DESIGN

We have designed a set of language extensions for approximate programming that satisfy four key criteria:

1. **Safety.** The extensions guarantee ***safe*** execution. In other words, approximation can never lead to catastrophic failures, such as array out-of-bound exceptions.
2. **Modularity.** The extensions are ***modular*** and do not hinder modular programming and reuse.
3. **Generality.** The extensions are ***general*** and enable utilizing a wide range of approximation techniques without exposing their implementation details.
4. **Scalability.** The extensions are ***scalable*** and let programmers annotate large programs with minimal effort.

We have incorporated these extensions in the Java language. This section describes approximate programming in the resulting language FLEXJAVA using a series of examples. In the examples, **`bold-underline`** highlight the safe-to-approximate data and operations that the FLEXJAVA compiler infers automatically from the programmer annotations. Section 5. presents the formal semantics of the annotations and the static analysis performed by the compiler.

3.1 Safe Programming in FLEXJAVA

Providing safety guarantees is the first requirement for practical approximate programming. That is, the approximation should never affect critical data and operations. The criticality of data and operations is a semantic property of the application that can only be identified by the programmer. The language must therefore provide a mechanism for programmers to specify where approximation is safe. This poses a language-compiler co-design challenge in order to alleviate the need for manually annotating all the approximate data and operations. To address this challenge, we provide two language annotations, called loosen and

tighten. These annotations provide programmers with full control over approximation without requiring them to manually and explicitly mark all the safe-to-approximate data and operations.

Selectively relaxing accuracy requirements. As discussed above, not all program data and operations are safe to approximate. Therefore, FLEXJAVA allows each data and operation in the program to be either precise or approximate. Approximate data can be allocated in the approximate sections of memory, and an approximate operation is a variant that may generate inexact results. All data and operations are precise by default. The loosen annotation allows to relax the accuracy requirement on a specified variable at a specified program point. That is, any computation and data that *exclusively* affects the annotated variable is safe to approximate. For example, in the following snippet, the programmer uses loosen(luminance) to specify that the computation of luminance can be safely approximated.

```
float computeLuminance (float r, float g, float b) {
  float luminance = r * 0.3f + g * 0.6f + b * 0.1f;
  loosen(luminance);
  return luminance;  }
```

From this single annotation, the FLEXJAVA compiler automatically infers that data r, g, b, and luminance can be safely allocated in the approximate memory. It also infers that all arithmetic operations, loads, and stores that contribute to calculating luminance are approximable. To provide memory safety and avoid null pointer exceptions, operations that calculate addresses to access r, g, and b are not approximable. A single annotation thus suffices to relax the accuracy of four variables and nine operations. Our language-compiler codesign alleviates the need to manually annotate all these variables and operations.

Control flow safety. To avoid unexpected control flow, FLEXJAVA keeps all the computation and data that affects control flow precise by default. Consider the following example:

```
int fibonacci(int n) {
  int r;
  if (n <= 1)
    r = n;
  else
    r = fibonacci(n - 1) + fibonacci(n - 2);
  loosen(r);
  return r;  }
```

Variable r is annotated as an approximate output and n affects r. But since n also affects control flow in the conditional, it is not safe to approximate.

In many cases, conditionals represent simple control flow that can be converted to data dependence. Programmers can add explicit loosen annotations to mark such conditionals approximate. However, to reduce programmer effort, the FLEXJAVA compiler automatically achieves this effect by conservatively converting control dependencies into data dependencies using a standard algorithm [1]. The following example illustrates this optimization:

```
double sobel(double[][] p){
  double x, y, g, r;
  x = p[0][0] + ...;
  y = p[0][2] + ...;
  g = sqrt(x * x + y * y);
  if (g > 0.7) r = 0.7;
  else r = g;
  loosen(r);
  return r;  }
```

```
double sobel(double[][] p){
  double x, y, g, r;
  x = p[0][0] + ...;
  y = p[0][2] + ...;
  g = sqrt(x * x + y * y);
  r = (g > 0.7) ? 0.7 : g;
  loosen(r);
  return r;  }
```

In the code snippet on the left, by annotating r, there are only a few opportunities for approximation since r depends on g which is used in the conditional. However, the FLEXJAVA compiler can convert this control dependence to data dependence. This conversion is illustrated in the snippet on the right using the ternary ?: operator. After conversion, r is only data dependent on g, which in turn makes g safe to approximate. Consequently, as the snippet on the right shows, all data and operations that affect g are also safe to approximate. As this example shows, this automation significantly increases approximation opportunities without the need for extra manual annotations.

Memory safety. Approximating address calculations may lead to memory access violations or contamination of critical data. To avoid such catastrophic failures and provide memory safety, any computation or data that affects address calculations is precise in FLEXJAVA. Similarly, any computation or data that affects object allocation size is also precise. However, objects that do not contribute to address calculations, allocation sizes, or control flow may be allocated in approximate memory in accordance with the programmer annotations. Consider the following example:

```
int computeAvgRed (Pixel[] pixelArray) {
  int sumRed = 0;
  for(int i = 0; i < pixelArray.length; i++)
    sumRed = sumRed + (int) pixelArray[i].r;
  int avgRed = sumRed / pixelArray.length;
  loosen(avgRed); return avgRed;  }
```

Variables i and pixelArray are not approximable since they are used for address calculations. But the contents of the Pixel objects pointed to by the pixelArray elements, e.g., pixelArray[i].**r**, are approximable due to loosen(avgRed). As discussed before, programmers can always override the default semantics and relax these strict safety guarantees.

Restricting approximation. FLEXJAVA provides the tighten annotation which is dual to loosen. Annotating a variable with tighten makes any data or operation that affects the variable precise, *unless* a preceding loosen makes a subset of those data and operations approximable. The following examples illustrate the interplay between loosen and tighten:

```
float computeAvg (Pixel p) {
 float sum= p.r + p.g + p.b;
 tighten(sum);
 float avg = sum / 2.0f;
 loosen(avg); return avg; }
```

```
float computeAvg (Pixel p) {
 float sum= p.r + p.g + p.b;
 loosen(sum);
 float avg = sum / 2.0f;
 tighten(avg); return avg; }
```

In the left example, we relax the accuracy of data and operations that affect avg except those that affect sum. Conversely, in the right example, we relax the accuracy of data and operations that affect sum while keeping the last step of computing avg precise. The FLEXJAVA compiler automatically introduces tighten annotations to prevent approximating control flow and address calculations. The tighten annotation could also be used by programmers when critical data and operations are intertwined with their approximate counterparts. No such cases appeared when annotating the evaluated benchmarks (Section 6.1).

3.2 Modular Approximate Programming

Scoped approximation. Modularity is essential when designing a language since it enables reusability. To make approximate programming with FLEXJAVA modular, we define a scope for the loosen annotation. The default scope is the code block that contains the annotation; e.g., the function or the loop body within which the loosen annotation is declared. As the following example illustrates, data and operations that are outside of the scope of the loosen annotation are not affected.

```
int p = 1;
for (int i = 0; i < a.length; i++)
  p *= a[i];
for (int i = 0; i < b.length; i++) {
  p += b[i];
  loosen(p);  }
```

Since loosen(p) is declared in the second loop that process the b array, the operations outside of this loop (e.g., p *= a[i]) are not affected and cannot be approximated. Assigning scope to the loosen annotation provides separation of concerns. That is, the loosen annotation only influences a limited region of code that makes it easier for programmers to reason about the effects of the annotation. Furthermore, the scope of approximation adheres to the syntactic structure of the program that makes annotating the code a natural part of the program development.

To ensure safety, the scope for the tighten annotation is the entire program. All data and operations in the program that affect the annotated variable in tighten will be precise. The same principle applies to the conditionals and pointers. The FLEXJAVA compiler automatically applies these global semantics and relieves programmers from safety concerns.

Reuse and library support in FLEXJAVA. Composing independently developed codes to build a software system is a vital part of development. Composability must be supported for the annotations. To this end, we define two variants for the loosen; the default case and the invasive case (loosen_invasive). These variants have different semantics when it comes to function calls. If a function call is in the scope of a loosen annotation and its results affects the annotated variable, it may be approximated only if there are loosen annotations within the function. In other words, the caller's annotations will not interfere with the annotations within the callee and may only enable them. If the callee does not affect caller's annotated variable, its internal loosen annotations will not be enabled. With this approach, the library developers can develop general approximate libraries independently regardless of the future specific use cases. The users can use these general libraries without concerning themselves with the internal annotations of the libraries. The following examples demonstrate the effects of loosen for function calls.

```
static int square(int a){
  int s = a * a;
  loosen(s);
  return s; }
public static void main
        (String[] args){
  int x = 2 + square(3);
  loosen(x);
  System.out.println(x);  }
```

```
static int square(int a){
  int s = a * a;
  loosen(s);
  return s; }
public static void main
        (String[] args){
  int x = 2 + square(3);

  System.out.println(x);  }
```

In the left example, as highlighted, loosen(x) declares the local operations with the main function as safe-to-approximate. The annotation also enables approximation in the square function that was called in the scope of the loosen(x) annotation. Within the square function, the approximation will be based on the annotations that are declared in the scope of square. As the right example illustrates, if there are no loosen annotations in the caller function, main, nothing will be approximated in the callee function, square.

An expert user may want to apply approximation to the callee functions even if they do not contain any internal annotations. FLEXJAVA provides the loosen_invasive for such cases. The loosen_invasive enables applying approximation to the conventional libraries that are not annotated for approximation. Note that loosen_invasive does not cause control flow or memory address calculations to be approximated as we discussed for loosen. The only difference is how approximation is enforced in the callee function as illustrated below.

```
static int square(int a){
  int s = a * a;
  return s; }
public static void main
        (String[] args){
  int x = 2 + square(3);
  loosen(x);
  System.out.println(x);  }
```

```
static int square(int a){
  int s = a * a;
  return s; }
public static void main
        (String[] args){
  int x = 2 + square(3);
  loosen_invasive(x);
  System.out.println(x);  }
```

In the left example, the loosen(x) annotation approximates the local operations in main function but will not lead to any approximation in the square function since it does not contain any loosen annotations. In contrast, in the right example, loosen_invasive(x) enforces safe approximation in square since its return value affects x.

Supporting separate compilation. FLEXJAVA supports separate compilation [7]. That is, a FLEXJAVA program can link with both annotated and unannotated pre-compiled code without having to re-compile it. If the precompiled code is not annotated, it executes precisely. If the precompiled code is annotated, its annotations are respected and its data and operations are approximated accordingly. Moreover, the annotations in the new program will not approximate any additional operations and data in the precompiled code other than the ones already approximated by annotations in them.

3.3 OO Programming in FLEXJAVA

To this point, we have described how to use FLEXJAVA annotations to identify approximate data and operations within methods of a class. This section describes how to declare class fields as approximate and how inheritance and polymorphism interplay with the annotations.

Approximating class fields. Since class fields are not declared in the scope of any of the methods, we allow the programmers to *selectively* relax their semantics in the constructor of the class. The fields will be allocated in the approximate section of the memory if an *outer-level* loosen enables approximation in the constructor. In principle, instantiation of an object involves a function call to the constructor. The outer-level loosen annotations have the same effect on constructors as they have on other function calls.

```
class A {
  float x, y;
  A (float x, float y) {
    this.x = x;
    this.y = y;
    loosen(x); }
  public static void main() {
    A a = new A(1.5f, 2.0f);
    float p = 3.5f + a.x;
    loosen(p);  }  }
```

The annotated p is affected by the instance of A. Therefore, loosen(p) enables approximation in the constructor. Consequently, the x field will be allocated in the approximation section of the memory because of the loosen(x) in the constructor. The y field will not be allocated in the approximation section since it is not annotated in the constructor.

Inheritance. When inheriting an annotated class, annotations are preserved in methods that are not overridden. Naturally, if the child class overrides a method, the overriding method must be re-annotated if approximation is desired.

Polymorphism due to approximation. Depending on the annotations, different instances of a class and different calls to a method may carry approximate or precise semantics. The FLEXJAVA compiler generates different versions of such classes and methods using code specialization [9].

3.4 Generality in FLEXJAVA: Support for Coarse-Grained Approximation

The annotations discussed so far enable fine-grained approximation at the level of single operations and data. This section describes another form of annotations, the begin_loose–end_loose pair, that enables coarse-grained approximation in FLEXJAVA. Any arbitrary code block that is enclosed between this pair of annotations can be approximated as a whole. Both annotations have a variable argument list. The first argument of begin_loose, which is a string, identifies the type of approximation that can be applied to the code block. The compiler or the runtime system then can automatically apply the corresponding approximation technique. Some approximation techniques may require programmers to provide more information. For example, function substitution [3] requires the programmer to provide an approximate version of the function. This extra information can be passed to the compiler or runtime system through the arguments of begin_loose or end_loose. This approach is flexible enough to enable a variety of coarse-grained approximation techniques. We describe how to use the approach with two such techniques: loop perforation [33] and NPUs [12, 16, 21].

Loop perforation. Loop perforation [33] allows the runtime to periodically skip iterations of loops. The programmer can set the initial rate of perforation (skipping the iterations). FLEXJAVA annotations can be used for loop perforation as the following example shows.

```
begin_loose("PERFORATION", 0.10);
  for (int i = 0; i < n; i++)  { ... }
end_loose();
```

The begin_loose("PERFORATION", 0.10) and end_loose() annotations identify the loop that can be approximated. The first argument of begin_loose, "PERFORATION", declares that the desired approximation technique is loop perforation. The second argument, 0.10, identifies the rate of perforation.

Neural acceleration. Neural Processing Units (NPU) [2, 12, 13, 16, 21] are a new class of accelerators that replace compute-intensive functions with hardware neural networks. We give an overview of the NPU compilation workflow since we use them to evaluate FLEXJAVA's coarse-grained annotations. The compiler first automatically trains a neural network on how an approximable code block behaves. Then, it replaces the original block with an efficient hardware implementation of the trained neural network or the NPU. This automatic code transformation also identifies the inputs and outputs of the region. The compiler performs the transformation in four steps:

1. **Input/output identification.** To train a neural network to mimic a code block, the compiler needs to collect the input-output pairs that represent the functionality of the block. Therefore, the first step is identifying the inputs and outputs of the delineated block. The compiler uses a combination of live variable analysis and Mod/Ref analysis [4] to automatically identify the inputs and outputs of the annotated block. The inputs are the intersection of live variables at the location of begin_loose("NPU") with the set of variables that are referenced within the segment. The outputs are the intersection of live variables at the location of end_loose() with the set of variables that are modified within the segment. In the example that follows, this analysis identifies x and y as the inputs to the block and p and q as the outputs.
2. **Code observation.** The compiler instruments the program by putting probes on the inputs and outputs of the block. Then, it profiles the instrumented program using representative input datasets such as those from a test suite. The probes log the block inputs and outputs. The logged input–output pairs form the training dataset.
3. **Training.** The compiler uses the collected input–output dataset to configure and train a multilayer perceptron neural network that mimics the approximable block.
4. **Code generation.** Finally, the compiler replaces the original block with a series of special instructions that invoke the NPU hardware, sending the inputs and receiving the computed approximate outputs.

The following example illustrates the use of FLEXJAVA annotations for NPU acceleration.

```
Double foo(Double x, Double y) {
  begin_loose("NPU");
    p = Math.sin(x) + Math.cos(y);
    q = 2 * Math.sin(x + y);
  end_loose();
  return p + q;  }
```

The programmer uses begin_loose–end_loose to indicate that the body of function foo is a candidate for NPU acceleration. The first argument of begin_loose("NPU") indicates that the approximation technique is NPU acceleration.

```
package edu.flexjava;
abstract class QualityMetric {
  double acceptableQualityLoss = 0.0;
  QualityMetric(double q) { acceptableQualityLoss = q; }
  abstract void checkQuality(Object... o);
  abstract void recover(Object... o);
}
```

Figure 2: An abstract class for defining the quality metric.

```
package edu.flexjava;
class FlexJava {
  static void loosen(Object... o) {}
  static void loosen_invasive(Object... o) {}
  static void tighten(Object... o) {}
  static void begin_loose(String type, Object... o) {}
  static void end_loose(Object... o) {}
}
```

Figure 3: FLEXJAVA annotations are implemented as a library.

3.5 Support for Expressing Quality Metrics, Quality Requirements, and Recovery

Practical and complete approximate programming languages need to provide a mechanism to specify and express quality metrics, quality requirements, and recovery mechanisms. As shown in prior works on approximation, quality metrics are application dependent [3, 11, 12, 30, 33]. For example, an image processing application may use signal-to-noise ratio as the quality metric, while the quality metric for web search is relevance of the results to the search query. The quality metric for machine learning algorithms that perform classification is the misclassification rate. Consequently, the common practice in approximate computing is for programmers to specify the application quality metric and the acceptable level of quality loss. The FLEXJAVA annotations can be naturally extended to express quality metrics and requirements.

As Figure 2 shows, we first provide an abstract class as a template for implementing the quality metric function. The programmer can implement this abstract class and override the checkQuality function to implement the quality metric. The constructor of this class can be used to set the acceptable level of quality loss, acceptableQualityLoss. The programmer can also override the recover to implement a recovery procedure for the occasions that the quality loss is greater than the requirements. Note that the quality requirement can be expressed as a probability if desired. After implementing the QualityMetric class, the programmer can pass its instance via the last argu-

$$\begin{array}{rlrl} (\textit{real constant}) & r \in \mathbb{R} & (\textit{variable}) & v \in \mathbb{V} \\ (\textit{real expression}) & e \in \mathbb{R} \cup \mathbb{V} & (\textit{operation label}) & l \in \mathbb{L} \end{array}$$
$$(\textit{statement})\ s ::= v :=^{l} \delta(e_1, e_2) \mid \mathsf{loosen}(v) \mid \mathsf{tighten}(v) \mid \mathsf{assume}(v) \mid s_1; s_2 \mid s_1 + s_2 \mid s^*$$

Figure 4: Language syntax.

$$\begin{array}{rl} (\textit{stack}) & \rho \in \mathbb{V} \to \mathbb{R} \qquad (\textit{tainted set})\ T \subseteq \mathbb{V} \\ (\textit{state}) & \omega ::= \langle s, \rho, T \rangle \mid \langle \rho, T \rangle \mid \mathsf{error} \mid \mathsf{halt} \end{array}$$

Figure 5: Semantic domains.

ment of loosen, loosen_invasive, or end_loose to the compiler or the runtime system. Clearly, the programmer need not specify the quality metric in each such annotation; it is usually specified only when annotating the final output or important functions of the application, as illustrated in the following example.

```
static int cube (int x) {
  int y = x * x * x;
  loosen(y);
  return y;  }
public static void main (String[] args) {
  int z = cube(7);
  loosen(z, new ApplicationQualityMetric(0.10));
  System.out.println(z);  }
```

Notice that the quality requirement is not specified in the function or library annotations (loosen(y)). It is specified only in the last annotation on the final output z of the program. In this example, the acceptable quality loss is 10%, which is passed to the constructor as 0.10.

4. FLEXJAVA IMPLEMENTATION

FLEXJAVA is a small set of extensions to Java that enables safe, modular, general, and scalable object-oriented approximate programming. It achieves these goals by introducing only four intuitive annotations: loosen, tighten, loosen_invasive, and the begin_loose–end_loose pair. In this section, we describe our implementation of these annotations and the development environment of FLEXJAVA.

Implementation of annotations. We implemented FLEXJAVA annotations as a library to make it compatible with Java programs and tools. Figure 3 illustrates this library-based implementation that provides the interface between the FLEXJAVA language and compiler. The FlexJava class implements the annotations as empty variable-length argument functions. Consequently, compiling a FLEXJAVA program with a traditional compiler yields a fully precise executable. The approximation-aware compiler, however, can intercept calls to these functions and invoke the necessary analyses and approximate transformations.

Integrated highlighting tool. FLEXJAVA is coupled with a static approximation safety analysis that automatically infers the safe-to-approximate operations and data from the programmer annotations. We have developed an integrated tool that highlights the source code with the result of this analysis. By visualizing the result, this tool further facilitates FLEXJAVA programming and can help programmers to refine their annotations. In its current form, the integrated tool adds comments at the end of each line showing which of the line's operations are safe to approximate. It is straightforward to convert this visual feedback to syntax highlighting. In fact, we used the result of this tool to highlight the examples in Section 3..

5. APPROXIMATION SAFETY ANALYSIS

In this section, we define the formal semantics of approximation safety for annotated programs in FLEXJAVA. We define a core language with loosen and tighten annotations. We give a concrete semantics parameterized by the set of operations to be approximated in an annotated program in the language. The semantics determines if a given set of operations is approximable. As this problem is undecidable, we develop a static analysis that conservatively infers the largest set of approximable operations in a given annotated program.

5.1 Core Language

Figure 4 shows the syntax of our core language. It supports real-valued data and control-flow constructs for sequential composition, branches, and loops. We elide conditionals in branches and loops, executing them nondeterministically and using the assume(v) construct that halts if $v \leq 0$.

We extend the language with annotations loosen(v) and tighten(v). These annotations arise from their source-level counterparts described in Section 3.. Further, tighten(v) is implicitly added by the FLEXJAVA compiler before each use of variable v in a conditional, an array index, a pointer dereference, or a program output. To statically identify operations that are approximable under the given annotations, each operation has a unique label l.

Example. We illustrate the above concepts for the program on the left below. For now, ignore the sets in annotations next to each line of the program.

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

The compiler introduces tighten(v1) on lines 3 and 7 to ensure that v1 > 0 executes precisely, and tighten(v2) on line 10 to ensure that the value of v2 output on line 11 is precise. The programmer relaxes the accuracy of v2 on line 9, which allows the operations writing to v2 on lines 2 and 6 to be approximated without violating the tighten(v2) requirement on line 10. However, the operations writing to v1 on lines 1 and 5 cannot be approximated as they would violate the tighten(v1) requirement on line 3 or 7, respectively. □

5.2 Concrete Semantics

We define a concrete semantics to formalize approximation safety for our language. Figure 5 shows the semantic domains. Each program state ω (except for special states error and halt described below) tracks a *tainted set* T of variables. A variable gets tainted if its value is affected by an approximate operation, and untainted if loosen is executed on it.

Figure 6 shows the semantics as a set of rules of the form:

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

It describes an execution of annotated program s when the set of approximated operations is L, starting with stack (i.e., valuation to variables) ρ_1 and tainted set T_1. The rules are similar to information flow tracking: approximated operations in L are *sources* (rule ASGN), loosen(v) are *sanitizers* (rule LOOSEN), and tighten(v) are *sinks* (rules TIGHTENPASS and TIGHTENFAIL). The execution ends in state error if some tighten(v) is executed when the tainted set contains v, as described by rule TIGHTENFAIL. The execution may also end in state halt, which is normal and occurs when assume(v) fails (i.e., $v \leq 0$), as described by

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

$$L \models \langle \mathsf{loosen}(v),\rho,T\rangle \rightsquigarrow \langle \rho, T \setminus \{v\}\rangle \quad \text{(LOOSEN)}$$

$$L \models \langle \mathsf{tighten}(v),\rho,T\rangle \rightsquigarrow \langle \rho,T\rangle \quad [\text{if } v \notin T] \quad \text{(TIGHTENPASS)}$$

$$L \models \langle \mathsf{tighten}(v),\rho,T\rangle \rightsquigarrow \mathsf{error} \quad [\text{if } v \in T] \quad \text{(TIGHTENFAIL)}$$

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

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

Figure 6: Concrete semantics of approximation safety.

rules ASMPASS and ASMFAIL. We omit the rules for compound statements and those that propagate error and halt, as they are relatively standard and do not affect the tainted set.

We now define approximation safety formally:

Defn 5..1 (Approximation safety) *A set of operations L in a program s is approximable if* $\forall \rho : L \models \langle s,\rho,\emptyset\rangle \not\rightsquigarrow \mathsf{error}$.

To maximize approximation, we seek as large a set of approximable operations as possible. In fact, a unique largest set exists, as our semantics satisfies the property that if operation sets L_1 and L_2 are approximable, then so is $L_1 \cup L_2$.

Example. In the example program, the largest set of approximable operations is those on lines 2 and 6. Column `L={2,6}` shows the tainted set as per our semantics after each statement under this set of approximated operations. The error state is unreachable in any run as the tainted set at each $\mathsf{tighten}(v)$ does not contain v. Hence, this set of operations is approximable. □

5.3 Static Analysis

The problem of determining if a given set of operations is approximable in a given annotated program even in our core language is undecidable. We present a novel static analysis that *conservatively* solves this problem, i.e., if the analysis deems a set of operations as approximable, then it is indeed approximable according to Defn. 5..1. Further, we apply an efficient algorithm that uses the analysis to automatically infer the largest set of approximable operations.

Our static analysis is shown in Figure 7. It *over-approximates* the tainted sets that may arise at a program point in the concrete semantics by an abstract state D, a set each of whose elements is $\top$ or an *abstract tainted set* π of variables.

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

Theorem 5..2 (Soundness) *For each program s, if* $\top \notin F_L[s](\{\emptyset\})$ *then for each state ρ,* $L \models \langle s,\rho,\emptyset\rangle \not\rightsquigarrow \mathsf{error}$.

Example. For our example from Section 5.1, the columns on the right show the abstract state computed by the analysis after each statement, 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 the operations on lines 1 and 5 are not approximable. But for `L={2,6}`, the final abstract state does not contain $\top$, proving that operations on lines 2 and 6 are approximable. □

$$\begin{array}{rcl} (\textit{abstract tainted set}) \quad \pi & \in & \Pi = 2^{\mathbb{V}} \\ (\textit{abstract state}) \quad D & \subseteq & \mathbb{D} = \Pi \cup \{\top\} \end{array}$$

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

for atomic statement t, where:

$$\begin{array}{rcl} \mathsf{trans}_L[t](\top) & = & \top \\ \mathsf{trans}_L[v :=^{l} \delta(e_1,e_2)](\pi) & = & \begin{cases} \pi \cup \{v\} \text{ if } l \in L \vee \\ \qquad uses(e_1,e_2) \cap \pi \neq \emptyset \\ \pi \setminus \{v\} \text{ otherwise} \end{cases} \\ \mathsf{trans}_L[\mathsf{tighten}(v)](\pi) & = & \begin{cases} \pi \text{ if } v \notin \pi \\ \top \text{ otherwise} \end{cases} \\ \mathsf{trans}_L[\mathsf{loosen}(v)](\pi) & = & \pi \setminus \{v\} \end{array}$$

Figure 7: Approximation safety analysis.

Our static analysis has the useful property that for any annotated program, there exists a unique largest set of operations that it considers approximable.

Theorem 5..3 (Unique largest solution) $\exists\ L_{max} \subseteq \mathbb{L} : \top \notin F_{L_{max}}[s](\{\emptyset\}) \wedge (\top \notin F_L[s](\{\emptyset\}) \Rightarrow L \subseteq L_{max})$.

We use a standard algorithm [40] to infer this largest set of approximable operations. Starting with all operations approximated, it iteratively finds a largest set of approximable operations which passes all the tighten checks in the program.

6. FLEXJAVA EVALUATION

This section aims to answer the following research questions.

- **RQ1**: Can FLEXJAVA significantly reduce the number of manual annotations?
- **RQ2**: Can FLEXJAVA significantly reduce the programmer effort and annotation time?
- **RQ3**: Can FLEXJAVA give significant speedup and energy gains with both fine- and coarse-grained approximation?

As the results of the evaluations show, FLEXJAVA reduces the number of annotations (between 2× and 17×) compared to EnerJ, the leading approximation language. We also conduct a user study that shows from 6× to 12× reduction in annotation time compared to EnerJ. FLEXJAVA; however, provides the same level of energy savings (from 7% to 38%) compared to EnerJ with fine-grained approximation. With coarse-grained approximation, FLEXJAVA achieves 2.2× energy reduction and 1.8× speedup for under 10% quality loss.

Benchmarks and quality metrics. As Table 2 shows, we evaluate FLEXJAVA using 10 Java programs. Eight are the EnerJ benchmarks [30]. We use two additional benchmarks, hessian and sobel. Five of these come from the SciMark2 suite. The rest are zxing, an Android bar code recognizer; jmeint, an algorithm to detect intersecting 3D triangles (part of the jMonkeyEngine game engine); sobel, an edge detection application based on the Sobel operator; and raytracer, a simple 3D ray tracer. To better study the scalability of our analysis, we added the hessian application from the BoofCV vision library with 10,174 lines of code. This application uses the Hessian affine region detector to find interesting points in an image. The code for this application uses Java generics that is not supported by the EnerJ compiler and simulator. However, our safety analysis supports Java generics and was able to analyze this application. Therefore, only for this specific application, our comparisons are limited to annotation effort and safety analysis. Table 2 also shows the application-specific quality metrics. We measure quality by comparing the

Table 2: Benchmarks, quality metrics, and results of safety analysis: analysis runtime and # of approximable data and operations.

	Description	Quality Metric	# of Lines		Analysis Runtime (sec)	# of Approximable Data		# of Approximated Operations	
			Bench	Library		Inferred	Potential	Inferred	Potential
sor	SciMark2 benchmark: scientific kernels	Avg entry difference	36	60K	6	8	14	133	282
smm		Avg normalized difference	38	60K	6	9	17	114	278
mc		Normalized difference	59	60K	4	7	13	129	184
fft		Avg entry difference	168	60K	11	11	14	226	485
lu		Avg entry difference	283	60K	15	12	19	201	600
sobel	Image edge detection	Avg pixel difference	163	284K	102	2	5	153	416
raytracer	3D image renderer	Avg pixel difference	174	214K	14	4	9	128	264
jmeint	jMonKeyEngine game: triangle intersection kernel	Percents of correct decisions	5,962	216K	296	71	71	832	943
hessian	Interest point detection in BoofCV library	Avg Euclidean distance	10,174	261K	6,228	73	119	663	4988
zxing	Bar code decoder for mobile phones	Percents of correct results	26,171	271K	12,722	996	1,053	2,673	8,454

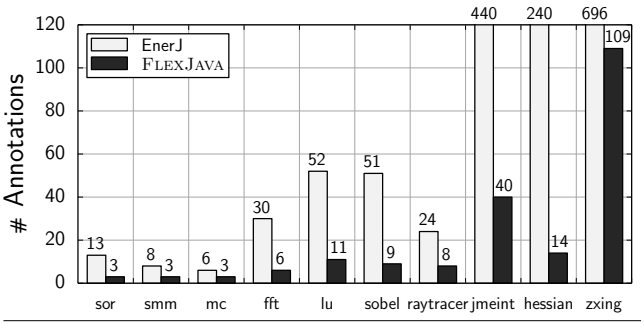

Figure 8: Number of annotations required to approximate the same set of data and operations using EnerJ and FlexJava.

output of the fully precise and the approximated versions of the program. For each benchmark, we use 10 representative input datasets such as 10 different images. The quality degradation is averaged over the input datasets.

6.1 RQ1: Number of Annotations

To answer RQ1, we compare the number of EnerJ annotations with FlexJava annotations. We use EnerJ as a point of comparison because it requires the minimum number of annotations among existing approximate languages [6, 30]. EnerJ requires programmers to annotate all the approximate data declarations using type qualifiers. Then, the EnerJ compiler infers the safe-to-approximate operations for fine-grained approximation. In contrast, our approximation safety analysis *infers* both approximate data and operations from a limited number of FlexJava annotations on the program or function outputs. We used the Chord program analysis platform [23] to implement our approximation safety analysis. Compared to EnerJ, our analysis infers at least as many number of safe-to-approximate data and operations with significantly fewer number of manual annotations.

Figure 8 shows the number of annotations with EnerJ and FlexJava. As Figure 8 illustrates, there is a significant reduction in the number of annotations with FlexJava. FlexJava requires between 2× (mc) to 17× (hessian) less annotations than EnerJ. The largest benchmark in our suite is zxing with 26,171 lines of code. It requires 696 annotations with EnerJ, 109 annotation with FlexJava. Thus, FlexJava reduces the number of annotations by a factor of 6×. The zxing benchmark needs several loosen annotations to mark its function outputs as approximable. Further, many condition variables are safe to approximate and such variables need to be annotated explicitly. Therefore, zxing requires a number of FlexJava annotations that is relatively large compared to all other benchmarks. These results confirm that FlexJava annotations and its approximation safety analysis can effectively reduce the number of manual annotations.

The results in Figure 8 are with no use of loosen_invasive. Using loosen_invasive only reduces the number of annotations with FlexJava. Moreover, in the evaluated benchmarks, there is no need for any manual tighten annotations. As described before, FlexJava's approximation safety analysis automatically inserts tighten annotations for the critical variables to ensure control flow and memory safety. The FlexJava highlighting tool was useful since it effectively visualizes the result of the automated approximation safety analysis.

Approximation safety analysis. In Table 2, columns "# of Lines" and "Analysis Runtime (sec)" report the number of lines in each program and the runtime of the approximation safety analysis. The analysis analyzes application code and reachable Java library (JDK) code uniformly although we report their sizes separately in the table. The analysis was performed using Oracle HotSpot JVM 1.6.0 on a Linux machine with 3.0 GHz quad-core processors and 64 GB memory.

The analysis runtime strongly correlates with the number of potentially approximable data and operations. The potential approximable elements include all the data declarations and all the operations that are not address calculations and jump or branch instructions in the byte code. The number of potential elements is presented in columns "# of Approximable Data-Potential" and "# of Approximable Operations-Potential", respectively. The analysis determines whether or not each of these elements is safe to approximate with respect to the programmer annotations. The number of all the potential approximable elements defines the search space of the analysis. Thus, the space of possible solutions that the approximation safety analysis explores for zxing is of size $2^{(1053+8454)}$. Automatically finding the largest set of approximable elements from this huge space justifies the 12,722 seconds (=3 hours and 32 minutes) of running time to analyze zxing. However, the analysis runtime is not exponential with respect to the number of potential elements. That is because in each iteration, the analysis eliminates at least one element from the potentials list.

Naturally, significantly reducing the number of manual annotations requires an automated analysis that takes some machine time. That is, the analysis is trading machine time for fewer annotations, potentially saving programmer time. Furthermore, we report the pessimistic runtime when all of the libraries and program codes are analyzed in a single compiler run without separate compilation. Separate compilation may reduce this runtime when precompiled approximate libraries are available.

6.2 RQ2: Programmer Effort/Annotation Time

To answer RQ2, we conduct a user study involving ten programmers. The programmers are asked to annotate three programs with both languages. To avoid bias in our study toward FlexJava, we used three programs from the EnerJ benchmark suite [29]. The benchmarks are not large so that the subjects can understand their functionality before annotating them. As presented in Figure 9, we measure the annotation time with EnerJ and FlexJava and compare the results. The subjects are computer science graduate students who have prior background

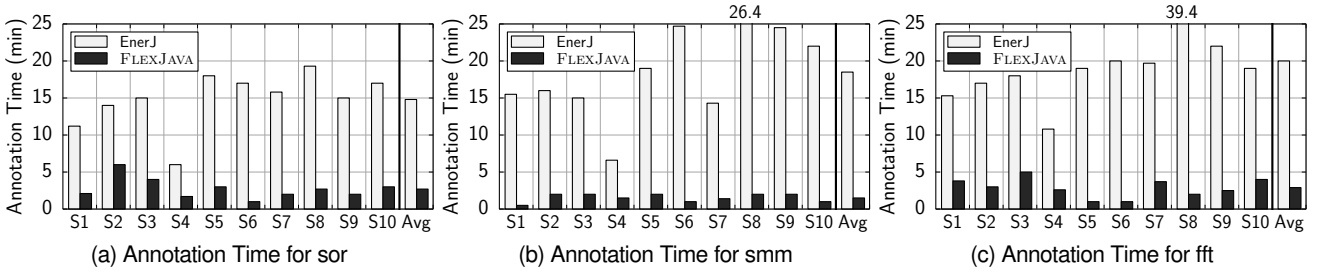

Figure 9: Annotation time with EnerJ and FlexJava for (a) sor, (b) smm, and (c) fft. The x-axes denotes the user study subjects.

in Java programming but have no experience in approximate programming. We measured the annotation time using the following procedure.

First, we orally explain how to annotate the programs with FlexJava and EnerJ. Then, we demonstrate the annotation process on a simple benchmark, mc, and show the subjects how to use the tools for both languages. For this study, the subjects then annotate three of benchmarks, sor, smm, and fft, using both languages. Half of the subjects use EnerJ annotations first and the other half use FlexJava first. The measured time for EnerJ constitutes annotation plus compilation time. Whereas the measured time for FlexJava constitutes annotation time, plus the time for running the approximation safety analysis, plus the time for analyzing the analysis results using the source highlighting tool. We provide the unannotated application and a description of its algorithm for the subjects. We allow the subjects to review each application code prior to annotating it. Our current highlighting tool is enough to check whether or not the analyzed results are equivalent between the two languages.

Figure 9 shows the annotation time. On average the annotation time with FlexJava is 6×, 12×, 8× less than EnerJ for sor, smm, and fft, respectively. Although we demonstrate how the subjects can use the languages, they need time to gain experience while annotating the first program. Once the subjects acclimate to FlexJava with the first benchmark (sor), they spend proportionally less time annotating the next benchmark. The FlexJava annotation time for the second benchmark (smm) is typically lower than the first benchmark (sor). In contrast, the annotation time with EnerJ does not reduce beyond a certain point even after gaining experience. We believe that this is because EnerJ requires manually annotating all the approximate variable declarations and more. Using FlexJava, sor and smm require three loosen annotation, but fft requires six. We believe that this explains why the time to annotate fft in FlexJava is greater than the time to annotate sor and smm. In summary, these results show that FlexJava significantly reduces programmer effort by providing intuitive language extensions and leveraging the automated approximation safety analysis.

6.3 RQ3: Energy Reduction and Speedup

To answer RQ3, we study energy gains and speedup of FlexJava with both fine- and coarse-grained approximation.

6.3.1 Fine-Grained Approximation

Tools and models. We modify and use the EnerJ open-source simulator [30] for error and energy measurements. The simulator provides the means to instrument Java programs based on the result of the analysis. It allows object creation and destruction in approximate memory space and approximating arithmetic and logic operations. The runtime simulator is a Java library that is invoked by the instrumentation. The simulator records memory-footprint and arithmetic-operation statistics while simultaneously injecting error to emulate approximate execution

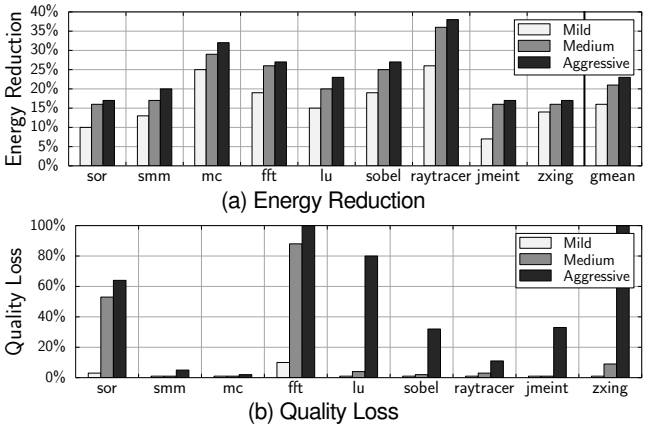

Figure 10: (a) Energy reduction and (b) quality loss when approximating all the safe-to-approximate data and operations.

and measure error. The simulator uses the runtime statistics to estimate the amount of energy dissipated by the program. The error and energy measurements are based on the system models described in Table 1. The models and the simulator do not support performance measurements. We measured the error and energy usage of each application over ten runs and average the results.

Figure 10 shows the energy reduction and the output quality loss when the safe-to-approximate data and operations are approximated. These results match those of EnerJ [29]. As shown, the geometric mean of energy reduction ranges from 16% with the Mild hardware setting to 23% with the Aggressive hardware setting. The energy reduction is least for jmeint (7% with Mild) and highest for raytracer (38% with Aggressive). All the applications show low and acceptable output quality loss with the Mild setting. However, in most cases, there is a jump in quality degradation when the hardware setting is changed to Aggressive. If this level of quality is not acceptable (fft), then the application should dial down the hardware setting to Medium or Mild. FlexJava provides the same level of benefits and quality degradations as EnerJ while significantly reducing the number of manual annotations.

6.3.2 Coarse-Grained Approximation

To evaluate FlexJava's generality, we use the NPU coarse-grained approximation [12]. NPU can only be used to approximate the benchmarks fft, sobel, raytracer, and jmeint. Each benchmark has only one function that can be approximated with NPUs. Each of these functions can be delineated using a single pair of begin_loose–end_loose annotation.

Tools and models. We measure the benefits of NPUs in conjunction with a modern Intel Nehalem (Core i7) processor. We use a source-to-source transformation that instruments the

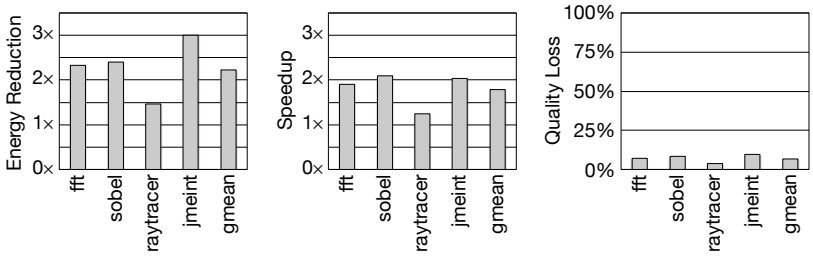

Figure 11: Speedup, energy reduction, and output quality loss when the approximate annotated functions using the NPU.

benchmarks' Java code to emit an event trace including memory accesses, branches, and arithmetic operations. This source-level instrumentation is unaffected by the JIT, garbage collection, or other VM-level systems. Using a trace-based simulator, we generate architectural event statistics. The architectural simulator includes a cache simulation. The simulation process outputs detailed statistics, including the cycle count, cache hit and miss counts, and the number of functional unit invocations. The trace-based CPU simulator is augmented with a cycle-accurate NPU simulator that also generates the statistics required for the NPU energy estimation. The resulting statistics are sent to a modified version of McPAT [18] to estimate the energy consumption of each execution. We model the energy consumption of an eight-processing-engine NPU using the results from CACTI 6.5 [22], McPAT [18], and [14].

Figure 11 shows the energy reduction, speedup, and quality loss with the NPU coarse-grained approximation. The baseline executes the precise version of the benchmark on the CPU without any NPU approximation. On average, the benchmarks see a 2.2× energy reduction and a 1.8× speedup. These benefits come for less than 10% quality degradation across all the benchmarks, which is commensurate with other approximation techniques [11,16,27,28] and prior NPU works [2,12,21]. The EnerJ system does not provide any coarse-grained approximation results for comparison.

These results demonstrate that coarse-grained approximation may have limited applicability but can provide higher benefits. Whereas, fine-grained approximation is more widely applicable with possibly lower gains. FlexJava supports both granularities as a general language to maximize opportunities for approximation in a wider range of applications.

7. RELATED WORK

There is a growing body of work on language design, reasoning, analysis, transformations, and synthesis for approximate computing. These works can be characterized based on (1) static vs. dynamic, (2) approximation granularity, (3) automation, and (4) safety guarantees. To this end, FlexJava is a language accompanied with an automated static analysis that supports both fine- and coarse-grained approximation and provides formal safety guarantees. We discuss the related work with respect to these characteristics.

EnerJ [30] is an imperative programming language that statically infers the approximable operations from approximate type qualifiers on program variables. In EnerJ, all approximable variables must be explicitly annotated. EnerJ works at the granularity of instructions and provides safety but not quality guarantees. Rely [6] is another 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 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 FlexJava when it automatically generates the approximate version of the program. Axilog [39] introduces a set of annotations for approximate hardware design in the Verilog hardware description language. Verilog does not support imperative programming constructs such as pointers, structured data, memory allocation, recursion, etc. The lack of these features results in fundamentally different semantics for safe approximation and annotation design.

Chisel [19] uses integer linear programming (ILP) formulation to optimize approximate computational kernels. A Chisel program consists of code written in an imperative language such as C and a kernel function written in Rely that will be optimized. Several works have focused on approximation at the granularity of functions or loops. Loop perforation [20, 32, 33] is an automated static technique that periodically skips loop iterations. Even though loop perforation provides statistical quality guarantees, the technique is not safe and perforated programs may crash. Green [3] 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 monitoring system that adjusts the level of approximation at runtime. Such runtime adjustments are feasible due to the coarse granularity of the approximation. FlexJava provides the necessary language extensions for supporting these coarse-grained approximation techniques as well as the fine-grained ones.

Similar to EnerJ, Uncertain<T> [5] is a type system for probabilistic programs that operate on uncertain data. It implements a Bayesian network semantics for computation on probabilistic data. Similarly, [31] uses Bayesian networks and symbolic execution to verify probabilistic assertions.

8. CONCLUSION

Practical and automated programming models for approximation techniques are imperative to enabling their widespread applicability. This paper described one such language model that leverages automated program analysis techniques for more effective approximate programming. The FlexJava language is designed to be intuitive and support essential aspects of modern software development: safety, modularity, generality, and scalability. We implemented FlexJava and its approximation safety analysis and evaluated its usability across different approximation techniques that deliver significant energy and performance benefits. The results suggest that FlexJava takes an effective and necessary step toward leveraging approximation in modern software development.

9. ACKNOWLEDGMENTS

We thank the reviewer and Kivanç Muşlu for insightful comments. This work was supported by a Qualcomm Innovation Fellowship, NSF award #1253867, Semiconductor Research Corporation contract #2014-EP-2577, and a gift from Google.

10. REPLICATION PACKAGE

The FLEXJAVA tool has been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations. The replication package is available at the following link: `http://act-lab.org/artifacts/flexjava`.

10.1 Overview

The replication package contains the FLEXJAVA compiler that supports fine-grained and coarse-grained approximation. All the benchmarks that we used for experiments in the paper are also included in the package.

For fine-grained approximation, the compiler first takes an input program annotated with FLEXJAVA annotations and performs the approximation safety analysis, which finds safe-to-approximate data and operations. The highlighting tool creates a copy of the input program's source code and marks the analysis results on the copied code in a form of comments. Programmers can use this tool to observe the analysis results and they may remove or add annotations if the results do not meet their expectations. We also provide the modified EnerJ simulator for quality and energy measurement on which FLEXJAVA binaries can be executed.

The FLEXJAVA language and compiler can support arbitrary types of coarse-grained approximation technologies but this replication package provides the NPU framework as an example of its use. NPiler [12] requires programmers to specify a program region so that the compiler can *train* the region with a multi layer perceptron neural network. The FLEXJAVA annotations, begin_loose and end_loose, are capable of delivering all the information needed by NPiler.

10.2 Download Tools

We have created a VHD (Virtual Hard Disk) on VirtualBox [38] so that users can readily download the entire image file and run the experiments without manually installing all the tools and setting up the environment. The link for the VHD file is provided at the aforementioned FLEXJAVA page. Users who wish to just investigate the source code can access it from the Git repositories on the Bitbucket page: `https://bitbucket.org/act-lab`. The Bitbucket page has a detailed README file explaining how to setup and run the tools. The source code embedded in the VHD image file has the same version as that in the Git repositories.

10.3 Instructions

These instructions assume that the user has downloaded the VHD file (VM image file) and built the VM environment using a virtualization tool such as VirtualBox. Users who wish to explore the source code may follow the instructions in the Bitbucket pages to setup the necessary environment.

10.3.1 Fine-grained Approximation

Build Tools We have already installed and built all source code necessary to run the analysis and simulation. Users can find these files in the `r2.code` directory under the home directory. The FLEXJAVA compiler is a combination of several tools together with a series of compilation steps for the tools that can be executed by running a bash script named `build.sh`.

Benchmarks All the benchmarks are placed under the `r2.apps` directory. The source code has already been annotated with FLEXJAVA annotations and it is located under the `src` directory of each individual benchmark's directory.

Running the Approximation Safety Analysis The user can next run the analysis by simply running the `analyze.py` script. This script will (1) compile the source code, (2) run the analysis, and (3) perform the source code highlighting (back annotation) on a replicated source directory, `src-marked`. The user can then observe where approximation was applied in the `src-marked` directory. If unsatisfied with the results, the user may update the annotations under the `src` directory and rerun the analysis.

Running the Simulation If the analysis results are satisfactory, the user may proceed to the next step, *simulation*. The script for running simulation, `runsimulation.py`, takes an argument which specifies a system model to use for the simulation. There are four system models that are supported by the EnerJ simulator: (1) aggressive (2) high (3) medium (4) low. Since the architecture model in the simulator is probabilistic, we ran the experiments multiple times and averaged to obtain the results in the paper. For this reason, the results from the simulation may not exactly match the results provided in the paper.

Adding New Benchmarks Users can add a new benchmark by following these five steps:

1. create a new directory under the `r2.apps` directory, named after the benchmark;
2. place its source code in the `src` directory;
3. copy a `build.xml` file from one of the pre-existing benchmarks into the new directory;
4. replace the benchmark name in the `build.xml` file (e.g., `mc`) with the new one; and
5. create symbolic links for four python scripts, `analyze.py`, `runsimulation.py`, `markJava.py`, and `Input.py`.

10.3.2 Coarse-grained Approximation

Benchmarks The NPU benchmarks and tools used in the paper are based on an approximate computing benchmark suite, AxBench (`http://axbench.org`) and they were ported from C/C++ to Java. The files are under the `flexjava.npubench` sub-directory in the home directory of the VHD image. The `application.java` directory contains the four benchmarks (fft, jmeint, sobel, and simpleRaytracer) that have been evaluated in the paper for coarse-grained approximation (NPU). We have provided a `Makefile` for each individual benchmark that performs the necessary preprocessing and compiles the processed Java source code.

Running NPU Code We have provided a script that (1) trains the specified approximable region, (2) algorithmically transforms the region into a neural network, and (3) runs the transformed program using a machine learning library, FANN [25]. The running script, `run_java.sh`, takes two arguments. For training, provide `run` as the first argument and the benchmark directory name (e.g., sobel) as the second argument. The script will then show the compilation parameters required for training (e.g., learning rate). The user can supply any values to the parameters to override the following default values:

- Learning rate [0.1-1.0]: 0.1
- Epoch number [1-10000]: 1
- Sampling rate [0.1-1.0]: 0.1
- Test data fraction [0.1-1.0]: 0.5
- Maximum number of layers [3|4]: 3
- Maximum number of neurons per layer [2-64]: 2

Adding New Benchmarks The required steps for adding new benchmarks in coarse-grained approximation are similar to those for fine-grained approximation. The current implementation lacks a well-designed interface that enables programmers to readily introduce a new benchmark without modifying the FLEXJAVA makefiles and scripts, which will be updated in the near future.

11. REFERENCES

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *POPL*, 1983.

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

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

[4] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 29–41. ACM, 1979.

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

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

[7] L. Cardelli. Program fragments, linking, and modularization. In *POPL*, 1997.

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

[9] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *PLDI*, 1995.

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

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

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

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. *Communications of the ACM Research Highlights*, 58(1):105–115, Jan. 2015.

[14] S. Galal and M. Horowitz. Energy-efficient floating-point unit design. *IEEE Transactions on Computers*, 60(7):913–922, 2011.

[15] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *HPCA*, 2011.

[16] B. Grigorian, N. Farahpour, and G. Reinman. BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing. In *HPCA*, 2015.

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

[18] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[19] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *OOPSLA*, 2014.

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

[21] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. SNNAP: Approximate computing on programmable socs via neural acceleration. In *HPCA*, 2015.

[22] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO*, 2007.

[23] M. Naik. Chord. `http://jchord.googlecode.com/`.

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

[25] S. Nissen. Implementation of a fast artificial neural network library (fann). Technical report, Department of Computer Science University of Copenhagen (DIKU), 2003. http://leenissen.dk/fann/wp.

[26] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, June 2014.

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

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

[29] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, , and D. Grossman. EnerJ. `http://sampa.cs.washington.edu/research/approximation/enerj.html`.

[30] 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.

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

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

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

[34] B. Thwaites, G. Pekhimenko, H. Esmaeilzadeh, A. Yazdanbakhsh, O. Mutlu, J. Park, G. Mururu, and T. Mowry. Rollback-free value prediction with approximate loads. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 493–494, August 2014.

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

[36] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: Reducing the energy of mature computations. In *ASPLOS*, 2010.

[37] G. Venkatesh, J. Sampson, N. Goulding, S. K. Venkata, S. Swanson, and M. Taylor. QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *MICRO*, 2011.

[38] J. Watson. Virtualbox: Bits and bytes masquerading as machines. *Linux J.*, 2008(166), Feb. 2008.

[39] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmailzadeh,

and K. Bazargan. Axilog: Language support for approximate hardware design. In *DATE*, 2015.

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