Sustainable Programming with Eco

Haitao Steve Zhu Chaoren Lin Yu David Liu
Department of Computer Science
SUNY Binghamton
Binghamton NY 13902, USA
{hzhu1,clin18,davidL}@binghamton.edu

ABSTRACT
This paper presents Eco, a novel energy-aware programming model centering around sustainability. A sustainable program adaptively adjusts its own behaviors to stay on a given energy budget, avoiding both deficit that would lead to battery drain or CPU overheating, and surplus that could have been used to improve the quality of the program output. Sustainability management in Eco is built on a key insight where sustainability is viewed as a form of supply and demand matching, and a sustainable program consistently maintains the equilibrium between supply and demand. Concretely, Eco introduces a novel language abstraction—sustainable blocks—to achieve fine-grained programmable sustainability. Eco is implemented as a minimal extension to Java, and we validate its design by upgrading real-world Java programs, enabling two crucial features toward energy-aware software: battery awareness and temperature awareness.

1. INTRODUCTION
Modern computing platforms move forward in two trajectories: they either go “nimble,” or go “mighty.” Nimble or mighty, a common hurdle lies ahead: energy consumption. “Nimble” computers—such as RFIDs, wearable electronics, smartphones, tablets, and laptops—often operate on battery. “Mighty” computers—represented by data centers, cloud servers, and many-core clusters—are not only power-hungry, but also overheating-anxious, with cooling taking up to 35% of their operational costs [3]. There is a growing interest in the software community to address these challenges, with innovations through software design methodologies [19, 21, 26, 28, 2], programming models [9, 27, 4, 30], and program analysis [5, 15]. A common theme of these efforts is to improve software energy efficiency, i.e., maximizing energy savings.

The focus of this paper is a much less explored property: sustainability, i.e. the ability for a program to “stay on budget,” neither yielding deficit nor surplus. At the first glance, the goal of maintaining sustainability is counter-intuitive: wouldn’t it be more desirable to yield as much surplus as possible? Ideally, yes, but free lunch is over. The most effective energy management strategies are known to be trade-offs, such as sacrificing performance [25], data precision [27], and quality of service [4]. To balance the trade-off, we believe it is more sensible for an energy-aware program to neither squander nor skimp, a design philosophy we term sustainable programming.

Our concrete proposal is Eco, a novel and simple programming model to help developers construct sustainable software by design, centering around three language features:

1. sustainable blocks: an Eco programmer can associate any code fragment with an energy-related budget—such as the execution of the code fragment does not consume more than 20% of remaining battery or raise temperature by 10°C—and the language runtime makes best effort to adjust program behaviors to meet the budget.

2. first-class mode cases: Eco programmers can define alternative program behaviors as first-class values, such as a value that captures the intention of setting the image resolution to 1024 × 768 when an energy budget can be easily met, and to 800 × 600 otherwise.

3. uniform blocks: Eco allows programmers to maintain the application-specific consistency in the presence of automatic program behavior adaption, such as preventing the change of image resolution half way through processing one image.

Eco is built upon an intuitive view that sustainability can only be achieved through recurrent and dynamic negotiation between supply and demand. Even though supply and demand modeling is not new in computer science [6, 31, 7, 23, 22], we believe Eco is the first system that builds supply and demand directly into programming models and bring “programmable sustainability” to modern software engineering. Among the numerous energy/thermal management systems [11, 29, 31, 8, 10, 12, 23]—most from VLSI, architecture, and OS communities—Eco is distinctive in its attempt to combine the strengths of the human (the programmer) and the machine (the language runtime):

1. programmable supply/demand characterizations (or, how the human helps the machine): To a sustainable block, an Eco programmer provides a supply characterization to specify the application-specific energy budget, and a
demand characterization to help the program runtime assess the "work" progress of the sustainable block.

2. automated supply/demand negotiation (or, how the machine helps the human): with the supply/demand characterizations, the Eco language runtime automatically monitors the changes in supply and demand as the sustainable block is executed, and adaptively adjust program behaviors to meet the goal of sustainability. Programmers do not need to consider when to adjust, what to adjust to, and how to maintain sustainability and uniformity.

In summary, this paper makes the following contributions:

• It calls for the need for sustainable programming in energy-aware software development, and describes the first programming language to make sustainability programmable.

• It provides a novel programming abstraction where sustainability is modeled as supply/demand characterization and negotiation.

• It supports a recurring theme in software-based energy management — alternative program behaviors for different energy needs — with first-class mode cases that support extendable programming.

• It defines the Eco language in a rigorous setting, with a formal model that captures the essence of sustainable programming.

• It reports a prototype implementation, and validates the language design through extending real-world applications with the ability of battery awareness and temperature awareness.

2. SUSTAINABLE PROGRAMMING BY EXAMPLE

We use a simple raytracing example in Figure 1 to demonstrate the basic ideas of sustainable programming. Raytracing is a technique in rendering where an object in the form of a Model is rendered into an Image—a 2D array of colored pixels—based on the rays cast from the Camera to the object.

Eco is an extension to Java, with several notable features we highlight next.

Sustainable blocks.

An energy-conscious programmer may declare the most expensive part of her program in sustainable blocks. In this example, the block encloses the ray tracing for IMGS number of images, as shown between Line 22 and Line 41. The programmer can set a budget for executing this code fragment through supply characterization: in Line 40, the budget (also known as supply sum) is set at 20% of the remaining battery.

To help the language runtime correlate the change in battery over time and the progress of the program execution, Eco programmers are asked to provide a "progress indicator" through demand characterization. In this oversimplified example, the programmer just says that the demand sum—i.e. the overall number of units of work—is IMGS (the number of images), and at a given moment during the sustainable block

```java
1 mode { lo <=: mid }; 2 mode { mid <=: hi }; 3 class Raytracer {
5 mcase<Res> res = {
6 lo: new Res(640, 480);
7 mid: new Res(800, 600);
8 hi: new Res(1024, 768);
9 };
11 mcase<int> depth = {
12 lo: 10;
13 mid: 20;
14 hi: 30;
15 };
17 Ray[][] getView(Camera camera, Res res) { ... }
18 Image[] run(Camera camera, Model[] models) {
19 Image[] images;
21 int IMAX, JMAX, IMGS = 100;
22 sustainable {
23 for (int num = 0; num < IMGS; num++) {
24 uniform { 25 Ray[][] view = getView(camera, res);
26 Image image = new Image(res);
27 IMAX = res.x;
28 JMAX = res.y;
29 } for (int i = 0; i < IMAX; i++) {
30 for (int j = 0; j < JMAX; j++) {
31 image.setColor(i, j, 32 view[i][j].trace(models[num], depth)
33 );
34 }
35 }
36 images[num] = image;
37 }
39 bsupply (0.2 * battery)
41 demand (IMGS) -> (IMGS - num);
42 return images;
43 }
44 }
45 class Model { ... } // model to be rendered
46 class Camera {...} // camera for rendering
47 class Image { ... } // 2D array of color pixels
48 class Ray {...} // ray
49 class Res {...} // resolution
```

Figure 1: A Battery-Aware Raytracing Application
“downgrades” the mode of its execution whenever deficit code within a sustainable block as uniform may unsettle (some) programmers. To produce a picture such as Figure 2, whose uneven rendering half way through raytracing an image, the program may for instance, had we allowed the image resolution to change arbitrary time may yield results unacceptable to programmers. For example, had we allowed the image resolution to change into a surplus. And to the more energy-consuming behavior when it runs into a deficit, our intuition of sustainability: the program adapts to the mode. Overall, the described behavior here is aligned with the concept of sustainability: the program adapts to the mode. Mode cases in Eco are first-class values: they can be assigned to variables, passed around as method parameters, or stored in fields.

To tie everything together, the executing sustainable block “downgrades” the mode of its execution whenever deficit happens—picking a precedent in the total order—and the mode case used in that block can be intuitively viewed as automatically destructed based on the new mode. For example, the mode case of depth under the execution of mid mode would be destructed into 20. The scenario for surplus is similar, except that the sustainable block “upgrades” its mode. Overall, the described behavior here is aligned with our intuition of sustainability: the program adapts to the less energy-consuming behavior when it runs into a deficit and to the more energy-consuming behavior when it runs into a surplus.

**Uniform Blocks.**

Coming with the flexibility of first-class mode cases is a design challenge: allowing for mode case destruction at arbitrary time may yield results unacceptable to programmers. For instance, had we allowed the image resolution to change half way through raytracing an image, the program may produce a picture such as Figure 2, whose uneven rendering may unsettle (some) programmers.

An Eco programmer may choose to declare a block of code within a sustainable block as uniform, as in Line 24 to Line 29. This precludes mode change from happening during the execution of the enclosed code block. This consistency-preserving construct is reminiscent of the property of atomicity [14, 20, 18].

**Battery-Awareness and Temperature Awareness.**

The example here is aware of battery states. With minimal changes, the same program can turn into a temperature-aware one: by simply changing Line 40 to tsupply(10), the same sustainable block is to be executed with a (temperature) budget of 10°C, i.e., not raising CPU temperatures by more than 10°C.

As we shall see, both bsupply and tsupply are syntactic sugars encodable by a more general form of supply characterization. Just as demand characterization includes demand case, supply characterization includes supply sum and demand gauge, supply can be symmetrically characterized by supply sum and supply gauge.

### 3. ECO LANGUAGE DESIGN

#### 3.1 Abstract Syntax

The core abstract syntax of Eco is defined in Figure 3. As Eco is an extension to Java, the formal core here is built upon Featherweight Java (FJ) [17]. Standard “overline” notation  represents a sequence of its. A program consists of a list of classes (C), together with a list of mode declarations (D). Selected Java expressions (je) are slightly richer than FJ, including numeric constants n, variables x, self reference this, field read e.fd, field write e.fd = e, method invocation e.md(e), instantiation new X, and variable declaration and continuation \( \tau = e; e' \). Pre-defined class name Object is the root class. A program bootstraps from the main method of a class named Main, which has no fields. For simplicity, we do not formalize constructors and multi-argument methods. Expression \( \text{sustainable}(e) \supset \text{supply} e_1 \supset c_1 \supset \text{demand} e_2 \supset c_2 \), or abbreviated as \( e \supset e_1 \supset c_1 \supset e_2 \supset c_2 \), defines a sustainable block for e where \( e_1 \) is called the supply sum, \( c_1 \) is called the supply gauge, \( e_2 \) is called the demand sum, and \( c_2 \) is called the demand gauge. Expression \( \{ \forall \tau \} \) represents a mode case, where each element \( m : e \) in the sequence — which we call a case member — intuitively says that under mode m, the mode case should behave as a value computed

### 3.2 Design Characterization

#### 3.2.1. Just as demand characterization includes demand case, supply characterization includes supply sum and demand gauge, supply can be symmetrically characterized by supply sum and supply gauge.
by $e$. Expression $\text{uniform} [e]$ defines a uniform block for $e$.

Given $\text{getB}$ and $\text{getT}$ are (effectful) 0-arity functions that can read the current battery capacity and current CPU temperatures, the sugared syntax we used in Section 2 can be desugared as in Figure 4.

For any program $P = (C, D)$, we require the modes defined in $D$ form a total order defined by $\prec$. Formally, this means the reflexive and transitive closure of relation $R$ is a total order, where $R$ is the smallest relation containing elements $(m : m')$ where $\text{mode } m \prec m \in D$. We further call the least element and greatest element in the total order the least mode and greatest mode respectively. Implicitly parameterized by $D$, unary operators $\uparrow$ and $\downarrow$ are standard successive and precedent operators over total order. Furthermore, $\uparrow m \overset{\text{def}}{=} m$ if $m$ is the greatest mode and $\downarrow m \overset{\text{def}}{=} m$ if $m$ is the least mode.

Since our examples frequently use values of primitive types, numeric values $n$ are explicitly supported, but implicit boxing and unboxing are used when they are assigned, stored, and passed around. For instance, the Eco programmer expression $x = 2$ is encoded as $x.\text{val} = 2$, and primitive value assignment $x = y$ is encoded as $x.\text{val} = y.\text{val}$, where $\text{val}$ is a built-in field of the primitive type at concern. As we shall see, this treatment helps us unify the discussion of semantics.

### 3.2 Supply/Demand Matching

The matching between supply and demand is captured by the following predicates, where numeric value $n_1$ corresponds to supply sum, $n_1'$ to supply gauge, $n_2$ to demand sum, and $n_2'$ to demand gauge, and $\epsilon$ is a small positive numeric constant close to 0:

- $\text{MATCH}(n_1, n_1', n_2, n_2') \overset{\text{def}}{=} |\frac{n_1'}{n_1} - \frac{n_2'}{n_2}| \leq \epsilon$
- $\text{SURPLUS}(n_1, n_1', n_2, n_2') \overset{\text{def}}{=} \frac{n_1'}{n_1} - \frac{n_2'}{n_2} > \epsilon$
- $\text{DEFICIT}(n_1, n_1', n_2, n_2') \overset{\text{def}}{=} \frac{n_2'}{n_2} - \frac{n_1'}{n_1} > \epsilon$

Intuitively, the supply and the demand match when the proportion of remaining supply ($\frac{n_1'}{n_1}$) and the proportion of remaining demand ($\frac{n_2'}{n_2}$) are close in range. If the former significantly exceeds the latter, we have a "surplus" of supply. If the latter significantly exceeds the former, we have a "deficit" of supply. Observe that in real number domains, equality is generally established through approximacy in distance. In practice, a small yet not minuscule $\epsilon$ can help increase the stability of the system—otherwise every supply/demand comparison would yield either a surplus or deficit. Eco language runtime selects $\epsilon = 0.1$: the program is considered "sustainable" if the remaining proportions of supply and demand are within $\pm 10\%$ difference.

With this we can define convenience function on mode adjustment:

$$\text{adjust}(m, v_1, v_1', v_2, v_2') \overset{\text{def}}{=} \begin{cases} m & \text{if } \text{MATCH}(v_1, v_1', v_2, v_2) \\ \uparrow m & \text{if } \text{DEFICIT}(v_1, v_1', v_2, v_2) \\ \downarrow m & \text{if } \text{SURPLUS}(v_1, v_1', v_2, v_2) \end{cases}$$

The language is neutral on the concrete "unit" of the supply or demand. For instance, a supply can either be battery capacity in joules, battery capacity in milliwhattour, or battery remaining time in seconds, or temperature in celsius, etc. The only requirement to keep the definitions above sound is the supply sum and supply gauge refer to the same unit. Similarly, the programmer is also at the liberty to decide on the "unit" of demand, as long as the demand sum and the demand gauge refer to the same unit. We will come back to this topic in Section 5.2.

### 3.3 Language Semantics

We now informally describe the dynamic behaviors of Eco programs, with a formal definition of operational semantics in the Appendix. Runtime values $v$ are either a numeric $n$, an object reference $o$, or a first-class mode case $\{m \mapsto v\}$. The runtime configuration of an Eco program consists of:

- a heap $H$, defined as a mapping from object references to field stores that represent the state of the objects.
- a context $K$, defined as a sequence of elements either in the form of $\ell : m$ or of $\ell$ alone, where $\ell$ is the program label for sustainable/uniform blocks, and $m$ is the mode

The evaluation order of Eco is consistent with Java. For expression $\{\text{π} : v\}$, all case expressions $v_1, \ldots, v_n$ are evaluated at mode case definition time.

### Sustainable/Uniform Block Entry/Exit

The runtime maintains context $K$ as a stack with several simple rules:

- **R1** When a sustainable block is encountered, entry $\ell : m$ is pushed onto $K$ (adding to the right most position of the sequence), where $\ell$ is the program label of the block, and $m$ is the "current" mode before block entry, defined shortly;
- **R2** When a uniform block is encountered, entry $\ell$ is pushed onto $K$ where $\ell$ is the program label of the block;
- **R3** Upon block exit, the stack top is popped (removing from the right most position of the sequence).

Together with one rule for bootstrapping:

- **R4** When a program bootstraps, set $K$ as $\ell_{\text{boot}} : m$, where $m$ is the greatest mode

With **R4**, code not in any sustainable block runs under the greatest mode: they should not subject to sustainability management anyways. The informal notion of "current mode" in **R1** is defined by convenience function $\text{current}(K)$, as $m$ where $(\ell : m)$ is an element in $K$, and there does not exist any $\ell' : m'$ in $K$ to its right in the sequence. Combining this with **R1** and **R4**, it means when a non-nested sustainable block is first encountered, the current mode is (initially) the greatest mode; when a nested block is encountered, the current mode is (initially) the current mode of the block immediately enclosing the nested block.

### Mode Case Destruction

Mode cases are destructed based on the current mode of the enclosing sustainable block. We delay the destruction until an evaluation must make the destruction to progress, so that the values can be first-class as long as possible. This means a first-class mode case is not destructed until it serves as the target of field access or messaging. To define this behavior, let us define $\overset{m}{v}$ as $v$ where $v = \{m_1 : v_1, \ldots, m_n :
there exists

\[ v_n \text{ and } m_i \] are the least mode among \( \{ m_1, \ldots, m_n \} \) that are equal to or greater than \( m \) according to the total order defined by the program.

**R5** Given \( v \) as a mode case value, expression \( \tau.v.f.d \) evaluates to \( \nu_0.f.d \), where \( K \) is the context and \( \nu_0=\langle \text{current}(K) \rangle \). Similarly, \( \nu.v.f.d = \nu' \) evaluates to \( \nu_0.f.d = \nu' \), and \( \nu.m.d(\nu') \) evaluates to \( \nu_0.m.d(\nu') \).

The use of \( \langle \text{current}(K) \rangle \) captures two intuitive facts: (i) if the mode case value explicitly includes a case member whose label is the current mode of the enclosing sustainable block, that case should be selected; (ii) otherwise, the case member whose mode is greater to the current mode—but closest to it—should be selected.

Recall that \( K \) at least contains \( \ell\text{\textunderscore\text{start}} : m \) where \( m \) is the greatest mode. This implies when a mode case value appears outside of any sustainable block, the greatest mode is used. This aligns with our intuition that code not subject to sustainability management should not be approximated.

The semantic definition here allows for nested first-class mode cases, with multiple **R5** applications.

**Sustainability Check.**

Several design possibilities exist as to *when* sustainability check should happen, i.e., the timing for determining whether MATCH, SURPLUS, or DEFICIT holds: (1) check periodically; (2) check whenever the supply gauge changes; (3) check whenever the demand gauge changes.

For route (1), we find it challenging and ad hoc to set a fixed “period” that fits all programs: an interval of 100ms might be appropriate for a 15-second execution, but excessive for a 15-minute execution. For (2), the supply gauge in energy-aware programming is often associated with system-level variables (as remaining battery) whose updates are often performed outside the program runtime and out of the control of programmers. This may lead to platform-dependent behaviors: the same program may have sustainability checks at very different rates on two different laptops, just because they are equipped with different battery drivers.

**Eco** chooses route (3). According to our practice, the demand gauge is typically characterized with an expression formed by program variables, such as loop index in the example in Figure 1. The state change of these variables is visible and predictable to the programmer. It is our belief that for application-level energy management strategies—a family that **Eco** belongs to—the programmer should have control on program behaviors related to energy awareness. One conservative implementation of (3) would be to perform sustainability check whenever state change happens, with the following rules, where predicate INUBLOCK(\( K \)) holds iff there exists \( \ell \in K \):

**R6** : The evaluation of \( o.f.d = v \) over heap \( H \) and context \( K \) is a FJ-like field update over \( H \) if INUBLOCK(\( K \)).

**R7** : The evaluation of \( o.f.d = v \) over heap \( H \) and context \( K \) is a FJ-like field update over \( H \), together with the update of \( K \) to \( K_0, (\ell : m') \), given all of the following conditions:

- \( \neg \text{INUBLOCK}(K) \)
- \( K = K_0, (\ell : m) \)
- \( \ell \) is a label for \( \langle e \rangle \) \( \{ e_1 \mapsto e'_1 \} \delta \{ e_2 \mapsto e'_2 \} \delta \)
- \( e_1 \) evaluates to \( v_1 \)
- \( e'_1 \) evaluates to \( v'_1 \)
- \( e_2 \) evaluates to \( v_2 \)
- \( e'_2 \) evaluates to \( v'_2 \)
- \( m' = \text{adjust}(m, v_1, v'_1, v_2, v'_2) \)

Rule R6 says that if the state change happens inside a uniform block, no sustainability check is needed—i.e., the end goal of sustainability check is to adjust current mode, but uniform blocks disable adjustment. R7 says the state change otherwise would lead to sustainability check—involving the evaluation of the supply sum/gauge expressions and the demand sum/gauge expressions—and ultimately the change of the current mode for the enclosing sustainable block. In practice, R7 is prohibitively conservative: the change of \( o.f.d \) may have no effect on \( e'_2 \) at all. We can optimize by introducing:

**R8** : The evaluation of \( o.f.d = v \) over heap \( H \) and context \( K \) is a FJ-like field update over \( H \) if

- \( K = K_0, (\ell : m) \)
- \( \ell \) is a label for \( \langle e \rangle \) \( \{ e_1 \mapsto e'_1 \} \delta \{ e_2 \mapsto e'_2 \} \delta \)
- \( \neg \text{EFFECT}(o, e_1, H) \), and \( \neg \text{EFFECT}(o, e'_1, H) \), and \( \neg \text{EFFECT}(o, e_2, H) \), and \( \neg \text{EFFECT}(o, e'_2, H) \)

where predicate \( \text{EFFECT}(o, e, H) \) says that changes in \( o \) may affect the evaluation of \( e \). We leave this definition abstract; one possible implementation is that any subexpression of \( e \) (including \( e \)) does not alias to \( o \). **R8** is thus only applied when the last condition described in the last bullet above does not hold.

### 3.4 Types

Type \( \tau \) is either primitive \( \text{num} \), or an object type \( x \), or a type for the mode case value \( \text{mcase}(\tau) \). The last type form says that the expression defined for each case member has type \( \tau \). A mode case definition is well-typed if the expressions for all case members have the same type \( \tau \), and we
1 int workdone = 0;
2 int workleft = IMGs * res. x * res. y;
3 sustainable {  
4     for (int num = 0; num < IMGs; num++) {
5         ...
6         for (int i = 0; i < MAX; i++) {
7             for (int j = 0; j < MAX; j++) {
8                 ...
9             }
10         }
11         workdone = MAX * MAX;
12         workleft = (IMGs-num) * MAX; JMAX;
13         images[num] = image;
14     }
15 }
16 bsupply (PER * BUDGET)  
17 demand (workdone + workleft) -> (workleft);

Figure 5: Refined Demand Characterization

type the mode case as mcased (τ). Subtyping is defined as: (1) Java-style nominal subtyping still holds, in that if class X is a subclass of Y, then X is a subtype of Y. (2) mcased (τ) is a subtype of τ. A derived fact is that for a mode case of type mcased (τ), it is well-typed if any of its case member expression has type τ’ as long as τ’ is a subtype of τ.

The second rule above is friendly for incremental energy-aware support. A programmer can start with Java program, and then incrementally change expressions such as new Res(1024, 768) to the mode case we used in Figure 1. The rest of the source code remains unchanged, and the program still typechecks. This feature is particularly attractive for upgrading legacy Java code such as libraries.

4. IMPLEMENTATION AND EVALUATION

Eco is implemented on top of the Polyglot compiler framework 3.0 [24], as an extension to Java. We selected the following Java benchmarks and modified them into Eco: (1) sunflow, a rendering system that uses raytracing, (2) jspider, a web crawler, (3) montecarlo, a financial simulation from the Java Grande benchmark suite, (4) xalan, an XSLT processor that converts XML documents, and (5) rasterizer, from the batik distribution, that rasterizes SVG files. Our main selection criterion is diversity, covering domains such as graphics, web, statistics, program transformation, and data transformation. These benchmarks cover a diverse range of applications to support variability in program behaviors, such as image resolution in sunflow, URL request intervals in jspider, sampling size in montecarlo, and rasterization quality in batik.

All experiments were performed on a Intel 2.53GHz Duo core CPU laptop with 8GB RAM. The battery data was queried through Advanced Configuration and Power Interface (ACPI) [1]. The system variable (in file form) being queried is RemainingCapacity in BatteryStatus directory, in milliwatt hours (mwh). The CPU temperature was queried with the CoreTemp tool. The tool queries temperature every 5 seconds, in Celsius.

4.1 Battery-Aware Programming (sunflow)

![Figure 6: sunflow (PER = 0.9)](image6)

![Figure 7: sunflow (PER = 0.7)](image7)

![Figure 8: sunflow (PER = 0.5)](image8)

![Figure 9: sunflow: statistics](image9)

We modified the Java sunflow benchmark with Eco syntax. The rendering logic in sunflow resembles the example in Figure 1. Our benchmarking execution renders 9 images (IMGs=9), and the resolution and depth are implemented as first-class mode case with identical definitions as in Figure 1.

The main difference between Figure 1 and our benchmark is that we have refined the demand characterization, as in Figure 5. It not only considers the number of images as demand indicators, but also the number of pixels in the images. In the new scheme, the change of resolution would alter both the demand sum and demand gauge, because the adoption of a lower resolution in rendering an image reduces both the work that has been completed (workdone), and our projection of the work ahead (workleft).

1 http://sunflow.sourceforge.net/
2 http://jspider.sourceforge.net/
3 http://www.javagrande.org/
4 http://xalan.apache.org/
5 http://xmlgraphics.apache.org/batik/
6 http://www.alcpu.com/CoreTemp
We first execute the unmodified Java program and use its battery consumption as the baseline supply. This number is used for setting the BUDGET constant in Figure 5. In our experiment, \texttt{BUDGET} = 20970mWh. We next execute the \texttt{Eco} program with 90\%, 70\%, 50\% of the \texttt{BUDGET}, with the time series of supply gauge (blue lines) and demand gauge (red lines) shown in Figure 6, Figure 7, Figure 8, respectively. All data are normalized. All experiments are started with the same battery level. As one can see, battery supply change is near linear. The change in demand happens at discrete time, depending on the updates of \texttt{workdone} and \texttt{workleft} in the program. Each “plateau” in the demand curve represents the processing of one image. As high-resolution rendering takes significantly longer time, the time series is also useful in identifying mode changes.

Sustainability—the scenario where the supply consistently meets the demand—is maintained in all three scenarios: observe the supply curve and the demand curve in all three figures follow similar patterns of decline. Figure 9 reports the statistics of our experiments. Observe that in all cases, \texttt{Eco} runtime is capable of meeting the budget (i.e., the “supply used” is less than the “supply set”). The \#hi, \#mid, \#lo columns in the table list the number of images rendered with high/mid/low resolutions. Predicably, more images are rendered with lower resolutions as the supply is set lower.

4.2 Temperature-Aware Programming (xalan)

We use \texttt{xalan} to transform 17 XML files with 8000 iterations. We modify the original Java program as follows. First, we set a fixed temperature threshold to 60$^\circ$C. This is expressed as \texttt{tsupply} (60 – \texttt{temperature}). Second, in between every other file transformation, we allow the CPU to sleep at a fixed interval. A mode case is used for this sleep interval, set with two case members: when the mode is \texttt{hi}, the interval is set at 4 milliseconds, whereas when the mode is \texttt{lo}, the interval is set at 10 milliseconds.

Figure 10 demonstrates the temperature changes of \texttt{xalan}, both in the Java execution and the \texttt{Eco} execution. \texttt{xalan} is a CPU-intensive benchmark. Without any sustainability management, the Java execution drives the temperature over 60$^\circ$C within one minute (a rise of about 15$^\circ$C). The \texttt{Eco} execution on the other hand successfully maintains the CPU temperature within the threshold. Figure 11 may explain the fundamental difference between the two executions, showing different levels of CPU activities (plotted with OS performance monitor). During the Java execution, the CPU utilization is mostly 100\%. The \texttt{Eco} execution however regulates the CPU utilization at a much lower and dynamic level.

The figures here also demonstrates the trade-off between temperature regulation and performance. Without any sleep, the Java execution is able to complete the transformation within about half the time of the \texttt{Eco} execution. This comes with no surprise: the latter execution intentionally slows down itself frequently. Figure 12 demonstrates the change of sleep intervals over time.

4.3 Other benchmarks

We designed a battery-aware variant of \texttt{batik}, processing 27 SVG files with the supply of 8050nwh \times 70\%. Similar to the fashion the \texttt{sunflow} experiment was constructed, 8050nwh is the battery usage when we first executed the unmodified Java program, which we informally call “the budget.” The battery-aware variant thus runs on 70\% of that
achieves the goal of sustainability by matching supply and demand closely. Figure 14 shows the dynamic adaptive behaviors when values of different case members are selected.

In montecarlo, a mode case is defined to adjust the sample size. Specifically, the value in each case member is the difference between the “perfect” sample size (280,000) and the sample size being used for that mode. We use a syntactic sugar to define this mode case (with a large number of cases whose mode names are implicit), in the same effect of defining \( \{ m_0 : 0; \ldots ; m_i : i \times \Delta ; \ldots ; m_n : n \times \Delta \} \) where \( \Delta = 10,000 \). In other words, our benchmark adjusts the sample size by the increment/decrement of \( \Delta \). The execution of the unmodified Java code (for Monte Carlo core algorithm) consumes 2060mwh of battery (“the budget”). The Eco execution operates on 80% of that budget, with results shown in Figure 15 and Figure 16. As Figure 15 suggests, Eco again matches supply and demand closely. Figure 16 shows the change of sample size over time.

We constructed a temperature-aware execution of jspider in Eco. Similar to xalan, the temperature is also set at the fixed threshold of 60°C. We employ a similar strategy for temperature control, by allowing the crawler to sleep at fixed intervals. Unlike xalan where each XML file is processed in a relatively short period of time (less than 0.5 second), each step in crawling—involving requesting a URL, parsing the webpage, and analyzing the links contained in the page—takes significantly longer. This allows us to design with longer intervals of sleeps (but much less frequent than xalan). The interval is implemented as a mode case, with hi mapping to 0.5 second, mid to 1 second, and hi to 2 seconds. Figure 17 shows the temperature results. Figure 17 shows the CPU utilization, and Figure 19 shows the interval change. Typical for a web crawler, there is no “completion” time unless the user stops the execution. Our demand is set (and reset) at fixed time intervals. As a result, all time series shown here for jspider can be viewed as the first 800 seconds of an (infinitely running) execution.

5. PROGRAMMING IDIOMS

We now summarize common programming idioms, many of which we encountered while programming in Eco. The summary—by no means complete—may serve as a first step toward understanding design patterns and micro-patterns of energy-aware software. Since our previous discussions are more focused on demand characterization, the following discussion focuses more on supply.

5.1 Fixed Supply Characterization

Various forms of fixed battery budget characterization can be directly supported:

- **fixed absolute battery budget**, e.g., \( \text{bsupply}(20000) \) says the sustainable block execution can consume up to 20000mwh of battery.
- **fixed relative battery budget**, e.g., \( \text{bsupply}(0.2 \times \text{battery}) \) says the sustainable block execution can consume up to 20% of the remaining battery.
- **maximum battery budget**: clause \( \text{bsupply}(\text{battery}) \) says the sustainable block execution can as much as the entire remaining battery.

Analogously, simple forms of temperature budget characterization are supported:
• **fixed temperature threshold**, e.g., the use of \texttt{tsupply}(70°C) says the sustainable block execution should not increase the CPU temperature to more than 70°C.

• **fixed temperature increase**, e.g., \texttt{tsupply}(20) says the sustainable block execution should not increase the CPU temperature by more than 20°C.

• **fixed temperature increase ratio**, e.g., \texttt{tsupply}(1.1 \times \text{temperature}) says the sustainable block execution should not increase the CPU temperature by more than 10%.

## 5.2 Programming with Battery-Specific Supply Units

The sugared syntax \texttt{bsupply} use battery capacity \texttt{(much)} as the unit of supply, and syntax \texttt{tsupply} uses temperature in Celsius. These units are not hardcoded in \texttt{Eco}. More generally, an \texttt{Eco} programmer may choose any unit intuitive and accurate for her sustainability management, as long as such values can be obtained from underlying OS and hardware.

For example, popular battery-powered consumer computers (laptops and smartphones) typically come with power management modules that can report the remaining operating time (or remaining percentage). A sustainable block that should consume no longer than 20% of the remaining operating time can be programmed as:

```plaintext
1 sustainable {
2   double startROT = 0.2 * getROT();
3   ...}
4 supply (startROT) (startROT - getROT())
5 demand (…) (…)
```

where \texttt{getROT} queries the system battery management module for the remaining operating time.

This example demonstrates an attractive trait of \texttt{Eco}: sustainable blocks are a high-level language abstraction independent of battery-specific units. From a language standpoint, what matters for a \texttt{supply}(e)\texttt{(e')} specification is the relative proportion of the value represented by \texttt{e'} within that represented by \texttt{e}. Indeed, the suitable unit to report battery State-of-Charge (SoC) [13] is dependent on many factors, such as (i) the characteristics of batteries, e.g. lithium-based or nickel-based; (ii) the support of OS, e.g. ACPI-compliant systems and different vendor extensions (such as PowerManager\textsuperscript{7} in Android); (iii) the estimation algorithms. The more user-friendly metrics—remaining percentage, or remaining operating time—are in fact derived metrics from lower-level battery status data e.g. voltage level, current level, and coulomb counts [13]. Under the backdrop of this diverse and fast-changing landscape, the abstraction provided by \texttt{Eco} maintains a level of stability in the presence of technology changes: if next-generation programmers decide the coulomb count is more precise for their energy-aware software development, the programming model advocated by \texttt{Eco} remains the same.

## 5.3 Programming with Custom Battery/Thermal Variations

Battery-powered computer users often observe that the same level of demand that reduces battery from 80% to 78%

\footnote{\texttt{http://developer.android.com}} \texttt{http://developer.android.com} may reduces the same battery from 5% to 1%. Such non-linear battery behaviors are generally minimized by built-in OS/firmware estimators for new batteries, but they may become more pronounced as batteries age. If high precision on battery supply is needed, such variations should not be ignored and can be modeled in \texttt{Eco} as:

```plaintext
1 sustainable {
2   double startB = 0.2 * getB();
3   ...}
4 supply (startB) (startB - getB())
5 demand (…) (…)
```

where \texttt{redress} is a custom-defined function mapping readings from \texttt{getBP} to the “compensated” values. For example, one simplistic \texttt{redress} function would be to map value \texttt{b} to \texttt{b} if \texttt{b} \geq 0.2 and \texttt{b} to 0.9 \times \texttt{b} otherwise. As energy-aware programming becomes more prevalent, we envision such “redress” functions would be part of the “battery profile” available to end programmers.

A similar programming idiom can be applied for non-linear thermal behaviors. According to solid matter physics, the energy to increase the temperature from 30°C to 35°C and that from 300°C to 305°C significantly differ. The range of operating temperatures for CPUs is usually much narrower, so the non-linear effect makes little impact.

## 6. RELATED WORK

Modeling computer systems with supply and demand is mostly known in operating system research. An incomplete list of examples include resource management and scheduling for grid environment [6], device energy management [31], utility computing in clouds [7], energy distribution among users [23]. Recently, the supply/demand model is also used for program runtime tuning [22]. \texttt{Eco} unleashes the power of supply and demand shaping to programmers in the context of energy-aware programming.

Our prior work Energy Types [9] promoted mode-based energy management in programming. The focus of that work is a type system design—orthogonal to \texttt{Eco}—but \texttt{Eco}
inherits the concept of modes from there. EnerJ [27] is another programming model where program data can be labeled as either “precise” or “approximated,” a special form of modes. Neither Energy Types nor EnerJ supports first-class mode cases, nor do they support sustainable programming. Green [4] defines a program approximation framework that guarantees programmer-specified QoS, and performs loop-/function approximations to save energy. Green focuses on QoS guarantees. It neither supports programmer-specified energy budget, nor program sustainability.

Eon [30] is a domain-specific language for sensor networks. Built upon a data-flow model, Eon runtime may adjust data flow paths based on the status of solar production and battery utilization. Eon supports a (hardcoded) form of sustainability: the runtime balances solar and battery, so that the program can run “perpetually.” Eco not only provides a sustainable program runtime, but also a sustainable programming model, where supply and demand are characterized by program elements.

Energy-aware software is an active research area in software engineering, with diverse efforts on program analysis (e.g., [5, 15]), cost estimation (e.g., [28]), debugging (e.g., [2]), software architectures (e.g., [19]), design patterns (e.g., [21, 26]), and compiler optimizations (e.g., [16]).

There is a long history in VLSI architecture, and OS communities to design battery-aware (e.g., [11]) or temperature-aware (e.g., [29]) computer systems. A fundamental challenge is the need to “predict the future,” with solutions ranging from profiling and monitoring [10], machine learning [8], to agent-based modeling [12]. Eco complements existing systems by bringing programmer knowledge into the design space. The approaches are not exclusive, and in the long run, we believe hybrid approaches may further improve the effectiveness.

7. CONCLUSION

This paper describes Eco, the first programming language of its kind to support sustainable programming in Java-like languages. With novel abstractions for supply and demand shaping, Eco brings programmers into the loop of sustainability management. With a surprisingly compact and simple model, Eco promotes battery-aware programming and temperature-aware programming, and more generally, energy-aware software development.

This work is supported by NSF CAREER Award CCF-1054515 and a REU supplement.

APPENDIX

A. OPERATIONAL SEMANTICS

Figure 20 defines the runtime structures, including heap $H$, context $K$, and extended expressions $e$ with push and pop operators. Furthermore, all sustainable blocks and uniform blocks are associated with unique program labels $\ell$. Evaluation context $E$ is useful for defining evaluation order, defined either as a hole $\sigma$, or an expression with a hole inside. Notation $E[e]$ means replacing the whole in $E$ with expression $e$. Given sequence $S = [a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$, we define $\text{dom}(S) \overset{\text{def}}{=} \{a_1, \ldots, a_n\}$. Given two sequences $S_1$ and $S_2$, $S_1 \sqcup S_2 \overset{\text{def}}{=} S_1, S_2$ if $\text{dom}(S_1) \cap \text{dom}(S_2) = \emptyset$; it is undefined otherwise.

Figure 21 defines the operational semantics of Eco, where notation $H, K, e \leadsto H', K', e'$ means $e$ evaluates to $e'$ where heap $H$ updates to $H'$ and context $K$ updates to $K'$. FJ function $\text{mbody}(md, x)$ computes the method body of method $md$ of an object of class $x$, in the form of $x.e$ where $x$ is the formal argument. FJ function $\text{fields}(x)$ computes the sequence of field definitions of class $x$. Notation $e[v/x]$ computes an expression identical to $e$, except every occurrence of $x$ is substituted with $v$.
B. REFERENCES


[17] Igarashi, A., Pierce, B., and Wadler, P. Featherweight java - a minimal core calculus for java and gj. In ACM Transactions on Programming Languages and Systems (1999), pp. 132–146.


