
# Proactive and Adaptive Energy-Aware Programming with Mixed Typechecking

Anthony Canino Yu David Liu
SUNY Binghamton, USA
{acanino1,davidL}@binghamton.edu

## Abstract

Application-level energy management is an important dimension of energy optimization. In this paper, we introduce ENT, a novel programming language for enabling *proactive* and *adaptive* mode-based energy management at the application level. The proactive design allows programmers to apply their application knowledge to energy management, by characterizing the energy behavior of different program fragments with modes. The adaptive design allows such characterization to be delayed until run time, useful for capturing dynamic program behavior dependent on program states, configuration settings, external battery levels, or CPU temperatures. The key insight is both proactiveness and adaptiveness can be unified under a *type system combined with static typing and dynamic typing*. ENT has been implemented as an extension to Java, and successfully ported to three energy-conscious platforms: an Intel-based laptop, a Raspberry Pi, and an Android phone. Evaluation shows ENT improves the programmability, debuggability, and energy efficiency of battery-aware and temperature-aware programs.

***CCS Concepts*** •**Software and its engineering** → *Extra-functional properties*; **Language features**

***Keywords*** Energy Efficiency, Energy-Aware Programming, Type systems

## 1. Introduction

A critical dimension of energy optimization of computer systems is application-level energy management [19, 32, 46, 51, 54, 59, 64, 69], *i.e.*, approaches that apply application-specific knowledge to improve energy efficiency. One promising direction with growing interest is *mode-based energy-aware programming* [32, 59, 64, 69]. Drawing analogy from hardware-level CPU modes, these application-level solutions define alternative application behavior for different energy states. For example, a smartphone programmer may write an App that produces high-resolution images under the "high battery" mode and low-resolution ones under the "low battery" mode.

From the standpoint of programming, *proactiveness* and *adaptiveness* are two central yet often competing goals of mode-based energy-aware programming. A proactive energy-aware programmer may *statically* assign all program components to different modes, and regulate how these components may — and may not — interact with each other [32, 59]. On the extreme end of adaptiveness, an energy-aware program may just be *any* program whose behavior adapts to energy changes, wherever in the program, and whatever program component to adapt to.

***This Paper*** We introduce ENT[1], a novel programming language to facilitate cooperative mode-based energy management between the programmer and the application runtime. The centerpiece of ENT is a type system that *mixes static and dynamic typing*. A program component with its energy behavior known to programmers is statically assigned with a type qualifier indicating the mode in which it is expected to be used, whereas one whose energy behavior is dependent on application or system runtime state is assigned with a dynamic type, indicating its mode will be determined at run time. Static or dynamic, the mode information is used by the type system to regulate interactions between program components, and possibly expose and eliminate a category of "energy bugs," — *e.g.*, a program fragment specifically defined for 'high battery" is accidentally used in the mode of "low battery" — through compile-time or run-time errors.

From an end user's perspective, ENT is a tool to promote both proactive and adaptive energy management, guided by a key insight that the goal of *unifying* and *striking a balance* between the two can be achieved through mixed static and dynamic type checking [17, 29, 34, 35, 48, 50, 61–63, 65–67]. With a type-based approach, enforcing the *law* of mode-based energy management becomes part of the experience of energy-aware programming. The choice between proactiveness and adaptiveness is presented to the programmer through static mode type and dynamic mode type dec-



*PLDI'17*, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062356

[1] Ent is a mixed race of the tree and the human in J. R. R. Tolkien's *Lord of the Rings*.

| features | adaptive support | proactive support |
|---|---|---|
| attributor | determines modes at run time | defines "how" by programmer |
| snapshot expression | determines modes at run time | defines "when" by programmer |
| mode cases | supports mode-alternative behaviors | N/A |
| static mode type declaration | N/A | characterizes by programmer |
| dynamic mode type declaration | delays decision to run time | identifies adaptive hotspot by programmer |
| compile-time type error | N/A | assists in debugging |
| run-time type error | enables exception handling | assists in debugging |

**Figure 1:** A Summary of ENT Programming Abstractions and Type System Features

larations. The questions of *how* and *when* to take into account the dynamic behavior of the application and underlying system runtime are answered through programming abstractions that connect dynamic typing and static typing. A summary of ENT features appears in Figure 1.

ENT is a rigorously defined programming language with a sound type system, but more importantly, ENT is an open-source project[2]. We have used ENT to "upgrade" a number of real-world applications, making them *battery-aware* and *temperature-aware* on three energy-conscious platforms — a mobile computer with conventional hardware configurations, an emerging platform with increasing adoption, Raspberry Pi [11], and an Android phone. In practice, we find the energy behavior of an application fragment frequently depends on runtime heap states, and more so, external factors *e.g.*, the configuration files associated with the application, or the files/database the application interacts with. ENT is uniquely suited for improving *programmability* under these scenarios. We also find the type errors reported by ENT — either in the form of a compile-time error or a run-time exception — valuable for understanding the program structure and its implications on energy behavior, good news for *debuggability*. Last but not least, our case studies show principled mode-based energy management can lead to application-level energy savings.

This paper makes the following contributions:

- an insight of supporting application-level mode-based energy management through a form of mixed typechecking to balance the proactiveness and adaptiveness of energy management.
- a language design and compiler implementation to promote programmability, debuggability, energy efficiency, and ease of use in the development of battery-aware and temperature-aware software.
- an evaluation on three energy-conscious mobile platforms: a widely used mobile computer with Intel chips, a Raspberry Pi, and an Android Phone.

## 2. Motivation: A Tale of Three Programmers

In this section, we motivate the design of ENT through use scenarios.

[2] We provide the compiler, benchmarks, and a technical report containing the full formal system, companion proofs, and experimental data at https://github.com/pl-ent-lang/ent.

***A Running Example*** Imagine Alice, Bob, and Christina are three "green conscious" application programmers who would like to write an energy-aware web crawler. Thanks to real-world applications such as `jspider` (see Section 5 for details), its basic program logic is well-known. Alice, Bob, and Christina all agree their Java implementation resembles Listing 1, when all code highlighted in red — including code in a pair of { and } — is elided. Here, the crawling `Agent` runs in a "discover-check-crawl" loop. It starts with reading the `Site` located by a seed URL, discovering the URL resources used by that site, checking whether the URLs should be crawled through pre-defined configuration `Rule`'s, and iteratively crawling the URLs passing the check.

***Alice: A Java Programmer*** Alice approaches energy-aware programming with two questions: how to be aware of physical energy/battery states, and how to make the program adaptive to the changes in energy states. The first question is orthogonal to programming language design. She quickly found some libraries – such as an application-level wrapper of ACPI [1] — and encapsulated them into a class similar to `Ext` class at Line 60. The second question is more relevant to energy-aware software design. Her design intentions are:

- (A1) if the remaining battery level is 75% or above, or the `Agent` will only crawl "local" `Site`'s, then the crawler will search up to three `depth` levels of nested resources for new `Sites` to be crawled.
- (A2) if battery is 50% - 75% the `Agent` will crawl `Site`'s with no more than 200 resources. Each `Site` is searched up to two `depth` levels of nested resources.
- (A3) if battery is below 50% the `Agent` will crawl `Site`'s with no more than 50 resources. Only immediate resources are searched.

As a first step, Alice can encode the intentions using "if-then-else" case analysis. In this example, it means every occurrence of the use of the `Site` object and the use of the `depth` value need to be guarded by a conditional expression checking the battery state. There are several limitations when this approach scales. First, since each object or primitive value may be used multiple times, the per-use case analysis may grow unwieldy as the number of energy-alternative behaviors scales up. Second, since each case analysis is hard-coded with checking external battery state, there is no statically sound approach to reason about the interactions between program components meant to be used consistently

```
1  modes { energy_saver <= managed; managed <= full_throttle; }
2
3  class Main {
4   public static void main(String[] args) {
5     // arg[0] is configuration file name with filtering rules
6     // arg[1] is seed URL
7
8     Agent da = new Agent(arg[0]);
9     ArraySet urls = new ArraySet (arg[1]);
10    while (true) {
11     ArraySet newbie = new ArraySet();
12     Agent a = snapshot da;
13     foreach(String s : urls) { newbie = a.work(s); }
14     sites.remove(s).add(newbie);
15 }}
16
17 class Agent@mode<?->X> {
18  Rule[] rules;
19  Agent(String file) { rules = parseConfigurationFile(file);  }
20  attributor {
21   if (Ext.battery >= 0.75) return full_throttle;
22   else foreach (r: rules)
23     { if (LocalOnlyRule instanceof r) return full_throttle; }
24   else if (Ext.battery >= 0.50) return managed;
25   else return energy_saver;
26  }
27  Set work(String url) {
28   Site@mode<?> ds = Database.lookupSite(url);
29   Site s = snapshot ds [_,X];
30   return s.crawl(depth); // crawl
31  }
32  Rule[] parseConfigurationFile(String file) { ... }
33  mcase<int> depth = mcase<int> {
34   energy_saver: 1;
35   managed: 2;
36   full_throttle: 3;
37  };
38 }
39
40 class Site@mode<?> {
41  Resources[] resources;
42  attributor {
43   if (resources.length > 200) return full_throttle;
44   else if (resources.length > 50) return managed;
45   else return energy_saver;
46  }
47  Site(String url) { this.resources = discoverLinks(url);  }
48  Resources[] discoverLinks(String url) { ... }
49  Set crawl(int depth) {
50   ArraySet moreWork = new ArraySet();
51   foreach (r: resources) { moreWork.add(r.process(depth)); }
52   return moreWork;
53  }
54 }
55
56 class Database {...} // Global store of found sites
57 class Resource {...} // URL link and data
58 class Rule {...} // filtering rule
59 class LocalOnlyRule extends Rule {...} // local crawl only
60 class Ext {...} // an ENT library class for external contexts
```

**Listing 1:** A Simplified Energy-Aware Web Crawler in ENT

together in the presence of dynamic battery variations. Overall, this is the "wild west" of energy-aware programming, where the principle behind Alice's energy management is buried in the code.

***Bob: An Energy Types Programmer*** To address the challenges faced by Alice, Bob resorts to energy-aware programming with a purely static type system, *e.g.*, Energy Types [32].

Bob concretizes the energy states Alice had in mind by defining 3 modes for his program: `full_throttle`, `managed`, and `energy_saver`, intuitively capturing (A1), (A2), (A3) respectively. Furthermore, he may define a partial order over them, such as in Line 1 in Listing 1. For example, `energy_saver <= managed` says the program runtime executing under the `energy_saver` mode is expected by the programmer to consume less energy than one under `managed`. Informally, we say `energy_saver` is a *lesser* mode than `managed`, and `managed` is a *greater* mode than `energy_saver`.

In Energy Types, modes can be used as type qualifiers. When Bob labels an object with a mode, it means that *the object is expected to be used under said mode*. In programming, this generally translates to a *workload characterization* view that the object's methods and data represent an expected workload, consuming a level of energy. For example, if Bob knows a particular `Site` object contains more than 200 resources at compile time, he may label the type of that object as `Site@mode<full_throttle>`, intuitively saying that crawling over this `Site` is likely to consume a large amount of energy. The only exception to this view is that a mode-aware execution must be booted from some object, such as the `Agent` object in the example. In this case, Bob may associate a mode with an `Agent` based how he wishes to bootstrap the mode-aware execution. For convenience we refer to the two views as *workload mode* and *boot mode*, respectively. Note, however, the two views are still unified under the original definition.

The key invariant enforced by Energy Types is the *waterfall invariant*: an object of mode `energy_saver` cannot invoke a method of an object of `full_throttle`, but the opposite *is* allowed. For example, an `Agent` object with boot mode `energy_saver` indicates the bootstrapping of energy-aware execution with little available energy; hence, it should only interact with `Site` objects that consumes little energy. As object messaging forms a chain, this invariant transitively enforces that all objects reachable from a boot mode are either of the same mode or lesser.

The assistance Bob receives from the static type system is twofold: (B1) any program Bob is able to compile cannot accidentally "equivocate" over the mode of a program fragment, such as declaring an object that belongs to an `full_throttle` mode, and then after the object is passed around method boundaries or aliased, the same object is used as if it belonged to the `energy_saver` mode; (B2) more importantly, any compiled program of Bob conforms to the *waterfall invariant*, helping Bob establish sound mode-based energy management throughout his code.

Energy Types represents an extreme case of *proactive* energy-aware programming: on one hand, Bob's knowledge of his program's energy consumption is taken into account for energy management through the mode type declarations; on the other hand, Bob needs to know the mode of each and every object *at compile time*, by type declaration or by type inference.

On a side note, readers may have noticed that the default granularity of characterizing energy behavior in Energy Types is the *object*. At first glance this may appear counter-intuitive because energy consumption results from code execution, and hence, a method may appear more ideal. However, consider the `crawl` method on Line 49 - 53. Its energy behavior depends on the number of `resources` defined as an object field. In other words, energy consumption should be characterized based on the closure of the method and the

data it uses. In an object-oriented language, an object is the most natural abstraction for a closure. Energy Types further provides mode overriding for finer-grained energy characterization at the level of methods, a feature useful in practice for excluding cheaper methods such as accessors.

***Christina: An* ENT *Programmer*** Christina realizes the mode that best characterizes the energy behavior of an object in many scenarios might be dependent on the dynamic behavior of the application and the underlying system. In particular, there are three scenarios where it is particularly challenging to categorize an object's behaviour with static mode types.

- **state-dependent**: Alice or Bob's wish to assign a `Site` object with an `energy_saver` mode depends on the object containing no more than 50 resources — something that is not known until the first `discoverLinks` method is called during `Site` creation (Line 47).
- **configuration-dependent**: Real-world applications routinely rely on reading external configuration files to instantiate objects. For example, Line 4–15 may be how Bob starts his application. Here, `arg[0]` is the name of a file defining filtering rules. To achieve (A1), Bob needs to parse the file before determining the mode for `Agent`.
- **context-dependent**: Energy goals (A1) - (A3) require that an `Agent` begin the call-chain with the proper mode determined by the available battery level of the system.

Christina recognizes the benefits that Energy Types brings to Bob — such as (B1) and (B2) — but high on her list are a number of features to account for dynamic behaviors:

- (C1) the modes of some objects can be decided at run time.
- (C2) the objects whose modes are decided at compile time and those whose modes are decided at run time should "mesh well": analogous properties such as (B1) and (B2) should still hold in the mixed setting, which we henceforth rename as (C2-1) and (C2-2) respectively.
- (C3) there is a need to support *mode-alternative* behaviors to further improve the adaptiveness of energy-aware software, *e.g.*, alternative behaviors in the presence of mode changes.

(C1) and (C3) can be viewed as complementary in the adaptiveness design space. If we take the classic view where the object provides a *service* and whoever holds a reference to that object is a *client* to that service, (C1) says ENT should provide service-side adaptiveness whereas (C3) calls for client-side adaptiveness. ENT addresses Christina's need (C1) through a pair of programming abstractions: the *attributor* construct and the *snapshot* expression, and (C3) through a programming abstraction called *mode cases*. (C2) is addressed through a type system design with mixed type-checking. We discuss these abstractions in Section 3.

# 3. Energy-Aware Programming in ENT

## 3.1 ENT Programming Abstractions

***Attributors and Snapshotting*** In ENT, if a class is declared to be dynamic — *i.e.*, its class declaration is associated with `@mode<?>` or its variant `@mode<?->X>` — it must be associated with an *attributor*. This code block defines what mode the enclosing object should be assigned with *at run time*. For example, Line 20 - 26 says the `Agent` object is assigned with the `full_throttle`, `managed`, `energy_saver` modes given intentions (A1), (A2), and (A3), respectively. The code within the attributor can be arbitrary Java code — allowing the object to inspect its own fields, the fields of other objects it holds references to, or lower-level system settings — to make a run-time decision for the mode of the object, which is the return value of an attributor.

In addition, ENT supports the *snapshot* expression. Upon the evaluation of **snapshot** $e$, the attributor of object $e$ is evaluated, whose returning mode from here on becomes the mode type for (a copy of) $e$. For example, Line 12 says the type of the `Agent` object is determined for every iteration of the **while** loop. Snapshotting can also be bounded. For example, Line 29 says the resulting snapshot must not have a mode greater than `X`. Otherwise, a run-time exception is thrown.

Philosophically, the attributor design reflects a cooperative decision-making process between the object and its client: the object decides on how its mode should be assigned, based on its view of the runtime state, whereas the client decides on the timing of mode type determination through the snapshot expression.

Snapshotting is beyond a simple evaluation of the attributor — otherwise, one might as well encode the attributor as a standard method. The key insight here is that the snapshot expression is the bridge between static typing and dynamic typing. In this aspect, the snapshot expression can be compared to the type coercion operator prevalent in mixed type systems. At Line 12, it is important to observe that `da` and `a` are assigned with different types in ENT. Variable `da` has type `Agent@mode<?>`, saying the object has the *dynamic mode type*, whereas variable `a` has type `Agent@mode<T>` where `T` may be one of the 3 modes defined in the program. In other words, variable `a` does have a *static* mode type. We will elaborate some subtle issues in this design space in Section 4.

***Mode Cases*** To address (C3), we adopt *mode cases*, a form of tagged union where each tag is a mode name. For example, mode case `depth` between Line 33 and Line 37 says the depth for crawling is 1 if it is used in an object with mode `energy_saver`, 2 if in an object with mode `managed`, and so on. This construct was used in untyped energy-aware programming before [69]. ENT's adoption focuses on how it is integrated with a type system: the elimination of a mode case is based on the enclosing object's mode type. For example, if an `Agent` object is snapshotted to be in the `managed`

```
1  modes { energy_saver <= managed; managed <= full_throttle; }
2  class Agent@mode<?->X> {
3   attributor {
4    if (Ext.battery >= 0.75) return full_throttle;
5    else if (Ext.battery >= 0.50) return managed;
6    else return energy_saver;
7   }
8   Set work(String url) {
9    Site@mode<X> s = new Site@mode<X>(url);
10   return s.crawl(new @mode<X>DepthRule(),
11    new @mode<X>MaxResourcesRule()); // crawl
12  }}
13  Rule@mode<X>[] parseConfigurationFile(String file) { ... }
14 }
15
16 class Site@mode<X> {
17  Resource[] resources;
18  Set crawl(Rule@mode<X> r1, Rule@mode<X> r2) { ... }
19 }
20
21 class DepthRule@mode<X> extends Rule {
22  mcase<int> depth = mcase<int> {
23   energy_saver: 1;
24   managed: 2;
25   full_throttle: 3;
26  };
27 }
28
29 class MaxResourcesRule@mode<X> extends Rule {
30  mcase<int> maxresources = mcase<int> {
31   energy_saver: 50;
32   managed: 100;
33   full_throttle: 200;
34  };
35 }
```

**Listing 2:** A Modified Energy-Aware Web Crawler in ENT

mode, `depth` is eliminated to 2. On the high level, this design allows the program to exhibit mode-alternative behaviors based on the mode of the enclosing object.

***Generic Modes and Internal-External Mode Distinction*** ENT fully supports *generic modes*, in that the energy mode of an object/method may be characterized to be *parametric* to its instantiation/invocation. This feature is intuitively analogous to Java generics, except that the generic type parameters in ENT range over modes.

More relevant to the design of mixed type checking is how ENT allows the type determined at run time to be generic, which in turn can be used by the static system. For example, the declaration of `@mode<?->X>` at Line 17 says that any `Agent` object is instantiated as dynamic, but within the scope of the object itself, one may view that type as a *static* generic type, `X`. Within the scope of the class, `X` may be used just as any type parameter. For example, Line 30 says the upper bound for snapshotting is `X`, even though we do not know the concrete mode for `X` when the program is written. This illustrates one benefit of generic mode programming: it helps the programmer correlate mode declarations at different program points without the need to know the concrete modes for each, yielding highly "generic" or "templated" code in nature.

Another implication of this novel design is that more program fragments can be statically typed, henceforth potentially catching "energy bugs" at compile time. For example, observe that at Line 30, we can *statically* guarantee that this messaging always conforms to the waterfall invariant, *i.e.*, the energy mode of the `Agent` object itself is always in the same mode as or a greater mode than that of `s`.

Type-theoretically, our system achieves *type distinction* between the view of the object from itself ("internal") and the view of the object from its client ("external"), which we will revisit in Section 4.

***Mode Co-Adaptation*** The combination of generic modes and internal-external mode distinction is powerful in supporting some expressive programming idioms. Concretely, multiple objects may need to "co-adapt" their modes, *i.e.*, when one is snapshotted to a particular mode, the parties it interacts with should also be adapted to objects of the same mode. This pattern is known to be useful in energy-aware programming, where program data flow changes in the presence of a change of energy states [64]. Co-adaptation in ENT can be supported by declaring one representative object as dynamic, and its interacting parties as mode cases which are eliminated based on the internal mode parameter of the dynamic object.

Listing 2 shows this co-adaptation pattern in practice. As opposed to our original `jspider` example, the code may be structured so a `Site` manages crawling using a pair of `Rules` (Line 11, 18), each with their own mode-alternative settings (Line 22 - 26, Line 30 - 34). We use the `Agent` as the representative object to determine what mode `Site`, `DepthRule`, and `MaxResourcesRule` should operate under, effectively co-adapting all three to have the same mode when an `Agent` is snapshotted.

***Method-Level Mode Characterization*** Our discussion so far has used the object as the granularity for energy mode characterization. ENT in addition supports method-grained mode characterization through method overriding.

Syntactically, method-level mode characterization is similar to its class-level counterpart. The `@mode` declaration that we previously introduced at the class level may also appear at the method level. To typecheck a method invocation that involves a method with an overridden mode type, we use the mode type associated with the method to enforce the waterfall invariant. For example, in Listing 3, the mode type annotation for `mediaCrawl` at Line 29 states that regardless of the mode of a `Site`, searching and downloading all multimedia is always an expensive operation and requires the `full_throttle` operating mode. As a result, a compile-time error is issued at Line 4.

Generic modes at the method level are also supported. In the same example above, both the mode characterization of the method `generateRules` and that of its returning `Rule[]` object are defined as that of its argument `Site` object. This is useful in capturing a programming idiom where the mode characterization of a method is dependent on its arguments. Note that Line 6 typechecks, because generic modes are handled in ENT in a similar fashion as Java generics.

We also support attributors at the method level, allowing method-level characterization to happen at runtime. Consider the `saveImages` method on Lines 20- 21. The amount of work `saveImages` must perform depends upon the input into the method, analogous to object's whose work-

```
1  class Agent@mode<managed> {
2    Set work(String url) {
3      Site s = Database.lookupSite(url);
4      if (Env.mediaSearch) { return s.mediaCrawl(); }
5      else {
6        Rule[] rules = generateRules(s);
7        return s.crawl(rules);
8      }
9    }}
10
11   @mode<Y> Rule@mode<Y>[] generateRules(Site@mode<Y>) { ... }
12
13   Set @mode<?->X> saveImages(Site s)
14     attributor {
15       if (s.parsedimgs.length > 20) return full_throttle;
16       else if (s.parsedimgs.length > 10) return managed;
17       else return energy_saver;
18     }
19   {
20     JPEGWriter@mode<X> writer = new JPEGWriter@mode<X>();
21     foreach (Image i : s.parsedimgs) { writer.write(s); }
22     ...
23   }
24 }
25 class Site@mode<X> {
26   Resources[] resources;
27   Image[] parsedimgs;
28   Set crawl(Rule@mode<X>[] rules) { ... }
29   @mode<full_throttle> Set mediaCrawl() { ... }
30 }
31 class JPEGWriter {...} // jpeg image writer
```

**Listing 3:** `jspider` Method-Level Mode Characterization.

| | | | |
|---|---|---|---|
| $P$ | ::= | $D\ \overline{C}$ | *program* |
| $D$ | ::= | $\overline{\mathtt{m} \leq \mathtt{m}}$ | *mode declaration* |
| $C$ | ::= | **class** c $\Delta$ **extends** c$\{\overline{F}\ \overline{M}\ A\ \}$ | *class* |
| $F$ | ::= | $T\ \mathtt{fd} = e$ | *field* |
| $M$ | ::= | $T\ \mathtt{md}(\overline{T}\ \overline{\mathtt{x}})\{e\}$ | *method* |
| $A$ | ::= | $e$ | *attributor* |
| $e$ | ::= | $\mathtt{x} \mid e.\mathtt{fd} \mid$ **new** $\mathtt{c}\langle\iota\rangle \mid e.\mathtt{md}(\overline{e}) \mid (T)e$ | *expression* |
| | \| | **snapshot** $e\ [\eta, \eta] \mid$ **mcase**$\langle T\rangle\{\overline{\mathtt{m} : e}\} \mid e \triangleright \eta$ | |
| m | $\in$ | **MCONST** | *mode name* |
| c | $\in$ | **CN** $\cup$ {Object, Main} | *class name* |
| md | $\in$ | **MN** $\cup$ {main} | *method name* |
| x | $\in$ | **VAR** | *variable name* |
| $T$ | ::= | $\mathtt{c}\langle\iota\rangle \mid$ **mcase**$\langle T\rangle$ | *programmer type* |
| $\iota$ | ::= | $\overline{\eta} \mid ?, \overline{\eta}$ | *object parameter list* |
| $\eta$ | ::= | $\mathtt{m} \mid \mathtt{mt} \mid \top \mid \bot$ | *static mode* |
| mt | | | *mode type variable* |
| ? | | | *dynamic mode type* |
| $\omega$ | ::= | $\eta \leq \mathtt{mt} \leq \eta'$ | *constrained mode* |
| $\Delta$ | ::= | $? \rightarrow \omega, \overline{\Omega} \mid \Omega$ | *class parameter list* |
| $\Omega$ | ::= | $\overline{\omega}$ | *constrained mode list* |

**Figure 2:** Abstract Syntax: Terms and Types

| | | | |
|---|---|---|---|
| $\mu$ | ::= | $\eta \mid ?$ | *mode* |
| $\tau$ | ::= | $T \mid \exists\omega.\tau \mid \mathtt{modev}$ | *type* |
| $\Gamma$ | ::= | $\overline{\mathtt{x} : \tau}$ | *typing environment* |
| K | ::= | $\overline{\eta \leq \eta'}$ | *constraints* |

**Figure 3:** Type System Elements

load depends upon its internal state. We support this with a method-level attributor on Line 14 - 18. In our case studies, we found that nearly all uses of method-level attributors can be refactored away, either through the Strategy design pattern [37] or though generic methods. Nonetheless, this convenience feature may lead to more readable programs, which is therefore supported in the current compiler.

## 4. A Formal Core for Ent

The abstract syntax of ENT is defined in Figure 2, on top of Featherweight Java (FJ) [43]. We use FJ function $\mathtt{mtype}(\mathtt{md}, T)$ to compute the signature for method md of object of type $T$, in the form of $\overline{T} \rightarrow T$, $\mathtt{mbody}(\mathtt{md}, T)$ to compute the body for method md of object of type $T$ in the form of $\overline{\mathtt{x}}.e$, and $\mathtt{fields}(T)$ to compute the field signature for object of type $T$, in the form of $\overline{T}\ \overline{\mathtt{fd}}$. The root of inheritance hierarchy is a class named Object. The bootstrapping expression is enclosed in class Main under method main. We omit constructors. We use the standard $X$,$Y$ notation to represent the concatenation of sequences $X$ and $Y$, and $\epsilon$ for the empty sequence.

ENT-specific terms include the **modes** declaration $D$, the **attributor** with body $A$, the **snapshot** expression, the mode case expression **mcase**$\langle T\rangle\{\overline{\mathtt{m} : e}\}$, and the mode case elimination expression $e \triangleright \eta$. In the concrete syntax, mode case elimination for local field access is implicitly projected based on the mode of the enclosing object. For example, in Line 30 of Figure 1, the depth field access is represented as depth $\triangleright$ X where X is declared in Line 17 of the same figure. In addition, we define $\mathtt{abody}(T)$ to compute the body of the attributor for object of type $T$.

An ENT programmer type $T$ can either be an object type $\mathtt{c}\langle\iota\rangle$ or a mode case type **mcase**$\langle T\rangle$. For example, object type Agent@mode<managed> in the concrete syntax is represented as $\mathtt{Agent}\langle\mathtt{managed}\rangle$ in the formal system, whereas Agent@mode<?> is represented as type $\mathtt{Agent}\langle?\rangle$. Our system fully supports generic modes, with expressive power on par with System $F_{<:}$ [28]. Additional mode type parameter instantiations may appear in the rest of the mode parameter list $\iota$ for $\mathtt{c}\langle\iota\rangle$. Correspondingly, the class-level mode parameter list is defined as $\Delta$. For example, the class parameter list for Agent of the running example is represented as $? \rightarrow \bot \leq \mathtt{X} \leq \top$.

### 4.1 The Type System

Type system elements are defined in Figure 3. In addition to the programmer type, we use constrained existential types $\exists\omega.\tau$ for typing the **snapshot** expression, and type modev for typing the attributor. Details of these two types will be explained shortly.

For an object of $\mathtt{c}\langle\iota\rangle$, the mode of this object is the first element in $\iota$. Formally, we define $\mathtt{omode}(\mathtt{c}\langle\iota\rangle) \triangleq \mu$ if $\iota = \mu, \overline{\eta}$. On the class level we analogously define function cmode over the class parameter list where $\mathtt{cmode}(? \rightarrow \omega, \Omega) \triangleq ?$ and $\mathtt{cmode}(\omega, \Omega) \triangleq \omega$. Finally, we define the parameter list of a class as $\mathtt{param}(\overline{\eta \leq \mathtt{mt} \leq \eta'}) \triangleq \overline{\mathtt{mt}}$ and $\mathtt{param}(? \rightarrow \omega, \Omega) \triangleq \mathtt{mt}, \mathtt{param}(\Omega)$ if $\omega = \eta \leq \mathtt{mt} \leq \eta'$. Notation $\Delta\{\iota/\iota'\}$ is standard point-wise type variable substitution.

***Expression Typing*** Selected expression typing rules are defined in Figure 4. Judgment $\Gamma; \mathtt{K} \vdash e : \tau$ says expression $e$ has type $\tau$ under *typing environment* $\Gamma$ and *constraint set* K. All rules are implicitly parameterized by the (immutable) program $P$. We define $\Gamma(x)$ as $\tau_i$ where $i$ is the right most position in $\Gamma$ where $x_i : \tau_i$ occurs. We define $\mathtt{FV}(e)$ as the set of free variables in $e$. Each element $\eta <: \eta'$ in the constraint set says mode $\eta$ represents a lesser mode than the mode of $\eta'$. Note that such mode may either be a mode constant that appears in $D$ or a mode type parameter. We define

$$\text{(T-New)}\ \frac{\iota = ?, \iota' \text{ iff } \textbf{class } \mathtt{c}\ \Delta \cdots \in P \text{ and } \mathtt{cmode}(\Delta) = ? \quad \mathtt{K} \looparrowright \mathtt{cons}(\Delta)}{\Gamma; \mathtt{K} \vdash \textbf{ new } \mathtt{c}\langle\iota\rangle : \mathtt{c}\langle\iota\rangle}$$

$$\text{(T-Msg)}\ \frac{\Gamma; \mathtt{K} \vdash e : T_0 \quad \mathtt{mtype}(\mathtt{md}, T_0) = \overline{T} \to T \quad \Gamma; \mathtt{K} \vdash \overline{e} : \overline{T} \quad \mathtt{sfall}(T_0, \Gamma(\textbf{this}), \mathtt{K})}{\Gamma; \mathtt{K} \vdash e.\mathtt{md}(\overline{e}) : T}$$

$$\text{(T-Snapshot)}\ \frac{\Gamma; \mathtt{K} \vdash e : \mathtt{c}\langle ?, \iota\rangle \quad \omega = \eta_1 \le \mathtt{mt} \le \eta_2}{\Gamma; \mathtt{K} \vdash \textbf{snapshot } e\ [\eta_1, \eta_2] : \exists\omega.\mathtt{c}\langle \mathtt{mt}, \iota\rangle}$$

$$\text{(T-MCase)}\ \frac{\overline{\mathtt{m}} = \mathtt{modes}(P) \quad \Gamma; \mathtt{K} \vdash e_i : T \text{ for all } i}{\Gamma; \mathtt{K} \vdash \textbf{mcase}\langle T\rangle\{\overline{\mathtt{m} : e}\} : \textbf{mcase}\langle T\rangle}$$

$$\text{(T-ElimCase)}\ \frac{\Gamma; \mathtt{K} \vdash e : \textbf{mcase}\langle T\rangle \quad \eta \in \mathtt{modes}(P) \text{ or } \eta \text{ appears in } \mathtt{K}}{\Gamma; \mathtt{K} \vdash e \rhd \eta : T}$$

**Figure 4:** Selected Expression Typing Rules

$$\begin{array}{rcl}
o.\mathtt{md}(\overline{v}') & \xrightarrow{\mathtt{m}} & \mathtt{cl}(\mu, e\{\overline{v}'/\overline{\mathtt{x}}\}\{o/\textbf{this}\}) \\
 & & \text{if } \mathtt{dfall}(o, \mathtt{m}) \\
\textbf{snapshot } o\ [\mathtt{m}_1, \mathtt{m}_2] & \xrightarrow{\mathtt{m}} & \textbf{check}(\mathtt{abody}(\mathtt{c}\langle ?, \iota\rangle)\{o/\textbf{this}\}, \mathtt{m}_1, \mathtt{m}_2, o) \\
 & & \text{if } \mu = ? \\
\textbf{check}(\mathtt{m}', \mathtt{m}_1, \mathtt{m}_2, o) & \xrightarrow{\mathtt{m}} & \mathtt{obj}(\alpha', \mathtt{c}\langle \mathtt{m}', \iota\rangle, \overline{v}) \\
 & & \text{if } \emptyset \looparrowright \{\mathtt{m}_1 \le \mathtt{m}', \mathtt{m}' \le \mathtt{m}_2\}, \alpha' \text{ fresh}
\end{array}$$

For all rules: $o = \mathtt{obj}(\alpha, \mathtt{c}\langle\mu, \iota\rangle, \overline{v}), \mathtt{mbody}(\mathtt{md}, \mathtt{c}\langle\mu, \iota\rangle) = \overline{\mathtt{x}}.e$.

**Figure 5:** Selected Reduction Rules

constraint entailment $\mathtt{K} \looparrowright \mathtt{K}'$, which holds iff the reflexive and transitive closure of $\mathtt{K}' \cup D$ is a subset of that of $\mathtt{K} \cup D$. We define $\mathtt{cons}(\overline{\eta \le \mathtt{mt} \le \eta'}) \triangleq \bigcup\{\eta \le \mathtt{mt}, \mathtt{mt} \le \eta'\}$ and $\mathtt{cons}(? \to \omega, \Omega) \triangleq \{\eta \le \mathtt{mt}, \mathtt{mt} \le \eta'\} \cup \mathtt{cons}(\Omega)$ if $\omega = \eta \le \mathtt{mt} \le \eta'$. Function $\mathtt{modes}(P)$ is defined as all modes appearing in $D$, where $P = D\ \overline{C}$.

In (T-Snapshot), the post-snapshot object is assigned with *some* static mode type. This notion of "some but unspecific" mode is well aligned with existential types. Since the **snapshot** expression may check the lower bound and the upper bound of the mode, any successful **snapshot** will yield an object whose mode is constrained. As a result, we use *bounded existential types*. All existential types are alpha-equivalent with respect to bounded variables.

In (T-Msg) the `sfall` predicate represents the waterfall invariant, which we formally define:

**Definition 1** (Static Waterfall Invariant)**.** $\mathtt{sfall}(T, T', \mathtt{K})$ *holds iff* $\mathtt{K} \looparrowright \{\mathtt{omode}(T) \le \mathtt{omode}(T')\}$.

This definition says that the receiver's mode must be equal to or less than the sender's mode. Observe that the constraint form requires that no ? may appear on either end of $\le$; hence, the invariant here implicitly requires that objects with a dynamic type must not be sent messages directly. Sending a message to an object with unknown energy mode characterization is undesirable as it may violate the intuition of mode-based energy management, *i.e.*, the receiver happens to be in a state only suitable for a `full_throttle` execution whereas the message sender is in mode `energy_saver`.

Judgment $\mathtt{K} \vdash \tau <: \tau'$ says $\tau$ is a subtype of $\tau'$ under constraint set K. Subtyping rules are standard: we support FJ nominal subtyping, existential introduction, existential elimintion, subtyping reflexivity, and subtyping transitivity. The only rule that is ENT specific is mode case subtyping; a mode case type $\textbf{mcase}\langle\tau\rangle$ is a subtype of $\textbf{mcase}\langle\tau'\rangle$ iff $\tau$ is a subtype of $\tau'$.

***Program Typing*** A program $P = D\ \overline{C}$ is well-typed iff $D$ forms a latice and each class in $\overline{C}$ is well-typed. Class typing and method typing are largely FJ standard. The only ENT-specific treatment is type distinction. For any class defined as **class** `c` $\Delta \ldots$, we allow `mt` to appear in the scope of any method where $\mathtt{param}(\Delta) = \mathtt{mt}, \iota$. Furthermore, expression **this** will be typed as $\mathtt{c}\langle\mathtt{mt}, \iota\rangle$. Observe that `param` computes the parameter list whose first element is the internal type parameter in the presence of classes with dynamic modes. In contrast, **this** is typed as $\mathtt{c}\langle ?, \iota\rangle$ in the attributor, as attributors are invoked externally.

### 4.2 Operational Semantics

We define operational semantics through relation $e \xrightarrow{\mathtt{m}} e'$ with selected reduction rules defined in Figure 5. Relation $e \xrightarrow{\mathtt{m}} e'$ denotes expression $e$ is single-step reduced to $e'$ where `m` is the *current* mode. We further use $\xrightarrow{\mathtt{m}}_*$ to denote the reflexive and transitive closure of $\xrightarrow{\mathtt{m}}$. Notation $e \Uparrow$ denotes that computation $e$ diverges.

We supplement our language with additional runtime expressions. $\mathtt{cl}(\mathtt{m}, e)$ defines a *closure*, representing the expression $e$ defined in an object of mode `m`. Expression $\mathtt{obj}(\alpha, T, \overline{v})$ is an object value with the unique ID $\alpha$, object type $T$, and field values $\overline{v}$. $\textbf{check}(e, \mathtt{m}', \mathtt{m}'', o)$ performs a run-time check to ensure the mode represented by $e$ for $o$ is greater than $\mathtt{m}'$ and less than $\mathtt{m}''$. A value $v$ can be an object value $o$, a mode name `m`, or a mode case value of the form $\textbf{mcase}\langle T\rangle\{\overline{\mathtt{m} : v}\}$. The initial configuration of reduction for $P$, denoted as $\mathtt{boot}(P)$ is $\mathtt{cl}(\top, e)$, with $e = \mathtt{mbody}(\mathtt{main}, \mathtt{Main}\langle\top\rangle)$.

Snapshoting leads to the evaluation of an object's attributor, and if the resulting mode falls within the range defined by the snapshot expression, it creates a shallow copy of an object with a fixed mode type, illustrated in the reduction of a snapshot and check expression. Our *copy semantics* addresses a key challenge of type soundness: if snapshotting were designed as merely a type cast it would be possible to have two type casted aliases of the same object hold different modes (types). That design would violate non-equivocation in mode-based energy management and result in type unsoundness. Foundationally speaking, copy semantics enforces monotonic type change. We will revisit the performance of copying in Section 5, and the programmability impact of the shallow copy vs. deep copy design in Section 6.3.

The runtime version of waterfall invariant is captured as:

**Definition 2** (Dynamic Waterfall Invariant)**.** $\mathtt{dfall}(o, \mathtt{m})$ *holds iff* $o = \mathtt{obj}(\alpha, T, \overline{v})$ *and* $\emptyset \looparrowright \{\mathtt{omode}(T) \le \mathtt{m}\}$.

This is a condition for the messaging expression to reduce. As we shall see in shortly, this condition always holds for well-typed programs, and is hence redundant.

### 4.3 Properties

We define runtime error *bad cast* and *bad check* as the following:

**Definition 3** (Bad Cast)**.** *Expression* $(T')\texttt{obj}(\alpha, T, \overline{v})$ *is a bad cast iff* $\emptyset \vdash T <: T'$ *does not hold.*

**Definition 4** (Bad Check)**.** *Expression* $\textbf{check}(\texttt{m}, \texttt{m}', \texttt{m}'', o)$ *is a bad check iff* $\emptyset \not\hookrightarrow \{\texttt{m}' \leq \texttt{m}, \texttt{m} \leq \texttt{m}''\}$ *does not hold.*

We establish type soundness through type preservation and progress lemmas:

**Lemma 1** (Preservation)**.** *If* $\Gamma; \texttt{K} \vdash e : \tau$*,* $\texttt{omode}(\Gamma(\textbf{this})) = \texttt{m}$*,* $e \stackrel{\texttt{m}}{\Longrightarrow} e'$*, then* $\Gamma; \texttt{K} \vdash e : \tau$*.*

**Lemma 2** (Progress)**.** *Suppose* $\Gamma; \emptyset \vdash e : \tau$*,* $\texttt{FV}(e) = \emptyset$*, and* $\texttt{omode}(\Gamma(\textbf{this})) = \texttt{m}$ *then either* $e \stackrel{\texttt{m}}{\Longrightarrow} e'$ *for some* $e'$*, or* $e$ *is a value, or a bad cast or a bad check exists in the parsing tree of* $e$*.*

Now we can establish type soundness:

**Theorem 1** (Type Soundness)**.** *If* $P$ *is well-typed and* $\texttt{boot}(P) = \texttt{cl}(\top, e)$*, then either* $e \stackrel{\top}{\Longrightarrow}_* v$*, or* $e \Uparrow$*, or* $e \stackrel{\top}{\Longrightarrow}_* e'$ *and* $e'$ *has a bad cast or bad check in its parsing tree.*

In particular, a well-typed program cannot get stuck over a messaging expression due to a violation of the `dfall` invariant. We explicitly state this as a corollary. Let us say $\texttt{cl}(\texttt{m}_0, e_0)$ is a *sub-redex* of reduction $e \stackrel{\texttt{m}}{\Longrightarrow} e'$ iff $e_0 \stackrel{\texttt{m}_0}{\Longrightarrow} e_0'$ is a sub-derivation of $e \stackrel{\texttt{m}}{\Longrightarrow} e'$. We next state an important property of ENT.

**Corollary 1** (Waterfall Invariant Preservation with Mixed Typing)**.** *If* $P$ *is well-typed,* $\texttt{boot}(P) = \texttt{cl}(\top, e)$*,* $e \stackrel{\top}{\Longrightarrow} \ldots e_1 \stackrel{\top}{\Longrightarrow} e_2$*, and* $\texttt{cl}(\texttt{m}, o.\texttt{md}(\overline{v'}))$ *is a sub-redex of* $e_1 \stackrel{\top}{\Longrightarrow} e_2$*, then* $\texttt{dfall}(o, \texttt{m})$ *holds.*

Observe that run-time errors are never delayed to messaging time. If any potential violation may happen due to dynamic typing, a run-time error would result from a bad check, *i.e.*, at snapshotting time.

## 5. Implementation and Evaluation Settings

***Compiler*** ENT was implemented as an extension of Java 7 using Polyglot [55]. It supports all syntactic and semantic features presented in the paper.

Two of our features require additional information to be tracked in our runtime. First, mixed typechecking requires run-time tagging of objects with modes. In ENT, only dynamic objects (*i.e.*, those instantiated from a class declared with `?`) need to be tagged, a small portion on the heap. Second, we optimize our cloning by adopting a lazy copying strategy for snapshotting. A physical object copy is only needed if a dynamic object is copied more than once. This is tracked in the metadata of the dynamic object.

| name | description | System | CLOC | ENT Changes | % Energy Overhead |
|---|---|---|---|---|---|
| `crypto` [60] | RSA encryption | A,B | 381 | 46 | 0.40% |
| `findbugs` [18, 42] | static analyzer | A | 147896 | 55 | 0.18% |
| `jspider` [4] | web crawler | A | 9194 | 49 | -0.56% |
| `jython` [20] | compiler | A | 215749 | 33 | 1.17% |
| `pagerank` [21, 22] | graph vertex ranking | A | 157 | 49 | 1.07% |
| `sunflow` [20] | renderer | A,B | 21946 | 76 | 2.62% |
| `xalan` [20] | transformer | A | 169927 | 33 | 3.41% |
| `camera` [12, 13] | picture timelapse | B | 143 | 40 | -6.54% |
| `video` [12, 13] | video recording | B | 115 | 40 | 0.21% |
| `javaboy` [3] | emulation | B | 6492 | 38 | -0.70% |
| `batik` [20] | rasterizer | A | 179284 | 225 | -7.47% |
| `newpipe` [7] | YouTube streaming | C | 8424 | 51 | 2.37% |
| `duckduckgo` [2] | web browser | C | 13802 | 78 | -2.39% |
| `soundrecorder` [15] | sound encoding | C | 1090 | 118 | -1.84% |
| `materiallife` [5] | simulation rendering | C | 1705 | 63 | 0.44% |

**Figure 6:** ENT benchmark descriptions and statistics.

In summary, each dynamic object contains two pieces of additional metadata: its mode tag, and a boolean to indicate whether it has been snapshotted before. Each post-snapshotting dynamic object copy contains one piece of metadata: its mode tag. Each generic object contains a mapping from type parameters to mode tags.

***Target Platforms*** Our compiler has been successfully ported to three mobile platforms:

- System A: Intel i5 CPU laptop with 4GB RAM, Ubuntu 14.04 LTS OS, and Java 1.8 JVM
- System B: Raspberry Pi 2 Model B with 1GB RAM, Raspbian Jessie OS, and Java 1.8 JVM
- System C: Google Nexus 5X, Android 6.0 Marshmellow OS, Android Runtime (ART) 2.1.0

Each platform represents a different, but energy critical, environment for application development. System A and System C are widely used for energy evaluation. System B, the Raspberry Pi (Pi for short), is an emerging platform that shares some traits of traditional embedded systems on one hand, while on the other hand emerges as an "inexpensive computer" for *application-level* software development [8, 9], with broader impact on, *e.g.*, bringing technologies to developing countries and assisting in education.

***Benchmarks*** We have upgraded 15 Java applications into ENT, detailed in Figure 6. For System A, our primary selection criterion is diversity in application characteristics where energy consumption is known to be relevant. For example, some are data-intensive (*e.g.*, `pagerank`), while others are computation-intensive (*e.g.*, `sunflow` and `crypto`). We also aimed at choosing applications using different system resources, such as file I/O (*e.g.*, `batik`) and network (*e.g.*, `jspider`).

For System B, we have developed benchmarks in two categories. First, we developed 3 Pi-specific ENT benchmarks. `camera` is repeatedly takes a picture for a given interval, designed to model time-lapse monitoring using the Pi. `video` takes a continuous stream of video. Both applications monitor continuously, and represent typical real-world use cases of the Pi [8, 9]. We also adapted `javaboy`, a small Gameboy emulator, to the Pi, inspired by the Pi's use in various handheld entertainment projects [10]. Second, considering the trajectory of Pi development is to embrace general-

| name | workload attribution by | energy_saver | managed | full_throttle | QoS adjustment | energy_saver | default (managed) | full_throttle |
|---|---|---|---|---|---|---|---|---|
| batik | file size | 16KB | 261KB | 2MB | image resolution | 512x512 | 1024x1024 | 2048x2048 |
| findbugs | code base (classes) | drjava(5363) | JavaRT(20136) | jBoss(56704) | analysis effort | min | default | max |
| jspider | site resources | 89 | 1058 | 1967 | spidering depth | 3 | 4 | 5 |
| pagerank | graph (number nodes) | cnr-2000(325557) | eswiki-2013(972933) | frwiki-2013(1352053) | minimum change | 0.01 | 0.001 | 0.0001 |
| sunflow | scene instances | 3 | 6 | 8 | anti-aliasing samples | 1/4 | 1/4 - 4 | 1/4 - 16 |
| crypto | file size | 1MB | 2MB | 4MB | encryption key strength | 768 | 1024 | 1280 |
| camera | picture resolution | 720x480 | 1280x720 | 1920x1080 | timelapse interval | 500ms | 1000ms | 1500ms |
| video | video resolution | 480p | 720p | 1080p | frames per second | 10 | 20 | 30 |
| javaboy | ROM size | 64KB | 512KB | 1MB | screen magnification | 2x | 4x | 6x |
| newpipe | video length | 2.5 min | 6.5 min | 16 min | stream resolution | 144p | 240p | 360p |
| duckduckgo | search queries | 8 | 16 | 24 | search quality | none | javascript | autosearch / javascript |
| soundrecorder | recording length | 3 min | 4 min | 5 min | sample rate (kHz) | 8 | 24 | 48 |
| materiallife | simulation population | 1000 | 2000 | 5000 | frame rate | 5 | 10 | 15 |

**Figure 7:** ENT Benchmark Settings

purpose applications and project itself as "just an (inexpensive) computer", we also successfully ported `crypto` and `sunflow` to the Pi, a stress test on the future adoptability of Pi for security and graphics applications.

Lastly, we ported 4 real-world Android Apps for System C into ENT applications. We selected Android Apps based on the following criteria: (1) They must be open source. (2) They must be real-world Apps that may be downloaded from the original authors — either on the Google Play or F-Droid store — and used on Android. `newpipe` is a light-weight YouTube streaming App, `duckduckgo` is an anonymous web browser, `soundrecorder` is a recording App for small sound snippets, and `materiallife` is a animated simulation of Conway's game of life. As Android applications are typically driven by user interaction, we used the RERAN [38] framework for recording and repeating experiments.

***Battery/Temperature Queries and Energy Measurements*** We updated each System A benchmark to support battery-aware programming and temperature-aware programming. The benchmarks for System B and C are updated for battery-aware programming only; the CPUs associated with System B and System C are low-power in general, with no prior report of thermal concerns that we know of.

ENT is shipped with a utility interface named `Ext` to query the battery level and CPU temperature of the underlying system. Our current implementation for System A includes an ACPI [1] wrapper for battery and temperature queries. The Pi as of now does not have a built-in programming interface for querying battery levels, so for System B, the battery level change is simulated. This is a fast-developing landscape where API support may happen in the near future, as we are aware of some initial efforts on battery monitoring through third-party boards [6, 14]. System C battery levels are queried through Android's `BatteryManager`.

For evaluation, energy consumption on System A was measured using jRAPL [49], a Java library that supports profiling programming using Intel's Running Average Power Limit (RAPL) support [33]. On System B and C, we measured energy consumption of devices using a Watts Up? Pro power monitor [16]. System B experiments were run on a Pi with a keyboard, mouse, HDMI monitor, and ethernet cable connected. All experiments were run using the respective systems default power governors.

***Data Collection*** We ran each experiment — as described soon — 11 times for System's A and B, discarding the first run to account for JIT compilation and averaging the rest of the results. We report the relative standard deviation for our medium and large experiments. System A's relative standard deviation is within 2% for 93% of our experiments and within 3% for 99% of our experiments, with the exception of `batik` which exhibited high deviation due to its relatively low energy consumption ($< 10$ J). System B's is within 2% for 100% of experiments and 3% for 100% of experiments. We ran each experiment 10 times for System C, rebooting the App each time. System C exhibited higher relative standard deviation — within 2% for 84.3% of experiments, 3% for 91.5% of experiments, and within 5% for 94.7% of experiments. We attribute the higher deviation of System C to the reliance on external factors such as internet response and touch interaction. While RERAN provides a framework for repeating touch input, there is still a level of non-determinism involved with running Apps. We report the raw data of our experiments in the supplemental material.

# 6. Evaluation

## 6.1 Battery- and Temperature-Aware Benchmark Design

We structured benchmarks in 2 different ways for battery-aware programming, conducting 2 separate evaluations:

- (E1) A "battery-exception" program. We structure the battery-aware application in a way to highlight the importance of ENT in throwing `EnergyException`'s in the presence of insufficient battery. The `EnergyExeption` is thrown when waterfall invariant is violated at run time.
- (E2) A "battery-casing" program. We design the battery-aware application with mode cases, so that in the presence of different battery levels, the application may exhibit adaptive behaviors by selecting different cases in the mode cases.

For applications where temperature-aware programming is relevant, we further restructure the benchmark in the following way:

- (E3) A "temperature-casing" program. We design the temperature-aware application with mode cases to exhibit alternative behaviors in the presence of CPU temperature change.

We define 3 modes for (E1) and (E2) experiments: `energy_saver`, `managed`, and `full_throttle`. For (E3), we define `safe`, `hot`, and `overheating` as modes.

In ENT applications, there is often one "entry" object (such as the `Agent` object in the running example) that queries battery levels or temperatures through its attributor. Recall in Section 2 we call the mode of this object the *boot mode*, and the modes for the rest of the objects the *workload mode*.

***Attributor Definition*** In our experiments of (E1) and (E2), the boot modes of `energy_saver`, `managed`, and `full_throttle` are set at battery levels of 40%, 70%, and 90%, respectively. For (E3) runs, the boot modes of `safe`, `hot`, and `overheating` are set when the current CPU temperature is below 60C, 60-65C, or above 65C, respectively. We set the boot mode for each benchmark using a battery-aware attributor for (E1) and (E2), and a temperature-aware attributor for (E3).

For workload modes, the attributor definitions for each benchmark can be found in Columns 2-5 of Figure 7. Generally for battery-aware programming, larger or more complex objects are labeled with `full_throttle`, reflecting the fact that the programmer expects them to be in a mode with the most available battery. For many benchmarks that come from the DaCapo benchmark suites [20], the application can often be configured into small/medium/large settings, which is also used for defining the mode for our workload objects. In cases where the original benchmark only comes with one default data size, our attributor defines the `managed` mode — the middle mode of the 3 — for that default workload size, and chose a reasonable step above or below for `energy_saver` and `full_throttle` modes respectively. Note that the decisions made for the attributor definition does not impact Quality of Service (QoS). Instead, they determine what type of workload may or may not be processed by the application under a particular energy mode.

***Mode Case Definitions*** For (E2) runs, mode cases are defined to adapt to different energy modes. The mode case definitions for each benchmark are summarized in Columns 6-9 of Figure 7. Defining mode-specific behaviors in ENT — whether in response to a `BatteryException` in (E1), or defined as in a mode case in (E2) — is known to have impact on application quality of service (QoS) [26, 36, 40, 70].

In our experience, the development process of identifying and defining mode cases is as follows. We first identify application configuration parameters; modern applications are often shipped either with a configuration file or a class defining global constants, which may serve as a starting point. Second, we find the associated code for the parameters, especially the immediate class enclosing the parameter as a field. This often lead us to other parameters that could be adjusted. Third, we test run the program in its default configuration parameter setting, and two additional settings within the application specification requirement, but may potentially lead to higher/lower QoS. We aimed at realistic QoS settings for all our experiments. Last, the energy impact is measured, and those sensitive to energy consumption are candidates mode cases. In Section 6.3, we describe an analogous process in the context of energy debugging with ENT.

### 6.2 Energy Behaviors of ENT Programs

In this section, we report the experimental results of (E1), (E2), and (E3).

***System A*** The results of (E1) are presented in Figure 8. We enclose each **snapshot** expression with an exception handler to catch the `EnergyException` that makes adjustment to the aforementioned knobs in Figure 7. For each benchmark, we report the application execution under all 9 combinations of boot mode and workload mode.

As predicted, the mixed type system is able to find runtime errors in the 3 scenarios of the 9 executions: `managed` boot mode operating on `full_throttle` workload mode, `energy_saver` on `managed`, and `energy_saver` on `full_throttle`. We complement each case with a *silent* version of ENT constructed by modifying the compiler so that `EnergyException` is never thrown — even though the runtime tagging remains in place. Intuitively, the silent cases show "what could have been" if the runtime type system was not in place. We highlight these runs in Figure 9 under System A for percent energy saved. Observe that in all three cases where the exception is thrown, their counterparts lead to a significant increase in energy consumption. This shows in practice, respecting the waterfall invariant does lead to energy savings.

For (E2), we have created mode cases for each benchmark's boot mode that eliminate to the selected QoS values in Figure 7. The results of the battery-casing runs are presented in Figure 10 under System A. Indeed, `full_throttle` boot mode consistently consumes more energy than the `energy_saver` boot mode, showing that combining mode cases with a battery-aware attributor is another effective way to create adaptive code. Notably, the energy consumption results in the figure are often proportional to the amount of available energy, good news for energy-proportional computing.

For temperature-aware runs, (E3), we used a mode case for a `Sleep` object which regulates the sleep time needed to cool the CPU at a given mode. We chose sleep intervals to be 1000 ms, 250 ms, and 0 ms for modes `overheating`, `hot`, and `safe` respectively. We selected five benchmarks that have a distinct notion of "unit of work" — such as each XML file transformed in `xalan` — where the application can sleep in between. For all experiments, we chose a large data set for benchmarks to stress the CPU. We ran each benchmark twice, one with ENT implementation, and the other with the original Java implementation. Figure 11 shows the results. Most ENT runs hover around the `hot` threshold — `sunflow` being the exception that hovers near the `overheating` threshold — while the Java runs often lead the temperature to rise continuously throughout program execution.

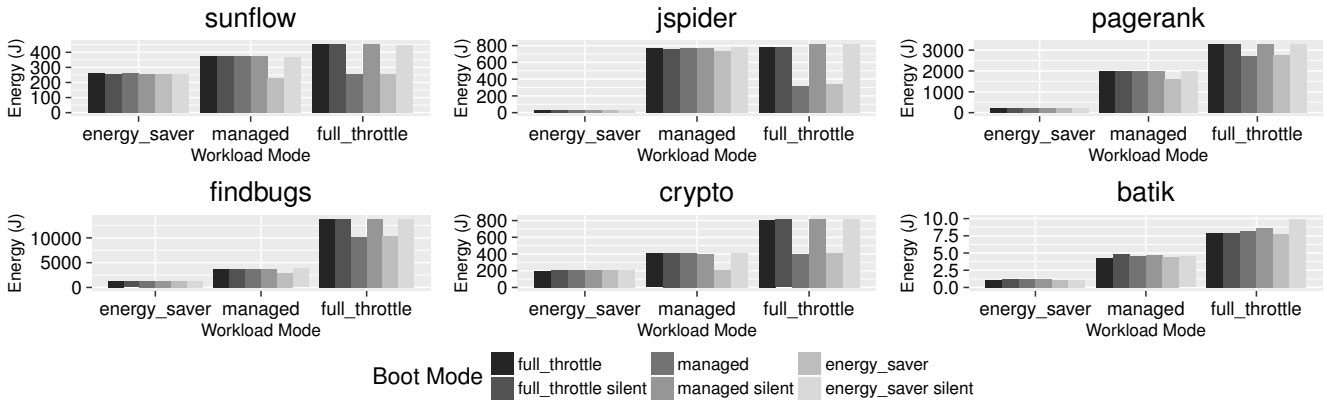


**Figure 8:** System A: Battery-Exception (E1) Runs: Each benchmark consists of three groups of results, each representing a workload mode, *i.e.*, `energy_saver`, `managed`, and `full_throttle` modes according to the descriptions in Figure 7. When waterfall invariant is violated an `EnergyException` is thrown and the quality of service is scaled down from default to `energy_saver` as defined in Figure 7. Within each group, three of them represent each boot mode, whereas their three counterparts represent the silent runs simulating the absence of Ent's type system by ignoring the `EnergyException`.

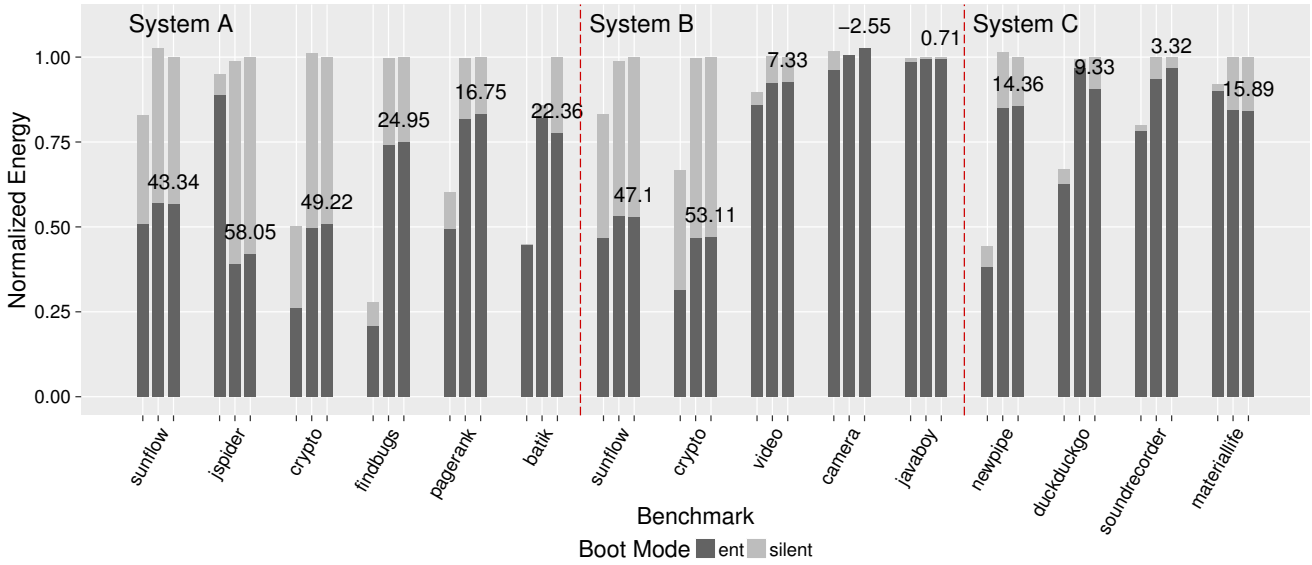


**Figure 9:** System A/B/C: Battery-Exception (E1) Runs (Normalized Energy Consumption over Boot-Workload Mode Combinations where `EnergyExceptions` are thrown): The three bars for each benchmark represent the `managed`/`full_throttle`, `energy_saver`/`managed`, and `energy_saver`/`full_throttle` boot mode/workload mode combinations respectively. Each bar consists of two overlapping (sub)-bars, with the darker representing the energy consumption incurred in ENT and the lighter representing the silent counterpart. Each (sub)-bar starts from the X-axis, *i.e.*, the two overlap instead of being stacked. The number on selective bars represents the percentage ratio between the two. For example, 43.34 in the first benchmark represents a 43.34% energy savings for the `energy_saver`/`full_throttle` run against its silent counterpart. All data are normalized against the silent `full_throttle` boot run.

***System B*** We evaluated ENT on the Pi by conducting (E1) and (E2) runs for 5 benchmarks. We adjust the input data size proportionally for `sunflow` and `crypto` to account for Pi's slower processor. We run each instance of `camera`, `video` and `javaboy` for 2 minutes; the minimum runtime where we observed acceptable relative standard deviation between the runs.

We highlight the runs where an `EnergyException` is thrown (E1) in Figure 9 under System B. Results for (E2) are presented in Figure 10 under System B. Overall, the results in System B follow a similar trend as in System A, so most earlier discussions apply here.

Compared with the Pi `sunflow` and `crypto` benchmarks, the Pi-specific benchmarks generally yield less percentage energy savings. For example for (E2) runs, the percentage energy savings of the `energy_saver` mode execution relative to the `full_throttle` mode execution are 6.38%, 19.63%, 1.34%, respectively, for `camera`, `video`,

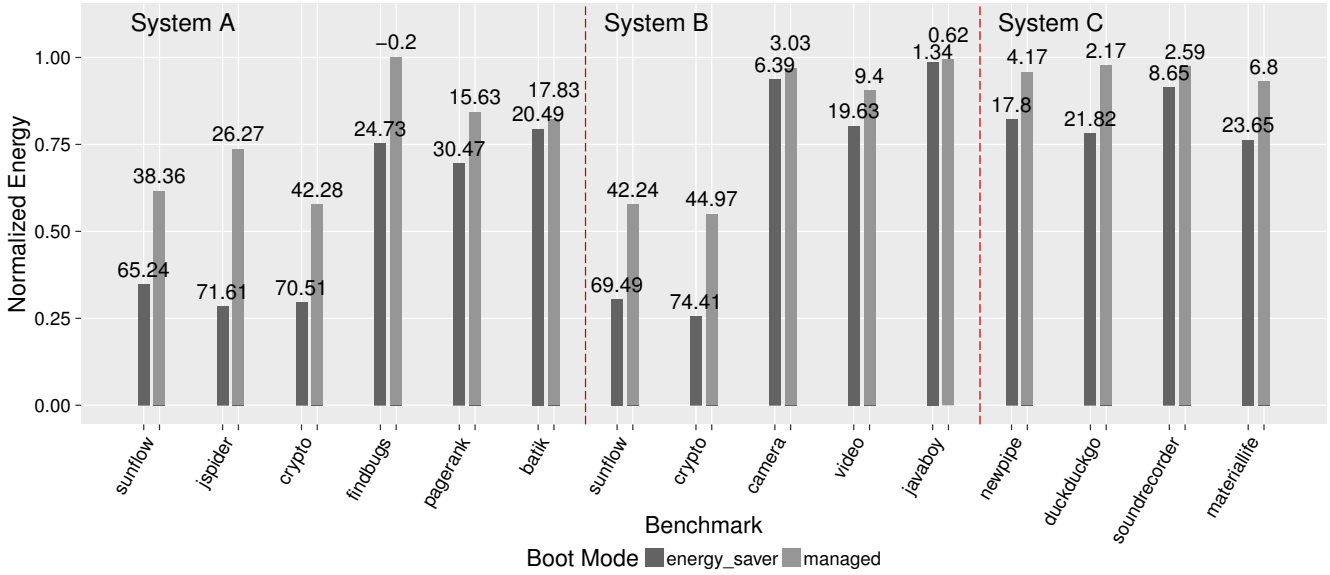


**Figure 10:** System A/B/C: Battery-Casing (E2) Runs (Normalized Energy Consumption): The two bars for each benchmark represent the `energy_saver` and `managed` boot mode normalized against the `full_throttle` boot mode using the large workload of each benchmark. The boot mode attributors are identical to those in Figure 8, and each boot mode selects the level of quality of service listed in Figure 7 through a mode case. Numbers indicate percentage energy saved against the `full_throttle` boot mode. For example, the first bar represents a 65.24% energy savings for sunflow when running under the `energy_saver` mode as opposed to the `full_throttle` mode.

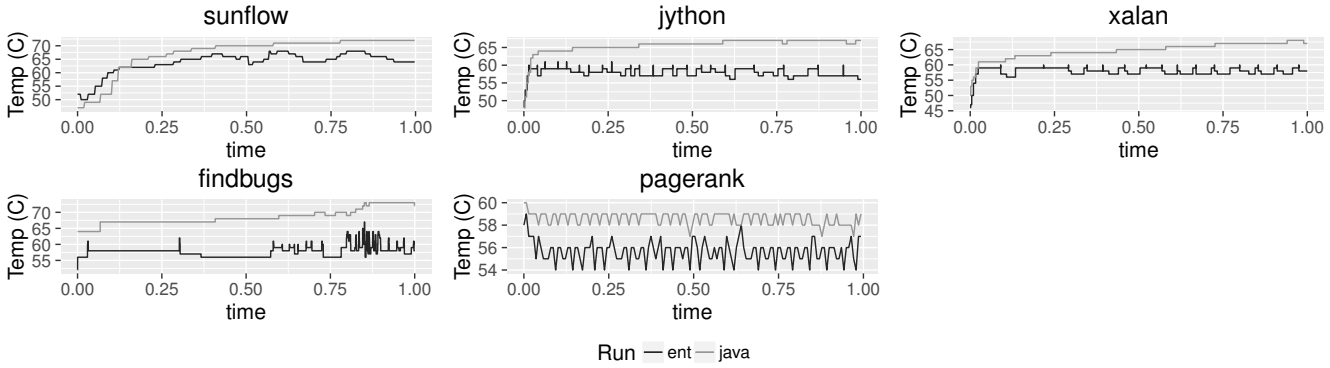


**Figure 11:** System A: Temperature-Casing (E3) Runs: Ent runs sleep for a selected period of time at the end of each task by snapshotting a dedicated `Sleep` object attributed to the current CPU temperature.

and `javaboy`. This may at first glance be disappointing. Observe however, all these 3 benchmarks are "time-fixed": when we compare results from different combinations of boot/workload mode, the *execution time* for all combinations is fixed. This is natural for Pi applications, as one *continuously* records video. Because of this, the difference of energy consumption in different runs must come from *power consumption* — the instant rate of energy consumption. In other words, when we run `video` in the `energy_saver` mode as opposed to the `full_throttle` mode, the Pi hardware system is running under a *lower-power* mode. This intriguing result shows the subtle interaction between application-level energy management and hardware-level energy management. It is intuitive: a lower video frame rate implies the lower intensity of encoding, and the Pi hardware components may have more opportunity to transition to a lower-power mode under the default governor (`ondemand`) of OS-level power management.

Finally, considering real-world use scenarios for Raspberry Pi often involve long-running and continuous executions (such as in habitat monitoring), a relatively small percentage in energy savings may yield large savings over time.

In all Pi-specific benchmarks, we did not observe *perceptible* degradation in quality when the application is run under the `energy_saver` or `managed` mode. This assessment may be subjective, but it may indicate the potential of application-level energy optimization for the Pi platform.

***System C*** We evaluated ENT on Android by conducting (E1) and (E2) runs for 4 benchmarks.Results are presented in Figure 9 and Figure 10. Like the Pi-specific benchmarks,

each Android benchmark runs for the same amount of time for each input size. This is consistent with the view that smartphone users rarely turn off their phone or exit Apps. Instead, most users treat their phone as a continuously running device; thus, the energy savings shown in the Figures represent a reduction in the power consumed continuously by the device, and show the potential savings ENT application programmers can achieve in the smartphone environment.

***Overhead*** Overhead introduced by our language design comes from runtime object tagging, and object copying. Note however, with the discussion in Section 5, both are considered in our compiler optimization. We measured overhead introduced by our mixed type system by creating a baseline as follows: it does not perform runtime tagging to objects, and it treats snapshot as a no-op. Figure 6 shows the percentage overhead of ENT compared with the baseline runs. The overhead is small. In several cases, the overhead is negative, indicating the variance in other factors (system behaviors, non-determinism in multi-threaded programs) often dominates. The only exception is `batik`, which has a large variation due to its low energy consumption ($< 10$ J).

### 6.3 Qualitative Characteristics

***Programmability*** In our case studies, we encountered a number of energy-aware programming patterns we find ENT is uniquely suited for, two of which we now summarize.

First, ENT allows programmers to express a number of adaptive energy behaviors challenging for statically-typed mode-based programming: (1) object modes are state-dependent; see discussion in Section 2; (2) object modes are configuration-dependent; see discussion in Section 2; (3) modes based on factors external to the application, such as lower-level system state (data rates in socket programming for example), battery level, and CPU temperatures. Such modes are particularly useful for defining boot modes; (4) when an object serves as a mediator or facade [37] between objects with different modes; (5) when an array may hold objects with discriminate modes.

Second, ENT instills principled energy management into mode-based programming. (1) all dynamic inspections on the run-time — be it on the application heap state or the underlying state — only happen in attributors. This in our experience leads to cleaner programs easier to understand and program. (2) generic modes can support expressive idioms — such as mode co-adaptation in Section 2 — without sacrificing correctness guarantees.

Our case studies also served as a reference for making decisions on designing semantic features. Two cases in point are whether changes between the snapshot and original object should be synchronized at the language level, and whether deep copy semantics should be used for snapshotting. Our current design left out both features, and this minimalistic design is driven by case studies.

In our case studies we did not encounter use scenarios where multiple snapshots are actively used and have overlapping lifetimes of mutating objects. Two common programming patterns are: (1) snapshotted objects whose mode changes due to system fluctuations such as battery, temperature, network states, buffer size, *etc*. For these objects, such as the `Agent` object in Listing 1, only the latest snapshot matters. (2) snapshotted objects whose mode is dependent on runtime information no longer change after a few initial steps of object construction. Examples are objects representing configuration settings, such as those passed in through `main` argument. In both cases state synchronization is not needed.

Our shallow copying design as opposed to deep copying was motivated the observation that even though object aggregates are common in our case studies, tightly coupled objects all with dynamic mode are uncommon. For example, the mode of a container object, *e.g.*, dependent on its size, often has no correlation with the objects contained inside. In such a scenario, deep copying at snapshotting time does not reflect the intuition behind dynamic mode characterization. An ENT programmer can mimic deep copying by following the Composite design pattern [37] where the container object and all its (potentially nested) components all implement a method, say `deep`, specifying what/how component objects need to be copied upon container object copying. This `deep` method should be invoked on the post-snapshot object.

***Debuggability*** ENT provides an "application-specific" view of energy bugs [56, 57] in the context of energy-aware programming: a violation of the principle of mode-based energy management. In ENT, bugs may be detected in two forms: (1) compile-time errors, when interactions between statically typed objects violate mode-based energy management; (2) run-time errors related to dynamically typed objects.

Typecheckers are often viewed as the first line of defense in debugging. Similarly, we believe the type system of ENT is conducive in debugging mode-based energy-aware programming. We return to Listing 1 to give an example. On Line 29, a programmer may have forgotten to place `[_, X]` on the **snapshot** expression. If so, a *compile-time* error will be thrown, indicating that the waterfall invariant needed at `s.crawl(depth)` is not satisfied. By adding `[_, X]`, the programmer acknowledges a `Site` could potentially be an *energy hotspot* with varying energy consumptions important for characterizing the energy behavior of `Agent`. Otherwise, she could have simply labeled the `Site` object as `energy_saver`.

Now suppose the programmer does not add the exception handler, and at runtime, an `EnergyException` is thrown. The programmer can focus on the following questions: (1) Why is a large `Site` crawled with low battery?, and (2) How to handle this error? In the case of (2), the programmer may `catch` the `EnergyException`, adjust the computation to one likely consuming less energy, and continue program execution. If the resulting program turns out not consuming much less energy — this feedback can be received if the programming environment is coupled with a jRAPL-like tool — it would be an indication that the `Site` object is not an energy hotspot after all.

In a nutshell, our type system aids the programmer in iteratively debugging and identifying the energy hotspots that lead to well-structured mode-based applications that exhibit energy behaviors consistent with our intuition.

***Ease of Use*** Because of the backward compatibility with Java, ENT programming does not require significant effort. Figure 6 shows the CLOC for each benchmark in Java, compared with the required ENT changes. For each benchmark, the CLOCs of ENT changes are negligible to the size of the code bases. In practice, we find transforming a Java application to a battery-aware or temperature-aware ENT program usually takes 1-2 hours. More time is devoted to discovering the energy hotspots following the debugging process described above, a process aided by ENT's mixed type system.

## 7. Related Work

A number of case studies [45, 58] have established the role of application-level energy management in energy optimization. Odyssey [54] is one of the earliest systems to highlight the importance and feasibility of application-level energy management. Green [19] is a QoS calibration framework with impact on energy consumption. PowerDial [40] is a control-theoretic framework for dynamically calibrating configuration parameters. PowerDial's dynamic knobs conceptionally represent potential quality of service hotspots that can be encoded in our system using mode cases, or adjusted when encountering `EnergyExceptions`. JouleGuard [39] is a cross-layer approach that provides energy guarantees by coordinating system-level energy efficiency and application-level accuracy. NRG-Loops [44] provides application specific adaptations through `for` loop constructs with respect to a RAPL monitored energy budget. These research efforts in general are framework-based, not language-based, but the overall goal of optimizing energy consumption at the application-level is shared by our work.

The history of mode-based energy management is perhaps as long as energy management itself, starting with CPU modes [31, 47]. Several programming languages exist for supporting mode-based energy management. In Eon [64], modes are energy states to determine data flows in sensor networks. LAB [46] allows programmers to prioritize among latency, accuracy, and battery, through a discrete set of mobile sensing algorithms. Eco [69] develops a supply/demand-based model for supporting sustainability, where mode cases were first used. While these frameworks provide mode-based abstractions, none of these systems are type-based. On the spectrum with proactiveness and adaptiveness on both ends, these efforts focus on adaptiveness.

Designing type systems for energy-efficient computing is an emerging direction. EnerJ [59] uses a type system to separate data that can be approximated from those that must be precise, with the goal of using approximation hardware to save power. EnerJ's approximate and precise annotations can be viewed as two modes. Like EnerJ, we support mode-based energy management at a finer-grained level. Rely [27]'s type system reasons about reliability of computations on unreliable but potentially more energy-efficient hardware. Chisel [52] allows accuracy and reliability to be specified as type signatures, where different signatures may have different energy impact. Uncertain<T> [23] provides a typed programming model for probabilistic reasoning under approximate computations. DECAF [24] uses type inference to manage the quality of overall program execution. There are type systems designed to reason about the cost of resource use in computations, with recent examples *e.g.*, [30, 41]. When tailored to energy consumption, these systems are complementary to our work. We regulate behavior in the presence of mode change; their systems can verify that such mode changes lead to quantitative difference in resource consumption. Closer to our work is Energy Types [32], where waterfall invariant first appeared. None of these systems except DECAF support dynamic typing. Together they highlight the importance of proactiveness in energy management.

Mixing static typing and dynamic typing in type system design is not new [17, 29, 34, 35, 50]. Gradual type systems [61–63, 65, 66] has recently been a particularly active area of study. Another related area is to support mixed type checking for dynamic languages, *e.g.*, [48, 67]. The type (*i.e.*, mode) of a value in ENT is determined at runtime. In contrast, the types of *values* in existing systems do not change during the lifetime of the program execution; what may change in their context is the concrete type of a program point on the data flow path based on runtime values flowing in. In that sense, there is a sink vs. source duality between existing work and our type system, leading to distinct challenges for both. DECAF has an optional dynamic tracking system for supporting probability reasoning constructed closer to the spirit of sink-oriented mixed type systems.

There is high-level analogy between our work and type-based information flow systems [53, 68]. For example, a recent effort HLIO [25] supports mixed type checking for information flow control in Haskell. The invariant enforced by these systems is non-interference. Our system on the other hand enforces waterfall invariant.

## 8. Conclusion

ENT is an energy-aware programming language to support proactive and adaptive mode-based energy management and promote the interaction between the programmer and the application runtime through mixed type checking. Our case studies show ENT is easy to use, and may improve the programmability, debuggability, and energy efficiency of battery-aware and temperature-aware software.

## Acknowledgments

We thank the anonymous reviewers for their useful suggestions, and Sasa Misailovic for his feedback. This work is supported by NSF CAREER Award CCF-1054515 and NSF CCF-1526205.

## References


[1] Advanced configuration and power interface, http://www.acpi.info.

[2] Duckduckgo, https://github.com/duckduckgo/android.

[3] Javaboy, http://www.millstone.demon.co.uk/download/javaboy/.

[4] jspider, http://j- spider.sourceforge.net.

[5] Materiallife, https://github.com/juankysoriano/MaterialLife.

[6] Mopi, https://pi.gate.ac.uk/pages/mopi.html.

[7] Newpipe, https://github.com/TeamNewPipe/NewPipe.

[8] Turn your pi into a low-cost hd serveillance cam, https://www.raspberrypi.org/blog/turn-your-pi-into-a-low-cost-hd-surveillance-cam/, .

[9] Satellite cameras saving endangered species, http://www.cambridgeconsultants.com/news/pr/release/140/en, .

[10] Emulation on raspberry pi 2, https://www.raspberrypi.org/blog/emulation-on-raspberry-pi-2/, .

[11] Raspberry pi, https://www.raspberrypi.org/, .

[12] Raspberry pi camera, https://www.raspberrypi.org/documentation/raspbian/applications/camera.md, .

[13] Raspberry pi camera java, https://blogs.msdn.microsoft.com/robert_mcmurray/2015/06/12/simple-java-wrapper-class-for-raspistill-on-the-raspberry-pi-2/, .

[14] Simbamon, https://github.com/hamishcunningham/pi-tronics/tree/master/simbamon.

[15] Soundrecorder, https://github.com/dkim0419/SoundRecorder.

[16] Watts up? power meters, https://www.wattsupmeters.com/secure/index.php.

[17] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *POPL '89*.

[18] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. PASTE '07, pages 1–8.

[19] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI'10*, pages 198–209.

[20] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190.

[21] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 595–602.

[22] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[23] J. Bornholt, T. Mytkowicz, and M. Kathryn. Uncertain¡t¿: a first-order type for uncertain data. In *ASPLOS '14*, pages 51–66.

[24] B. Boston, A. Sampson, D. Grossman, and L. Ceze. Probability type inference for flexibile approximate programming. In *OOPSLA '15*.

[25] P. Buiras, D. Vytiniotis, and A. Russo. Hlio: Mixing static and dynamic typing for information-flow control in haskell. In *ICFP 2015*.

[26] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ICSA '12*.

[27] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA '13*, pages 33–52.

[28] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system f with subtyping. In *Information and Computation*, pages 750–770. Springer-Verlag, 1991.

[29] R. Cartwright and M. Fagan. Soft typing. In *PLDI '91*.

[30] E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. Relational cost analysis. In *POPL '17*, pages 316–329.

[31] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low power cmos digital design. *IEEE JOURNAL OF SOLID STATE CIRCUITS*, 27:473–484, 1995.

[32] M. Cohen, H. S. Zhu, S. E. Emgin, and Y. D. Liu. Energy types. In *OOPSLA '12*.

[33] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *ISLPED '10*, pages 189–194.

[34] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP '02*, pages 48–59.

[35] C. Flanagan. Hybrid type checking. In *POPL '06*.

[36] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *SOSP '99*.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[38] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *ICSE '13*.

[39] H. Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 198–214.

[40] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS '11*, .

[41] J. Hoffmann, A. Das, and S.-C. Weng. Towards automatic resource bound analysis for ocaml. In *POPL '17*, pages 359–373, .

[42] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[43] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java - a minimal core calculus for java and gj. In *TOPLAS '99*, pages 132–146.

[44] M. Kambadur and M. Kim. Nrg-loops: Adjusting power from within applications. In *CGO '16*, .

[45] M. Kambadur and M. Kim. An experimental survey of energy management across the stack. In *OOPSLA '14*, pages 329–344, .

[46] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. The latency, accuracy, and battery (lab) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA '13*, pages 661–676.

[47] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 1st edition, 2008.

[48] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. TeJaS: Retrofitting type systems for JavaScript. In *Dynamic Languages Symposium (DLS) '13*.

[49] K. Liu, G. Pinto, and Y. D. Liu. Data-oriented characterization of application-level energy optimization. In *FASE '15*.

[50] Y. Long, Y. D. Liu, and H. Rajan. Intensional effect polymorphism. In *ECOOP '15*, pages 346–370.

[51] G. Mainland, G. Morrisett, and M. Welsh. Flask: staged functional programming for sensor networks. In *ICFP '08*.

[52] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *OOPSLA '14*, pages 309–328.

[53] A. C. Myers. Jflow: practical mostly-static information flow control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, page 228241, January 1999. URL `http://www.cs.cornell.edu/andru/papers/popl99/popl99.pdf`.

[54] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. pages 276–287, 1997.

[55] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *CC '03*, pages 138–152.

[56] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. HotNets-X '11, pages 5:1–5:6, .

[57] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys '12*, pages 267–280, .

[58] G. Pinto, F. Castor, and Y. D. Liu. Understanding energy behaviors of thread management constructs. In *OOPSLA '14*.

[59] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI'11*.

[60] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. Specjvm2008 performance characterization. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, 2009.

[61] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP '07*.

[62] J. G. Siek and W. Taha. Gradual typing for functional languages. *Scheme and Functional Programming Workshop*, 6: 81–92, 2006.

[63] J. G. Siek, M. Vitousek, M. Cimini, and B. John. Refined criteria for gradual typing. In *SNAPL '15*.

[64] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys '07*, pages 161–174.

[65] Takikawa, Feltey, Dean, Flatt, Findler, Tobin-Hochstadt, and Felleisen. Toward practical gradual typing. In *ECOOP '15*, .

[66] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *OOPSLA '12*, pages 793–810, .

[67] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *POPL '10*, pages 377–388.

[68] S. Zdancewic and A. C. Myers. Secure information flow and cps. In *10th European Symposium on Programming*, volume 2028, page 4661, 2001. URL `http://www.cs.cornell.edu/andru/papers/lincont.pdf`.

[69] H. S. Zhu, C. Lin, and Y. D. Liu. A programming model for sustainable software. In *ICSE'15*, pages 767–777, 2015.

[70] Y. Zhu and V. Reddi. Greenweb: Language extensions for qos-aware energy-efficient mobile web computing. In *PLDI '16*.