EnerJ: Approximate Data Types for Safe and General Low-Power Computation

Adrian Sampson Werner Dietl Emily Fortuna Danushen Gnanapragasam
Luis Ceze Dan Grossman

University of Washington, Department of Computer Science & Engineering
http://sampa.cs.washington.edu/

Abstract

Energy is increasingly a first-order concern in computer systems. Exploiting energy-accuracy trade-offs is an attractive choice in applications that can tolerate inaccuracies. Recent work has explored exposing this trade-off in programming models. A key challenge, though, is how to *isolate parts of the program that must be precise from those that can be approximated* so that a program functions correctly even as quality of service degrades.

We propose using type qualifiers to declare data that may be subject to approximate computation. Using these types, the system automatically maps approximate variables to low-power storage, uses low-power operations, and even applies more energy-efficient algorithms provided by the programmer. In addition, the system can statically guarantee isolation of the precise program component from the approximate component. This allows a programmer to control explicitly how information flows from approximate data to precise data. Importantly, employing static analysis eliminates the need for dynamic checks, further improving energy savings. As a proof of concept, we develop EnerJ, an extension to Java that adds approximate data types. We also propose a hardware architecture that offers explicit approximate storage and computation. We port several applications to EnerJ and show that our extensions are expressive and effective; a small number of annotations lead to significant potential energy savings (7%–38%) at very little accuracy cost.

Categories and Subject Descriptors C.0 [*Computer Systems Organization*]: General—Hardware/software interfaces; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

General Terms Languages, Performance, Design

Keywords Accuracy-aware computing, power-aware computing, energy, soft errors, critical data

1. Introduction

Energy consumption is an increasing concern in many computer systems. Battery life is a first-order constraint in mobile systems, and power/cooling costs largely dominate the cost of equipment in data-centers. More fundamentally, current trends point toward a "utilization wall," in which the amount of active die area is limited by how much power can be fed to a chip.

Much of the focus in reducing energy consumption has been on low-power architectures, performance/power trade-offs, and resource management. While those techniques are effective and can be applied without software knowledge, exposing energy considerations at the programming language level can enable a whole new set of energy optimizations. This work is a step in that direction.

Recent research has begun to explore energy-accuracy trade-offs in general-purpose programs. A key observation is that systems spend a significant amount of energy guaranteeing correctness. Consequently, a system can save energy by exposing faults to the application. Many studies have shown that a variety of applications are resilient to hardware and software errors during execution [1, 8, 9, 19, 21–23, 25, 31, 35]. Importantly, these studies universally show that applications have portions that are more resilient and portions that are "critical" and must be protected from error. For example, an image renderer can tolerate errors in the pixel data it outputs—a small number of erroneous pixels may be acceptable or even undetectable. However, an error in a jump table could lead to a crash, and even small errors in the image file format might make the output unreadable.

While approximate computation can save a significant amount of energy, distinguishing between the critical and non-critical portions of a program is difficult. Prior proposals have used annotations on code blocks (e.g., [9]) and data allocation sites (e.g., [23]). These annotations, however, do not offer any guarantee that the fundamental operation of the program is not compromised. In other words, these annotations are either *unsafe* and may lead to unacceptable program behavior or *need dynamic checks* that end up consuming energy. We need a way to allow programmers to compose programs from approximate and precise components safely. Moreover, we need to guarantee safety statically to avoid spending energy checking properties at runtime. The key insight in this paper is the application of type-based information-flow tracking [32] ideas to address these problems.

Contributions. This paper proposes a model for approximate programming that is both *safe* and *general*. We use a type system that isolates the precise portion of the program from the approximate portion. The programmer must explicitly delineate flow from approximate data to precise data. The model is thus *safe* in that it guarantees precise computation unless given explicit programmer permission. Safety is statically enforced and no dynamic checks are required, minimizing the overheads imposed by the language.

We present EnerJ, a language for principled approximate computing. EnerJ extends Java with type qualifiers that distinguish between *approximate* and *precise* data types. Data annotated with the "ap-

PLDI'11, June 4–8, 2011, San Jose, California, USA.

proximate" qualifier can be stored approximately and computations involving it can be performed approximately. EnerJ also provides *endorsements*, which are programmer-specified points at which approximate-to-precise data flow may occur. The language supports programming constructs for algorithmic approximation, in which the programmer produces different implementations of functionality for approximate and precise data. We formalize a core of EnerJ and prove a non-interference property in the absence of endorsements.

Our programming model is *general* in that it unifies approximate data storage, approximate computation, and approximate algorithms. Programmers use a single abstraction to apply all three forms of approximation. The model is also *high-level and portable:* the implementation (compiler, runtime system, hardware) is entirely responsible for choosing the energy-saving mechanisms to employ and when to do so, guaranteeing correctness for precise data and "best effort" for the rest.

While EnerJ is designed to support general approximation strategies and therefore ensure full portability and backward-compatibility, we demonstrate its effectiveness using a proposed approximation-aware architecture with approximate memory and imprecise functional units. We have ported several applications to EnerJ to demonstrate that a small amount of annotation can allow a program to save a large amount of energy while not compromising quality of service significantly.

Outline. We first detail the EnerJ language extensions in Section 2. Section 3 formalizes a core of the language, allowing us to prove a non-interference property. The full formalism and proof are presented in an accompanying technical report [33]. Next, Section 4 describes hypothetical hardware for executing EnerJ programs. While other execution substrates are possible, this proposed model provides a basis for the evaluation in Sections 5 and 6; there, we assess EnerJ's expressiveness and potential energy savings. The type checker and simulation infrastructure used in our evaluation are available at `http://sampa.cs.washington.edu/`. Section 7 presents related work and Section 8 concludes.

2. A Type System for Approximate Computation

This section describes EnerJ's extensions to Java, which are based on a system of type qualifiers. We first describe the qualifiers themselves. We next explain how programmers precisely control when approximate data can affect precise state. We describe the implementation of approximate operations using overloading. We then discuss conditional statements and the prevention of implicit flows. Finally, we describe the type system's extension to object-oriented programming constructs and its interaction with Java arrays.

EnerJ implements these language constructs as backwards-compatible additions to Java extended with type annotations [11]. Table 1 summarizes our extensions and their concrete syntax.

2.1 Type Annotations

Every value in the program has an approximate or precise type. The programmer annotates types with the @Approx and @Precise qualifiers. Precise types are the default, so typically only @Approx is made explicit. It is illegal to assign an approximate-typed value into a precise-typed variable. Intuitively, this prevents direct flow of data from approximate to precise variables. For instance, the following assignment is illegal:

```
@Approx int a = ...;
int p; // precise by default
p = a; // illegal
```

Approximate-to-precise data flow is clearly undesirable, but it seems natural to allow flow in the opposite direction. For primitive Java types, we allow precise-to-approximate data flow via subtyping. Specifically, we make each precise primitive Java type a subtype of its approximate counterpart. This choice permits, for instance, the assignment `a = p;` in the above example.

For Java's reference (class) types, this subtyping relationship is unsound. The qualifier of a reference can influence the qualifiers of its fields (see Section 2.5), so subtyping on mutable references is unsound for standard reasons. We find that this limitation is not cumbersome in practice.

We also introduce a @Top qualifier to denote the common supertype of @Approx and @Precise types.

Semantics of approximation. EnerJ takes an all-or-nothing approach to approximation. Precise values carry traditional guarantees of correctness; approximate values have no guarantees. A more complex system could provide multiple levels of approximation and guaranteed error bounds, but we find that this simple system is sufficiently expressive for a wide range of applications.

The language achieves generality by leaving approximation patterns unspecified, but programmers can informally expect approximate data to be "mostly correct" and adhere to normal execution semantics except for occasional errors. The degree of precision is an orthogonal concern; a separate system could tune the frequency and intensity of errors in approximate data.

2.2 Endorsement

Fully isolating approximate and precise parts of a program would likely not be very useful. Eventually a program needs to store data, transmit it, or present it to the programmer—at which point the program should begin behaving precisely. As a general pattern, programs we examined frequently had a phase of fault-tolerant computation followed by a phase of fault-sensitive reduction or output. For instance, one application consists of a resilient image manipulation phase followed by a critical checksum over the result (see Section 6.3). It is essential that data be occasionally allowed to break the strict separation enforced by the type system.

We require the programmer to control explicitly when approximate data can affect precise state. To this end, we borrow the concept (and term) of *endorsement* from past work on information-flow control [2]. An explicit static function endorse allows the programmer to use approximate data as if it were precise. The function acts as a cast from any approximate type to its precise equivalent. Endorsements may have implicit runtime effects; they might, for example, copy values from approximate to precise memory.

The previous example can be made legal with an endorsement:

```
@Approx int a = ...;
int p; // precise by default
p = endorse(a); // legal
```

By inserting an endorsement, the programmer certifies that the approximate data is handled intelligently and will not cause undesired results in the precise part of the program.

2.3 Approximate Operations

The type system thus far provides a mechanism for approximating *storage*. Clearly, variables with approximate type may be located in unreliable memory modules. However, approximate *computation* requires additional features.

We introduce approximate computation by overloading operators and methods based on the type qualifiers. For instance, our language provides two signatures for the + operator on integers: one taking two precise integers and producing a precise integer and the other taking two approximate integers and producing an approximate integer. The latter may compute its result approximately and thus may run on low-power hardware. Programmers can extend this concept by overloading methods with qualified parameter types.

Bidirectional typing. The above approach occasionally applies precise operations where approximate operations would suffice.

Construct	Purpose	Section
@Approx, @Precise, @Top	Type annotations: qualify any type in the program. (Default is @Precise.)	2.1
endorse(e)	Cast an approximate value to its precise equivalent.	2.2
@Approximable	Class annotation: allow a class to have both precise and approximate instances.	2.5
@Context	Type annotation: in approximable class definitions, the precision of the type depends on the precision of the enclosing object.	2.5.1
_APPROX	Method naming convention: this implementation of the method may be invoked when the receiver has approximate type.	2.5.2

Table 1. Summary of EnerJ's language extensions.

Consider the expression a = b + c where a is approximate but b and c are precise. Overloading selects precise addition even though the result will only be used approximately. It is possible to force an approximate operation by upcasting either operand to an approximate type, but we provide a slight optimization that avoids the need for additional annotation. EnerJ implements an extremely simple form of bidirectional type checking [7] that applies approximate arithmetic operators when the result type is approximate: on the right-hand side of assignment operators and in method arguments. We find that this small optimization makes it simpler to write approximate arithmetic expressions that include precise data.

2.4 Control Flow

To provide the desired property that information never flows from approximate to precise data, we must disallow *implicit flows* that occur via control flow. For example, the following program violates the desired isolation property:

```
@Approx int val = ...;
boolean flag; // precise
if (val == 5) { flag = true; } else { flag = false; }
```

Even though flag is precise and no endorsement is present, its value is affected by the approximate variable val.

EnerJ avoids this situation by prohibiting approximate values in conditions that affect control flow (such as if and while statements). In the above example, val == 5 has approximate type because the approximate version of == must be used. Our language disallows this expression in the condition, though the programmer can work around this restriction using if(endorse(val == 5)).

This restriction is conservative: it prohibits approximate conditions even when the result can affect only approximate data. A more sophisticated approach would allow only approximate values to be produced in statements conditioned on approximate data. We find that our simpler approach is sufficient; endorsements allow the programmer to work around the restriction when needed.

2.5 Objects

EnerJ's type qualifiers are not limited to primitive types. Classes also support approximation. Clients of an *approximable* class can create precise and approximate instances of the class. The author of the class defines the meaning of approximation for the class. Approximable classes are distinguished by the @Approximable class annotation. Such a class exhibits qualifier polymorphism [14]: types within the class definition may depend on the qualifier of the instance.

Note that precise class types are not subtypes of their approximate counterparts, as is the case with primitive types (Section 2.1); such a subtyping relationship would be fundamentally unsound for the standard reasons related to mutable references.

2.5.1 Contextual Data Types

The @Context qualifier is available in definitions of non-static members of approximable classes. The meaning of the qualifier depends on the precision of the instance of the enclosing class. (In terms of qualifier polymorphism, @Context refers to the class' qualifier parameter, which is determined by the qualifier placed on the instance.) Consider the following class definition:

```
@Approximable class IntPair {
  @Context int x;
  @Context int y;
  @Approx int numAdditions = 0;
  void addToBoth(@Context int amount) {
    x += amount;
    y += amount;
    numAdditions++;
  }
}
```

If a is an approximate instance of IntPair, then the fields a.x, a.y, and a.numAdditions are all of approximate integer type. However, if p is a precise instance of the class, then p.x and p.y are precise but p.numAdditions is still approximate. Furthermore, the argument to the invocation p.addToBoth() must be precise; the argument to a.addToBoth() may be approximate.

2.5.2 Algorithmic Approximation

Approximable classes may also specialize method definitions based on their qualifier. That is, the programmer can write two implementations: one to be called when the receiver has precise type and another that can be called when the receiver is approximate. Consider the following implementations of a mean calculation over a list of floats:

```
@Approximable class FloatSet {
  @Context float[] nums = ...;
  float mean() {
    float total = 0.0f;
    for (int i = 0; i < nums.length; ++i)
      total += nums[i];
    return total / nums.length;
  }
  @Approx float mean_APPROX() {
    @Approx float total = 0.0f;
    for (int i = 0; i < nums.length; i += 2)
      total += nums[i];
    return 2 * total / nums.length;
  }
}
```

EnerJ uses a naming convention, consisting of the _APPROX suffix, to distinguish methods overloaded on precision. The first implementation of mean is called when the receiver is precise. The second implementation calculates an approximation of the mean: it averages only half the numbers in the set. This implementation will be used for the invocation s.mean() where s is an approximate instance of FloatSet. Note that the compiler automatically decides which implementation of the method to invoke depending on the receiver type; the same invocation is used in either case.

It is the programmer's responsibility to ensure that the two implementations are similar enough that they can be safely substituted. This is important for backwards compatibility (a plain Java compiler will ignore the naming convention and always use the precise

$$
\begin{array}{rcl}
Prg & ::= & \overline{Cls},\ C,\ e \\
Cls & ::= & \texttt{class}\ Cid\ \texttt{extends}\ C\ \{\ \overline{fd}\ \overline{md}\ \} \\
C & ::= & Cid \mid \texttt{Object} \\
P & ::= & \texttt{int} \mid \texttt{float} \\
q & ::= & \texttt{precise} \mid \texttt{approx} \mid \texttt{top} \mid \texttt{context} \mid \texttt{lost} \\
T & ::= & q\ C \mid q\ P \\
fd & ::= & T\ f; \\
md & ::= & T\ m(\overline{T\ pid})\ q\ \{\ e\ \} \\
x & ::= & pid \mid \texttt{this} \\
e & ::= & \texttt{null} \mid \mathcal{L} \mid x \mid \texttt{new}\ q\ C() \mid e.f \mid e_0.f{:=}e_1 \mid e_0.m(\overline{e}) \\
 & & \mid (q\ C)\ e \mid e_0 \oplus e_1 \mid \texttt{if}(e_0)\ \{e_1\}\ \texttt{else}\ \{e_2\}
\end{array}
$$

f	field identifier	pid	parameter identifier
m	method identifier	Cid	class identifier

Figure 1. The syntax of the FEnerJ programming language. The symbol $\overline{A}$ denotes a sequence of elements A.

version) and "best effort" (the implementation may use the precise version if energy is not constrained).

This facility makes it simple to couple algorithmic approximation with data approximation—a single annotation makes an instance use both approximate data (via @Context) and approximate code (via overloading).

2.6 Arrays

The programmer can declare arrays with approximate element types, but the array's length is always kept precise for memory safety. We find that programs often use large arrays of approximate primitive elements; in this case, the elements themselves are all approximated and only the length requires precise guarantees.

EnerJ prohibits approximate integers from being used as array subscripts. That is, in the expression a[i], the value i must be precise. This makes it easier for the programmer to prevent out-of-bounds errors due to approximation.

3. Formal Semantics

To study the formal semantics of EnerJ, we define the minimal language FEnerJ. The language is based on Featherweight Java [16] and adds precision qualifiers and state. The formal language omits EnerJ's endorsements and thus can guarantee isolation of approximate and precise program components. This isolation property suggests that, *in the absence of endorsement*, approximate data in an EnerJ program cannot affect precise state.

The accompanying technical report [33] formalizes this language and proves type soundness as well as a non-interference property that demonstrates the desired isolation of approximate and precise data.

3.1 Programming Language

Figure 1 presents the syntax of FEnerJ. Programs consist of a sequence of classes, a main class, and a main expression. Execution is modeled by instantiating the main class and then evaluating the main expression.

A class definition consists of a name, the name of the superclass, and field and method definitions. The @Approximable annotation is not modeled in FEnerJ; all classes in the formal language can have approximate and precise instances and this has @Context type. The annotation is required only in order to provide backward-compatibility with Java so that this in a non-approximable class has @Precise type.

We use C to range over class names and P for the names of primitive types. We define the precision qualifiers q as discussed in Section 2.1, but with the additional qualifier `lost`; this qualifier is used to express situations when context information is not expressible (i.e., lost). Types T include qualifiers.

Field declarations consist of the field type and name. Method declarations consist of the return type, method name, a sequence of parameter types and identifiers, the method precision, and the method body. We use the method precision qualifier to denote overloading of the method based on the precision of the receiver as introduced in Section 2.5.2. Variables are either a parameter identifier or the special variable `this`, signifying the current object.

The language has the following expressions: the null literal, literals of the primitive types, reads of local variables, instantiation, field reads and writes, method calls, casts, binary primitive operations, and conditionals. For brevity, the discussion below presents only the rules for field reads, field writes, and conditionals.

Subtyping. Subtyping is defined using an ordering of the precision qualifiers and subclassing.

The following rules define the ordering of precision qualifiers:

$\boxed{q <:_{\mathrm{q}} q'}$ ordering of precision qualifiers

$$\frac{q \neq \texttt{top}}{q <:_{\mathrm{q}} \texttt{lost}} \qquad \frac{}{q <:_{\mathrm{q}} \texttt{top}} \qquad \frac{}{q <:_{\mathrm{q}} q}$$

Recall that `top` qualifies the common supertype of `precise` and `approx` types. Every qualifier other than `top` is below `lost`; every qualifier is below `top`; and the relation is reflexive. Note that the `precise` and `approx` qualifiers are not related.

Subclassing is the reflexive and transitive closure of the relation induced by the class declarations. Subtyping takes both ordering of precision qualifiers and subclassing into account. For primitive types, we additionally have that a precise type is a subtype of the approximate type as described in Section 2.1.

Context adaptation. We use *context adaptation* to replace the `context` qualifier when it appears in a field access or method invocation. Here the left-hand side of $\triangleright$ denotes the qualifier of the receiver expression; the right-hand side is the precision qualifier of the field or in the method signature.

$\boxed{q \triangleright q' = q''}$ combining two precision qualifiers

$$\frac{q' = \texttt{context} \ \wedge\ (q \in \{\texttt{approx}, \texttt{precise}, \texttt{context}\})}{q \triangleright q' = q}$$

$$\frac{q' = \texttt{context} \ \wedge\ (q \in \{\texttt{top}, \texttt{lost}\})}{q \triangleright q' = \texttt{lost}} \qquad \frac{q' \neq \texttt{context}}{q \triangleright q' = q'}$$

Note that `context` adapts to `lost` when the left-hand-side qualifier is `top` because the appropriate qualifier cannot be determined.

We additionally define $\triangleright$ to take a type as the right-hand side; this adapts the precision qualifier of the type.

We define partial look-up functions FType and MSig that determine the field type and method signature for a given field/method in an access or invocation. Note that these use the adaptation rules described above.

Type rules. The static type environment ${}^{s}\Gamma$ maps local variables to their declared types.

Given a static environment, expressions are typed as follows:

$\boxed{{}^{s}\Gamma \vdash e : T}$ expression typing

$$\frac{{}^{s}\Gamma \vdash e_0 : q\ C \qquad \mathrm{FType}(q\ C, f) = T}{{}^{s}\Gamma \vdash e_0.f : T}$$

$$\frac{{}^{s}\Gamma \vdash e_0 : q\ C \qquad \mathrm{FType}(q\ C, f) = T \qquad \texttt{lost} \notin T \qquad {}^{s}\Gamma \vdash e_1 : T}{{}^{s}\Gamma \vdash e_0.f\ :=\ e_1 : T}$$

$$\frac{{}^{s}\Gamma \vdash e_0 : \texttt{precise}\ P \qquad {}^{s}\Gamma \vdash e_1 : T \qquad {}^{s}\Gamma \vdash e_2 : T}{{}^{s}\Gamma \vdash \texttt{if}(e_0)\ \{e_1\}\ \texttt{else}\ \{e_2\} : T}$$

A field read determines the type of the receiver expression and then uses FType to determine the adapted type of the field.

A field write similarly determines the adapted type of the field and checks that the right-hand side has an appropriate type. In addition, we ensure that the adaptation of the declared field type did not lose precision information. Notice that we can read a field with lost precision information, but that it would be unsound to allow the update of such a field.

Finally, for the conditional expression, we ensure that the condition is of a precise primitive type and that there is a common type T that can be assigned to both subexpressions.

3.2 Operational Semantics

The runtime system of FEnerJ models the heap h as a mapping from addresses ι to objects, where objects are a pair of the runtime type T and the field values v of the object. The runtime environment ${}^{r}\Gamma$ maps local variables x to values v.

The runtime system of FEnerJ defines a standard big-step operational semantics:

$\boxed{{}^{r}\Gamma \vdash h, e \rightsquigarrow h', v}$ big-step operational semantics

$$\frac{{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h', \iota_0 \qquad h'(\iota_0.f) = v}{{}^{r}\Gamma \vdash h, e_0.f \rightsquigarrow h', v}$$

$$\frac{{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \qquad {}^{r}\Gamma \vdash h_0, e_1 \rightsquigarrow h_1, v \qquad h_1[\iota_0.f := v] = h'}{{}^{r}\Gamma \vdash h, e_0.f := e_1 \rightsquigarrow h', v}$$

$$\frac{{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h_0, (q, {}^{r}\mathcal{L}) \qquad {}^{r}\mathcal{L} \neq 0 \qquad {}^{r}\Gamma \vdash h_0, e_1 \rightsquigarrow h', v}{{}^{r}\Gamma \vdash h, \texttt{if}(e_0)\ \{e_1\}\ \texttt{else}\ \{e_2\} \rightsquigarrow h', v}$$

$$\frac{{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h_0, (q, 0) \qquad {}^{r}\Gamma \vdash h_0, e_2 \rightsquigarrow h', v}{{}^{r}\Gamma \vdash h, \texttt{if}(e_0)\ \{e_1\}\ \texttt{else}\ \{e_2\} \rightsquigarrow h', v}$$

These rules reflect precise execution with conventional precision guarantees. To model computation on an execution substrate that supports approximation, the following rule could be introduced:

$$\frac{{}^{r}\Gamma \vdash h, e \rightsquigarrow h', v \qquad h' \cong \tilde{h'} \qquad v \cong \tilde{v}}{{}^{r}\Gamma \vdash h, e \rightsquigarrow \tilde{h'}, \tilde{v}}$$

We use $\cong$ to denote an equality that disregards approximate values for comparing heaps and values with identical types. The rule permits any approximate value in the heap to be replaced with any other value of the same type and any expression producing a value of an approximate type to produce any other value of that type instead. This rule reflects EnerJ's lack of guarantees for approximate values.

3.3 Properties

We prove two properties about FEnerJ: type soundness and non-interference. The full proofs are listed in the accompanying technical report [33].

The usual type soundness property expresses that, for a well-typed program and corresponding static and runtime environments, we know that (1) the runtime environment after evaluating the expression is still well formed, and (2) a static type that can be assigned to the expression can also be assigned to the value that is the result of evaluating the expression. Formally:

$$\left.\begin{array}{l} \vdash Prg\ \texttt{OK} \wedge \vdash h, {}^{r}\Gamma : {}^{s}\Gamma \\ {}^{s}\Gamma \vdash e : T \\ {}^{r}\Gamma \vdash h, e \rightsquigarrow h', v \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \vdash h', {}^{r}\Gamma : {}^{s}\Gamma \\ h', {}^{r}\Gamma(\texttt{this}) \vdash v : T \end{array}\right.$$

The proof is by rule induction over the operational semantics; in separate lemmas we formalize that the context adaptation operation $\triangleright$ is sound.

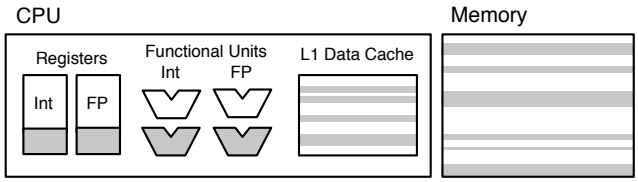

Figure 2. Hardware model assumed in our system. Shaded areas indicate components that support approximation. Registers and the data cache have SRAM storage cells that can be made approximate by decreasing supply voltage. Functional units support approximation via supply voltage reduction. Floating point functional units also support approximation via smaller mantissas. Main memory (DRAM) supports approximation by reducing refresh rate.

The non-interference property of FEnerJ guarantees that approximate computations do not influence precise values. Specifically, changing approximate values in the heap or runtime environment does not change the precise parts of the heap or the result of the computation. More formally, we show:

$$\left.\begin{array}{l} \vdash Prg\ \texttt{OK} \wedge \vdash h, {}^{r}\Gamma : {}^{s}\Gamma \\ {}^{s}\Gamma \vdash e : T \\ {}^{r}\Gamma \vdash h, e \rightsquigarrow h', v \\ h \cong \tilde{h} \wedge {}^{r}\Gamma \cong {}^{r}\tilde{\Gamma} \\ \vdash \tilde{h}, {}^{r}\tilde{\Gamma} : {}^{s}\Gamma \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} {}^{r}\tilde{\Gamma} \vdash \tilde{h}, e \rightarrow \tilde{h'}, \tilde{v} \\ h' \cong \tilde{h'} \\ v \cong \tilde{v} \end{array}\right.$$

For the proof of this property we introduced a *checked* operational semantics that ensures in every evaluation step that the precise and approximate parts are separated. We can then show that the evaluation of a well-typed expression always passes the checked semantics of the programming language.

4. Execution Model

While an EnerJ program distinguishes abstractly between approximate and precise data, it does not define the particular approximation strategies that are applied to the program. (In fact, one valid execution is to ignore all annotations and execute the code as plain Java.) An approximation-aware execution substrate is needed to take advantage of EnerJ's annotations. We choose to examine approximation at the architecture level. Alternatively, a runtime system on top of commodity hardware can also offer approximate execution features (e.g., lower floating point precision, elision of memory operations, etc.). Also, note that algorithmic approximation (see Section 2.5) is independent of the execution substrate. This section describes our hardware model, the ISA extensions used for approximation, and how the extensions enable energy savings.

4.1 Approximation-Aware ISA Extensions

We want to leverage both approximate *storage* and approximate *operations*. Our hardware model offers approximate storage in the form of unreliable registers, data caches, and main memory. Approximate and precise registers are distinguished based on the register number. Approximate data stored in memory is distinguished from precise data based on address; regions of physical memory are marked as approximate and, when accessed, are stored in approximate portions of the data cache. For approximate operations, we assume specific instructions for approximate integer ALU operations as well as approximate floating point operations. Approximate instructions can use special functional units that perform approximate operations. Figure 2 summarizes our assumed hardware model.

An instruction stream may have a mix of approximate and precise instructions. Precise instructions have the same guarantees as instructions in today's ISAs. Note that an approximate instruction is simply a "hint" to the architecture that it may apply a variety of

energy-saving approximations when executing the given instruction. The particular approximations employed by a given architecture are not exposed to the program; a processor supporting no approximations just executes approximate instructions precisely and saves no energy. An approximation-aware ISA thus allows a single binary to benefit from new approximations as they are implemented in future microarchitectures.

Layout of approximate data. Our hardware model supports approximate memory data at a cache line granularity, in which software can configure any line as approximate. This can be supported by having a bit per line in each page that indicates whether the corresponding line is approximate. Based on that bit, a cache controller determines the supply voltage of a line (lower for approximate lines), and the refresh rate for regions of DRAM. This bitmap needs to be kept precise. With a typical cache line size of 64 bytes, this is less than 0.2% overhead. Note that both selective supply voltage for caches [13] and selective refresh rate for DRAM [15] are hardware techniques that have been proposed in the past.

Setting approximation on a cache line basis requires the runtime system to segregate approximate and precise data in different cache lines. We propose the following simple technique for laying out objects with both approximate and precise fields. First, lay out the precise portion of the object (including the `vtable` pointer) contiguously. Each cache line containing at least one precise field is marked as precise. Then, lay out the approximate fields after the end of the precise data. Some of this data may be placed in a precise line (that is, a line containing some precise data already); in this case, the approximate data stays precise and saves no memory energy. (Note that wasting space in the precise line in order to place the data in an approximate line would use more memory and thus more energy.) The remaining approximate fields that do not fit in the last precise line can be placed in approximate lines.

Fields in superclasses may not be reordered in subclasses. Thus, a subclass of a class with approximate data may waste space in an approximate line in order to place precise fields of the subclass in a precise line.

While we simulate the artifacts of this layout scheme for our evaluation, a finer granularity of approximate memory storage would mitigate or eliminate the resulting loss of approximation. More sophisticated layout algorithms could also improve energy savings; this is a target for compile-time optimization. Note that even if an approximate field ends up stored in precise memory, it will still be loaded into approximate registers and be subject to approximate operations and algorithms.

The layout problem is much simpler for arrays of approximate primitive types. The first line, which contains the length and type information, must be precise, with all remaining lines approximate.

4.2 Hardware Techniques for Saving Energy

There are many strategies for saving energy with approximate storage and data operations. This section discusses some of the techniques explored in prior research. We assume these techniques in our simulations, which we describe later. The techniques are summarized in Table 2.

Voltage scaling in logic circuits. Aggressive voltage scaling can result in over 30% energy reduction with $\sim 1\%$ error rate [10] and 22% reduction with $\sim 0.01\%$ error rate. Recent work [9, 17] proposed to expose the errors to applications that can tolerate it and saw similar results. In our model, we assume aggressive voltage scaling for the processor units executing approximate instructions, including integer and floating-point operations. As for an error model, the choices are single bit flip, last value, and random value. We consider all three but our evaluation mainly depicts the random-value assumption, which is the most realistic.

	Mild	Medium	Aggressive
DRAM refresh: per-second bit flip probability	10^{-9}	10^{-5}	10^{-3}
Memory power saved	17%	22%	24%
SRAM read upset probability	$10^{-16.7}$	$10^{-7.4}$	10^{-3}
SRAM write failure probability	$10^{-5.59}$	$10^{-4.94}$	10^{-3}
Supply power saved	70%	80%	90%*
`float` mantissa bits	16	8	4
`double` mantissa bits	32	16	8
Energy saved per operation	32%	78%	85%*
Arithmetic timing error probability	10^{-6}	10^{-4}	10^{-2}
Energy saved per operation	12%*	22%	30%

Table 2. Approximation strategies simulated in our evaluation. Numbers marked with * are educated guesses by the authors; the others are taken from the sources described in Section 4.2. Note that all values for the Medium level are taken from the literature.

Width reduction in floating point operations. A direct approach to approximate arithmetic operations on floating point values is to ignore part of the mantissa in the operands. As observed in [34], many applications do not need the full mantissa. According to their model, a floating-point multiplier using 8-bit mantissas uses 78% less energy per operation than a full 24-bit multiplier.

DRAM refresh rate. Reducing the refresh rate of dynamic RAM leads to potential data decay but can substantially reduce power consumption with a low error rate. As proposed by Liu et al. [23], an approximation-aware DRAM system might reduce the refresh rate on lines containing approximate data. As in that work, we assume that reducing the refresh rate to 1 Hz reduces power by about 20%. In a study performed by Bhalodia [4], a DRAM cell not refreshed for 10 seconds experiences a failure with per-bit probability approximately 10^{-5}. We conservatively assume this error rate for the reduced refresh rate of 1 Hz.

SRAM supply voltage. Registers and data caches consist of static RAM (SRAM) cells. Reducing the supply voltage to SRAM cells lowers the leakage current of the cells but decreases the data integrity [13]. As examined by Kumar [18], these errors are dominated by *read upsets* and *write failures*, which occur when a bit is read or written. A read upset occurs when the stored bit is flipped while it is read; a write failure occurs when the wrong bit is written. Reducing SRAM supply voltage by 80% results in read upset and write failure probabilities of $10^{-7.4}$ and $10^{-4.94}$ respectively. *Soft failures*, bit flips in stored data due to cosmic rays and other events, are comparatively rare and depend less on the supply voltage.

Section 5.4 describes the model we use to combine these various potential energy savings into an overall CPU/memory system energy reduction. To put the potential energy savings in perspective, according to recent studies [12, 24], the CPU and memory together account for well over 50% of the overall system power in servers as well as notebooks. In a smartphone, CPU and memory account for about 20% and the radio typically close to 50% of the overall power [6].

5. Implementation

We implement EnerJ as an extension to the Java programming language based on the pluggable type mechanism proposed by Papi et al. [28]. EnerJ is implemented using the Checker Framework[1]

[1] `http://types.cs.washington.edu/checker-framework/`

infrastructure, which builds on the JSR 308[2] extension to Java's annotation facility. JSR 308 permits annotations on any explicit type in the program. The EnerJ type checker extends the rules from Section 3 to all of Java, including arrays and generics. We also implement a simulation infrastructure that emulates an approximate computing architecture as described in Section 4. [3]

5.1 Type Checker

EnerJ provides the type qualifiers listed in Table 1—@Approx, @Precise, @Top, and @Context—as JSR 308 type annotations. The default type qualifier for unannotated types is @Precise, meaning that any Java program may be compiled as an EnerJ program with no change in semantics. The programmer can add approximations to the program incrementally.

While reference types may be annotated as @Approx, this only affects the meaning of @Context annotations in the class definition and method binding on the receiver. Our implementation never approximates pointers.

5.2 Simulator

To evaluate our system, we implement a compiler and runtime system that executes EnerJ code as if it were running on an approximation-aware architecture as described in Section 4. We instrument method calls, object creation and destruction, arithmetic operators, and memory accesses to collect statistics and inject faults. The runtime system is implemented as a Java library and is invoked by the instrumentation calls. It records memory-footprint and arithmetic-operation statistics while simultaneously injecting transient faults to emulate approximate execution.

To avoid spurious errors due to approximation, our simulated approximate functional units never raise divide-by-zero exceptions. Approximate floating-point division by zero returns the NaN value; approximate integer divide-by-zero returns zero.

5.3 Approximations

Our simulator implements the approximation strategies described in Section 4.2. Table 2 summarizes the approximations used, their associated error probabilities, and their estimated energy savings.

Floating-point bit-width reduction is performed when executing Java's arithmetic operators on operands that are approximate float and double values. SRAM read upsets and write failures are simulated by flipping each bit read or written with a constant probability. For DRAM refresh reduction, every bit also has an independent probability of inversion; here, the probability is proportional to the amount of time since the last access to the bit.

For the purposes of our evaluation, we distinguish SRAM and DRAM data using the following rough approximation: data on the heap is considered to be stored in DRAM; stack data is considered SRAM. Future evaluations not constrained by the abstraction of the JVM could explore a more nuanced model.

5.4 Energy Model

To summarize the effectiveness of EnerJ's energy-saving properties, we estimate the potential overall savings of the processor/memory system when executing each benchmark approximately. To do so, we consider a simplified model with three components to the system's energy consumption: instruction execution, SRAM storage (registers and cache), and DRAM storage. Our model omits overheads of implementing or switching to approximate hardware. For example, we do not model any latency in scaling the voltage on the logic units. For this reason, our results can be considered optimistic; future work should model approximate hardware in more detail.

To estimate the savings for instruction execution, we assign abstract energy units to arithmetic operations. Integer operations take 37 units and floating point operations take 40 units; of each of these, 22 units are consumed by the instruction fetch and decode stage and may not be reduced by approximation strategies. These estimations are based on three studies of architectural power consumption [5, 20, 27]. We calculate energy savings in instruction execution by scaling the non-fetch, non-decode component of integer and floating-point instructions.

We assume that SRAM storage and instructions that access it account for approximately 35% of the microarchitecture's power consumption; instruction execution logic consumes the remainder. To compute the total CPU power savings, then, we scale the savings from SRAM storage by 0.35 and the instruction power savings, described above, by 0.65.

Finally, we add the savings from DRAM storage to get an energy number for the entire processor/memory system. For this, we consider a server-like setting, where DRAM accounts for 45% of the power and CPU 55% [12]. Note that in a mobile setting, memory consumes only 25% of power so power savings in the CPU will be more important [6].

6. Results

We evaluate EnerJ by annotating a variety of existing Java programs. Table 3 describes the applications we used; they have been selected to be relevant in both mobile and server settings.

Applications. We evaluate the FPU-heavy kernels of the SciMark2 benchmark suite to reflect scientific workloads.[4] ZXing is a bar code reader library targeted for mobile devices based on the Android operating system.[5] Our workload decodes QR Code two-dimensional bar code images. jMonkeyEngine is a 2D and 3D game engine for both desktop and mobile environments.[6] We run a workload that consists of many 3D triangle intersection problems, an algorithm frequently used for collision detection in games.

ImageJ is an image-manipulation program; our workload executes a flood fill operation.[7] This workload was selected as representative of error-resilient algorithms with primarily integer—rather than floating point—data. Because the code already includes extensive safety precautions such as bounds checking, our annotation for ImageJ is extremely aggressive: even pixel coordinates are marked as approximate. Raytracer is a simple 3D renderer; our workload executes ray plane intersection on a simple scene.[8]

Annotation approach. We annotated each application manually. While many possible annotations exist for a given program, we attempted to strike a balance between reliability and energy savings. As a rule, however, we attempted to annotate the programs in a way that never causes them to crash (or throw an unhandled exception); it is important to show that EnerJ allows programmers to write approximate programs that never fail catastrophically. In our experiments, each benchmark produces an output on every run. This is in contrast to approximation techniques that do not attempt to prevent crashes [22, 23, 35]. Naturally, we focused our effort on code where most of the time is spent.

[2] http://types.cs.washington.edu/jsr308/

[3] The EnerJ type checker and simulator are available from our website: http://sampa.cs.washington.edu/sampa/EnerJ

[4] SciMark2: http://math.nist.gov/scimark2/

[5] ZXing: http://code.google.com/p/zxing/

[6] jMonkeyEngine: http://www.jmonkeyengine.com/

[7] ImageJ: http://rsbweb.nih.gov/ij/

[8] Raytracer: http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=5590&lngWId=2

Application	Description	Error metric	Lines of code	Proportion FP	Total decls.	Annotated decls.	Endorse-ments
FFT	Scientific kernels from the SciMark2 benchmark	Mean entry difference	168	38.2%	85	33%	2
SOR		Mean entry difference	36	55.2%	28	25%	0
MonteCarlo		Normalized difference	59	22.9%	15	20%	1
SparseMatMult		Mean normalized difference	38	39.7%	29	14%	0
LU		Mean entry difference	283	31.4%	150	23%	3
ZXing	Smartphone bar code decoder	1 if incorrect, 0 if correct	26171	1.7%	11506	4%	247
jMonkeyEngine	Mobile/desktop game engine	Fraction of correct decisions normalized to 0.5	5962	44.3%	2104	19%	63
ImageJ	Raster image manipulation	Mean pixel difference	156	0.0%	118	34%	18
Raytracer	3D image renderer	Mean pixel difference	174	68.4%	92	33%	10

Table 3. Applications used in our evaluation, application-specific metrics for quality of service, and metrics of annotation density. "Proportion FP" indicates the percentage of dynamic arithmetic instructions observed that were floating-point (as opposed to integer) operations.

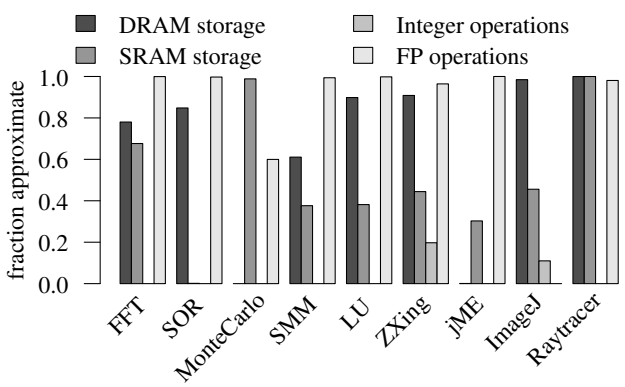

Figure 3. Proportion of approximate storage and computation in each benchmark. For storage (SRAM and DRAM) measurements, the bars show the fraction of byte-seconds used in storing approximate data. For functional unit operations, we show the fraction of dynamic operations that were executed approximately.

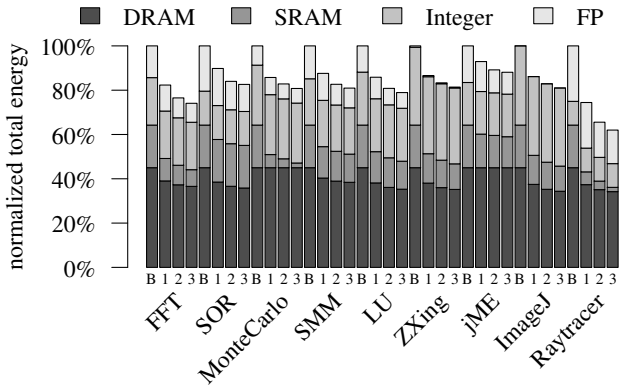

Figure 4. Estimated CPU/memory system energy consumed for each benchmark. The bar labeled "B" represents the baseline value: the energy consumption for the program running without approximation. The numbered bars correspond to the Mild, Medium, and Aggressive configurations in Table 2.

Three of the authors ported the applications used in our evaluation. In every case, we were unfamiliar with the codebase beforehand, so our annotations did not depend on extensive domain knowledge. The annotations were not labor intensive.

QoS metrics. For each application, we measure the degradation in output quality of approximate executions with respect to the precise executions. To do so, we define application-specific quality of service (QoS) metrics. Defining our own ad-hoc QoS metrics is necessary to compare output degradation across applications. A number of similar studies of application-level tolerance to transient faults have also taken this approach [3, 8, 19, 21, 25, 35]. The third column in Table 3 shows our metric for each application.

Output error ranges from 0 (indicating output identical to the precise version) to 1 (indicating completely meaningless output). For applications that produce lists of numbers (e.g., SparseMatMult's output matrix), we compute the error as the mean entry-wise difference between the pristine output and the degraded output. Each numerical difference is limited by 1, so if an entry in the output is NaN, that entry contributes an error of 1. For benchmarks where the output is not numeric (i.e., ZXing, which outputs a string), the error is 0 when the output is correct and 1 otherwise.

6.1 Energy Savings

Figure 3 divides the execution of each benchmark into DRAM storage, SRAM storage, integer operations, and FP operations and shows what fraction of each was approximated. For many of the FP-centric applications we simulated, including the jMonkeyEngine and Raytracer as well as most of the SciMark applications, nearly all of the floating point operations were approximate. This reflects the inherent imprecision of FP representations; many FP-dominated algorithms are inherently resilient to rounding effects. The same applications typically exhibit very little or no approximate integer operations. The frequency of loop induction variable increments and other precise control-flow code limits our ability to approximate integer computation. ImageJ is the only exception with a significant fraction of integer approximation; this is because it uses integers to represent pixel values, which are amenable to approximation.

DRAM and SRAM approximation is measured in byte-seconds. The data shows that both storage types are frequently used in approximate mode. Many applications have DRAM approximation rates of 80% or higher; it is common to store large data structures (often arrays) that can tolerate approximation. MonteCarlo and jMonkeyEngine, in contrast, have very little approximate DRAM data; this is because both applications keep their principal data in local variables (i.e., on the stack).

The results depicted assume approximation at the granularity of a 64-byte cache line. As Section 4.1 discusses, this reduces the number of object fields that can be stored approximately. The impact of this constraint on our results is small, in part because much of the approximate data is in large arrays. Finer-grain approximate memory could yield a higher proportion of approximate storage.

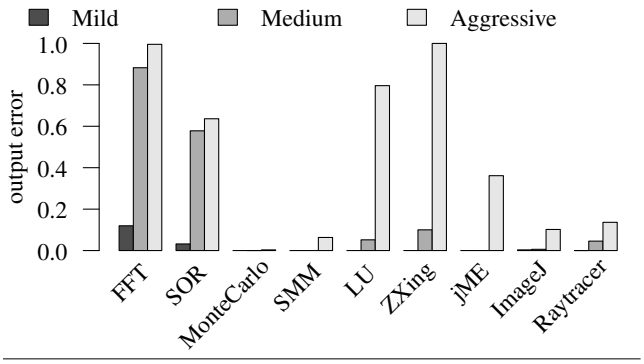

Figure 5. Output error for three different levels of approximation varied together. Each bar represents the mean error over 20 runs.

To give a sense of the energy savings afforded by our proposed approximation strategies, we translate the rates of approximation depicted above into an estimated energy consumption. Figure 4 shows the estimated energy consumption for each benchmark running on approximate hardware relative to fully precise execution. The energy calculation is based on the model described in Section 5.4. These simulations apply all of the approximation strategies described in Section 4.2 simultaneously at their three levels of aggressiveness. As expected, the total energy saved increases both with the amount of approximation in the application (depicted in Figure 3) and with the aggressiveness of approximation used.

Overall, we observe energy savings from 7% (SOR in the Mild configuration) to 38% (Raytracer in the Aggressive configuration). The three levels of approximation do not vary greatly in the amount of energy saved—the three configurations yield average energy savings of 14%, 19%, and 21% respectively. The majority of the energy savings come from the transition from zero approximation to mild approximation. As discussed in the next section, the least aggressive configuration results in very small losses in output fidelity across all applications studied.

The fifth column of Table 3 shows the proportion of floating point arithmetic in each application. In general, applications with principally integer computation (e.g., ZXing and ImageJ) exhibit less opportunity for approximation than do floating-point applications (e.g., Raytracer). Not only do floating-point instructions offer more energy savings potential in our model, but applications that use them are typically resilient to their inherent imprecision.

6.2 Quality-of-Service Tradeoff

Figure 5 presents the sensitivity of each annotated application to the full suite of approximations explored. This quality-of-service reduction is the tradeoff for the energy savings shown in Figure 4.

While most applications show negligible error for the Mild level of approximation, applications' sensitivity to error varies greatly for the Medium and Aggressive configurations. Notably, MonteCarlo, SparseMatMult, ImageJ, and Raytracer exhibit very little output degradation under any configuration whereas FFT and SOR lose significant output fidelity even under the Medium configuration. This variation suggests that an approximate execution substrate for EnerJ could benefit from tuning to the characteristics of each application, either offline via profiling or online via continuous QoS measurement as in Green [3]. However, even the conservative Mild configuration offers significant energy savings.

Qualitatively, the approximated applications exhibit gradual degradation of perceptible output quality. For instance, Raytracer always outputs an image resembling its precise output, but the amount of random pixel "noise" increases with the aggressiveness of approximation. Under the Mild configuration, it is difficult to distinguish the approximated image from the precise one.

We also measured the relative impact of various approximation strategies by running our benchmark suite with each optimization enabled in isolation. DRAM errors have a nearly negligible impact on application output; floating-point bit width reduction similarly results in at most 12% QoS loss in the Aggressive configuration. SRAM write errors are much more detrimental to output quality than read upsets. Functional unit voltage reduction had the greatest impact on correctness. We considered three possibilities for error modes in functional units: the output has a single bit flip; the last value computed is returned; or a random value is returned. The former two models resulted in significantly less QoS loss than the random-value model (25% vs. 40%). However, we consider the random-value model to be the most realistic, so we use it for the results shown in Figure 5.

6.3 Annotation Effort

Table 3 lists the number of qualifiers and endorsements used in our annotations. Only a fraction of the types in each program must be annotated: at most 34% of the possible annotation sites are used. Note that most of the applications are short programs implementing a single algorithm (the table shows the lines of code in each program). Our largest application, ZXing, has about 26,000 lines of code and only 4% of its declarations are annotated. These rates suggest that the principal data amenable to approximation is concentrated in a small portion of the code, even though approximate data typically dominates the program's dynamic behavior.

Endorsements are also rare, even though our system requires one for every approximate condition value. The outlier is ZXing, which exhibits a higher number of endorsements due to its frequency of approximate conditions. This is because ZXing's control flow frequently depends on whether a particular pixel is black.

Qualitatively, we found EnerJ's annotations easy to insert. The programmer can typically select a small set of data to approximate and then, guided by type checking errors, ascertain associated data that must also be marked as approximate. The requirements that conditions and array indices be precise helped quickly distinguish data that was likely to be sensitive to error. In some cases, such as jMonkeyEngine and Raytracer, annotation was so straightforward that it could have been largely automated: for certain methods, every `float` declaration was replaced indiscriminately with an `@Approx float` declaration.

Classes that closely represent data are perfect candidates for `@Approximable` annotations. For instance, ZXing contains `BitArray` and `BitMatrix` classes that are thin wrappers over binary data. It is useful to have approximate bit matrices in some settings (e.g., during image processing) but precise matrices in other settings (e.g., in checksum calculation). Similarly, the jMonkeyEngine benchmark uses a `Vector3f` class for much of its computation, which we marked as approximable. In this setting, approximate vector declarations (`@Approx Vector3f v`) are syntactically identical to approximate primitive-value declarations (`@Approx int i`).

We found that the `@Context` annotation helped us to approach program annotation incrementally. A commonly-used class that is a target for approximation can be marked with `@Context` members instead of `@Approx` members. This way, all the clients of the class continue to see precise members and no additional annotation on them is immediately necessary. The programmer can then update the clients individually to use the approximate version of the class rather than addressing the whole program at once.

An opportunity for algorithmic approximation also arose in ZXing. The `BitArray` approximable class contains a method `isRange` that takes two indices and determines whether all the bits between

the two indices are set. We implemented an approximate version of the method that checks only some of the bits in the range by skipping some loop iterations. We believe that application domain experts would use algorithmic approximation more frequently.

In one case, we found it convenient to introduce a slight change to increase the fault tolerance of code dealing with approximate data. ZXing has a principally floating-point phase that performs an image perspective transform. If the transform tried to access a coordinate outside of the image bounds, ZXing would catch the ArrayIndexOutOfBoundsException and print a message saying that the image transform failed. We modified the algorithm to silently return a white pixel in this case. The result was that the image transform became more resilient to transient faults in the transformation coordinates. We marked these coordinates as approximate and then endorsed them at the point they are used as array indices. In no case, however, does an application as we annotated it do *more* computation than the pristine version.

7. Related Work

Space constraints preclude a discussion of the vast body of compiler or hybrid hardware/software work to improve energy efficiency. Instead, we focus on work we are aware of that exploits approximate computing to improve energy.

Many studies have shown that a variety of applications have a high tolerance to transient faults [8, 19, 21, 22, 25, 35]. However, certain parts of programs are typically more fault-tolerant than others. Our work exploits this property by allowing the programmer to distinguish critical from non-critical computation.

Our work at the language level was influenced by previous work on techniques for trading off correctness for power savings. Flikker [23] proposes a programming model for reducing the DRAM refresh rate on certain heap data via low-level program annotations. Besides being limited to heap storage, Flikker does not provide any safety guarantees. Relax [9] is an architecture that exposes timing faults to software as opposed to providing error recovery automatically in hardware; its goal is to improve error tolerance with lower power by exploiting portions of code that are tolerant to error. While Relax focuses on error recovery and hardware design simplicity, EnerJ emphasizes energy-efficiency over error detectability and supports a wider range of power-saving approximations. Moreover, Relax explores a *code-centric* approach, in which blocks of code are marked for failure and recovery while EnerJ employs *data-centric* type annotations.

Work by Rinard et al. proposes approximate code transformations in the compiler [1, 25, 31]. Relatedly, EnerJ's support for algorithmic approximation, the ability to write an approximate implementation and a precise implementation of the same functionality, bears some similarity to the Green programming model [3]. However, Green primarily concerns itself with online monitoring of application QoS; EnerJ's guarantees are entirely static. Overall, EnerJ's type system makes approximation-based approaches to energy savings general (it supports approximate operations, storage, and algorithms) and safe (it provides static isolation guarantees).

Using types to achieve fine-grained isolation of program components is inspired by work on information flow types for secure programming [26, 32]. That body of work also influenced the *endorsement* construct for explicitly violating non-interference.

Work by Perry et al. also focuses on static verification of fault tolerance [29, 30]. That work focuses on detection and recovery rather than exposing transient faults.

8. Conclusion

Approximate computing is a very promising way of saving energy in large classes of applications running on a wide range of systems, from embedded systems to mobile phones to servers. We propose to use a type system based on information-flow tracking ideas: variables and objects can be declared as approximate or precise; approximate data can be processed more cheaply and less reliably; and we can statically prove that approximate data does not unexpectedly affect the precise state of a program. Our type system provides a general way of using approximation: we can use approximate storage by mapping data to cheaper memory, cache, and registers; we can use approximate operations by generating code with cheaper, approximate instructions; and we can use method overloading and class parameterization to enable algorithmic approximation.

We implement our type system on top of Java and experiment with several applications, from scientific computing to image processing to games. Our results show that annotations are easy to insert: only a fraction of declarations must be annotated and endorsements are rare. Once a program is annotated for approximation, the runtime system or architecture can choose several approximate execution techniques. Our hardware-based model shows potential energy savings between 7% and 38%.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. We also thank the members of the Sampa group for their feedback on the manuscript. This work was supported in part by NSF grant CCF-1016495, NSF CAREER REU grant CCF-0846004, an NSF Graduate Fellowship and a Microsoft Research Faculty Fellowship.

References

[1] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical report, MIT, 2009.

[2] A. Askarov and A. C. Myers. A semantic framework for declassification and endorsement. In *ESOP*, 2010.

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

[4] V. Bhalodia. SCALE DRAM subsystem power analysis. Master's thesis, MIT, 2005.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.

[6] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *USENIX*, 2010.

[7] A. Chlipala, L. Petersen, and R. Harper. Strict bidirectional type checking. In *TLDI*, 2005.

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

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

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

[11] M. D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, 2008.

[12] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *ISCA*, 2007.

[13] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *ISCA*, 2002.

[14] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*, 1999.

[15] M. Ghosh and H.-H. S. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMs. In *MICRO*, 2007.

[16] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3), 2001.

[17] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Designing a processor from the ground up to allow voltage/reliability tradeoffs. In *HPCA*, 2010.

[18] A. Kumar. *SRAM Leakage-Power Optimization Framework: a System Level Approach*. PhD thesis, University of California at Berkeley, 2008.

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

[20] 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.

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

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

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

[24] A. Mahesri and V. Vardhan. Power consumption breakdown on a modern laptop. In *PACS*, 2004.

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

[26] A. C. Myers. JFlow: practical mostly-static information flow control. In *POPL*, 1999.

[27] K. Natarajan, H. Hanson, S. W. Keckler, C. R. Moore, and D. Burger. Microprocessor pipeline energy analysis. In *ISLPED*, 2003.

[28] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, 2008.

[29] F. Perry and D. Walker. Reasoning about control flow in the presence of transient faults. In *SAS*, 2008.

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

[31] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Onward!*, 2010.

[32] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security*, 21(1), 2003.

[33] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation — Full Proofs. Technical Report UW-CSE-10-12-01, University of Washington, 2011.

[34] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Trans. VLSI Syst.*, 8(3), 2000.

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