Eon: A Language and Runtime System for Perpetual Systems

Jacob Sorber Alexander Kostadinov Matthew Garber Matthew Brennan†
Mark D. Corner Emery D. Berger

Department of Computer Science
University of Massachusetts Amherst
Amherst, MA

†Viterbi School of Engineering
USC, Los Angeles, CA

{sorber, akostadi, mgarber, mcorner, emery}@cs.umass.edu, mdbrenna@usc.edu

Abstract

Embedded systems can operate perpetually without being connected to a power source by harvesting environmental energy from motion, the sun, wind, or heat differentials. However, programming these *perpetual systems* is challenging. In response to changing energy levels, programmers can adjust the execution frequency of energy-intensive tasks, or provide higher service levels when energy is plentiful and lower service levels when energy is scarce. However, it is often difficult for programmers to predict the energy consumption resulting from these adjustments. Worse, explicit energy management can tie a program to a particular hardware platform, limiting portability.

This paper introduces Eon, a programming language and runtime system designed to support the development of perpetual systems. To our knowledge, Eon is the first *energy-aware* programming language. Eon is a declarative coordination language that lets programmers compose programs from components written in C or nesC. Paths through the program ("flows") may be annotated with different *energy states*. Eon's *automatic energy management* then dynamically adapts these states to current and predicted energy levels. It chooses flows to execute and adjusts their rates of execution, maximizing the quality of service under available energy constraints.

We demonstrate the utility and portability of Eon by deploying two perpetual applications on widely different hardware platforms: a GPS-based location tracking sensor deployed on a threatened species of turtle and on automobiles, and a solar-powered camera sensor for remote, ad-hoc deployments. We also evaluate the simplicity and effectiveness of Eon with a user study, in which novice Eon programmers produced more efficient efficient energy-adaptive systems in substantially less time than experienced C programmers.

Categories and Subject Descriptors D.3.2 [*Software*]: Language Classifications—Specialized application languages

General Terms Languages, Design, Management, Performance

Keywords Coordination Languages, Energy Management, Energy Harvesting, Embedded Systems

1. Introduction

Sensor devices that rely exclusively on an electrical connection or on batteries suffer from numerous deployment disadvantages, from limited range and mobility to high maintenance costs and limited deployment scale and length. Sensor deployments that rely on the limited energy storage of a battery must either sacrifice data quality in order to maximize the lifetime of the network, or force maintainers to frequently change batteries, limiting the scale and coverage. In mobile deployments, such as wildlife tracking, the size of the battery may determine which deployments are even feasible.

Sensor devices can overcome these obstacles by harvesting energy from their environment. Available energy sources include solar, wind, and vibration energy [17, 27, 21]. Harvesting environmental energy enables the deployment of large-scale remote sensor systems that can run indefinitely: we call these *perpetual systems*. Notable examples of applications ideal for perpetual operation include wildlife tracking (ZebraNet [13]), volcanic eruption monitoring [31], and forest fire detection [20, 12].

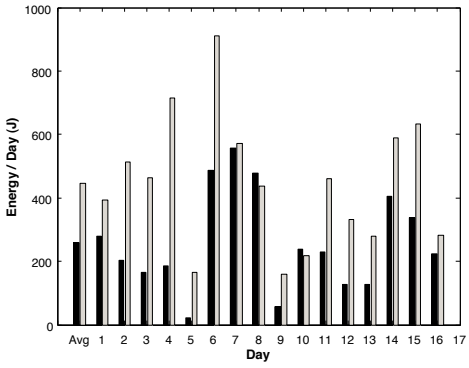
However, despite their deployment advantages, systems that employ harvested energy face numerous challenges.

SenSys'07, November 6–9, 2007, Sydney, Australia.

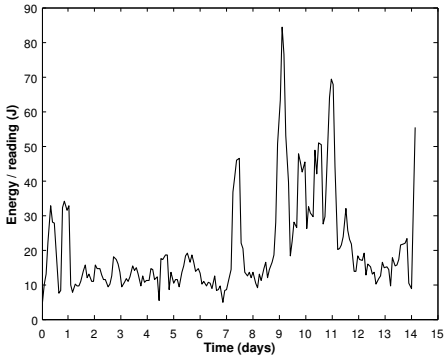
- *Dynamic energy availability.* Sensor devices that rely on environmental energy must cope with highly variable energy availability. The amount of available energy is often difficult to predict, and may change dramatically with location, time of day, time of year, weather, and other environmental factors. Figure 1(a) provides an example of these variations, in which the amount of energy gathered by two mobile, solar-powered devices over the same two week period is plotted on a per day basis. Although both devices show elements of the same general weather trend, the two devices show significant variation in the amount of gathered energy.
- *Varying energy costs.* The amount of energy required to perform tasks varies widely, making it difficult to plan for future energy needs. Figure 1(b) shows the amount of energy per reading that one device required to acquire GPS data over the same two week period as Figure 1(a). Because the device is mobile and the degree of cloud cover varies, the amount of time (and thus energy) required to synchronize with satellites varies over an order of magnitude. Comparing the graphs from Figures 1(a) and 1(b) shows that times of substantial cost do not necessarily correspond with times of plentiful energy; in this example, they are almost opposite.
- *Heterogeneous hardware platforms.* It is difficult to make a perpetual system portable or useful across a heterogeneous set of devices. Different hardware platforms have widely varying energy characteristics. Because of the complexities of testing and tuning a system to perform well across different processors, storage systems, radios, energy harvesting sources, and batteries, it is extremely difficult to write a perpetual system that will work across different platforms.

When programming systems to cope with these shifting conditions and platforms, designers are forced to incorporate adaptation with the core logic of the system. Such programs are difficult to port, maintain, and understand. Further, there is a great deal of runtime functionality that must be replicated each time the system is deployed on a a new platform with new energy characteristics.

Contributions: This paper presents **Eon**, a new language and runtime system designed for programming perpetual computing systems. To our knowledge, Eon is the first *energy-aware* programming language. Eon is a declarative coordination language based on Flux [3] that allows programmers build programs from code written in a variety of languages, including nesC and C. Eon provides a simple way to associate particular control flows with abstract *energy states* that represent the available energy in the system. The Eon runtime system executes only those flows that the Eon programmer has marked as suitable for the given energy state. Thus, an Eon programmer can easily write programs that provide different functionality or data quality based on current and future energy availability.



(a) A histogram of the average amount of daily energy gathered by two devices.



(b) The amount of energy needed to take a GPS reading over the same period.

Figure 1. Energy traces from solar-powered GPS devices over a two week period.

This flow and energy state information enables *automatic energy management*, allowing the runtime system to handle the complexities of adaptation. In response to changes in energy, the Eon runtime system dynamically adjusts the execution rate of flows and enables or disables application features. Because Eon programs describe energy abstractly (e.g., "high" and "low"), they are portable to hardware platforms with arbitrary energy profiles. The language itself is also highly portable: the current Eon compiler generates code for a variety of embedded platforms and operating systems, including Linux and TinyOS.

To demonstrate Eon's utility and portability, we have built and deployed several Eon-based perpetual systems, including two solar-powered systems: one for tracking turtles and automobiles using GPS and another for capturing and transmitting images from remote locations. To quantify the ease of programming perpetual systems in Eon, we conducted a user study showing that programmers who had just learned Eon outperformed a control group using C, taking less time to produce equally efficient code.

Outline: The remainder of this paper is organized as follows. First, Section 2 describes the Eon language, focusing on the description of flows and energy states. Next, Section 3

describes Eon's automatic energy management algorithms. Section 4 describes implementation details of the hardware and software systems, including the compiler, runtime system, and the trace-based simulator that the compiler can generate to predict performance before deployment. Section 5 describes three Eon-based perpetual systems we have built. Section 6 presents empirical results both for our user study and for one of the perpetual systems deployments. Finally, Section 7 discusses the most closely-related work, and Section 8 concludes with directions for future work.

2. The Eon Programming Language

Eon is a domain-specific language intended to support a broad range of perpetual systems. These include energy-limited systems that follow an event-response model of operation, such as devices that respond to external stimuli or to periodic, internally created interrupts. Eon combines both simplicity and elegance: its goals are to make energy-adaptive systems simple to write and easy to understand and to enable the use of optimized energy-aware runtime systems that automatically choose the highest sustainable service level.

The Eon programmer writes code that describes the sequence of operations that follow in response to external events and the desired adaptation policy, i.e., which sequences (flows) correspond to higher or lower power energy states. The Eon runtime system measures the probable costs of each operation, the probable workload in the system, and the probable amount of energy the system will acquire. The runtime system then adjusts the rate of execution of flows that the programmer has indicated are appropriate for a given energy state.

It would be possible to build Eon's energy-aware features into either an entirely new general-purpose programming language or as extensions to an existing language. The first approach would require programmers to learn a new language while muddling basic constructs such as loops and conditionals with policy. This approach would also preclude the reuse of the vast amount of code already written in general purpose languages. While using annotations would simplify adoption for new programmers, the annotation syntax would have to be adapted to each new language. The resulting system would still muddle the issues of adaptation with logic. Most importantly, conventional programming languages do not explicitly manage program flows: these are implicit in program execution, and thus difficult to annotate.

Instead, Eon is a *coordination language* [9] that ties together code written in a conventional programming language, like Java, C, or nesC [8]. This approach provides programmers with a high level of abstraction that separates the concerns of energy adaptation from program logic. It also makes it straightforward to reuse existing code. Eon currently supports a range of different languages (C/nesC) and operating systems (Linux/TinyOS).

This approach also makes it simple to port an Eon program to a new platform. For example, porting an Eon program from an XScale-based device to a mote-class device required only modification of the platform-specific code used to implement the program logic. This portability makes Eon a natural candidate for use in embedded devices, given the wide variety of platforms, operating systems, and languages currently in use.

```
1  // Predicate Types
2  // SYNTAX: typedef PRED_TYPE PRED_TEST
3  typedef gotfix TestGotFix;
4
5  // Source Node Declaration
6  // SYNTAX: NODENAME () => (OUTPUTS);
7  ListenBeacon() => (msg_t msg);
8  GPSTimer() => ();
9
10 // Concrete Node Declaration
11 // SYNTAX: NODEAME (INPUTS) => (OUTPUTS);
12 GetGPS() =>
13    (GpsData_t data, bool valid);
14 LogGPSData(GpsData_t data bool valid)
15         => ();
16 LogGPSTimeout(GpsData_t data bool valid)
17         => ();
18 LogConnectionEvent(msg_t msg) => ();
19
20 // Regular Sources
21 // SYNTAX: source NODENAME => NODENAME;
22 source ListenBeacon => HandleBeacon;
23
24 // Timer Sources
25 // SYNTAX: source timer NODENAME
26         => NODENAME;
27 // Eon Timer Source
28 source timer GPSTimer => GPSFlow;
29
30 // Eon States
31 // there is always an implicit BASE state
32 stateorder {HiPower};
33
34 // Abstract Nodes and Predicate Flows
35 // SYNTAX: ABSTRACT[[type,..][state]] =
36 // CONCRETE->...CONCRETE;
37 GPSFlow = GetGPS -> StoreGPSData;
38 StoreGPSData:[*,gotfix][*] = LogGPSData;
39 StoreGPSData:[*,*][*] = LogGPSTimeout;
40
41 // Abstract Node using Energy Predicates
42 HandleBeacon:[*,*][HiPower]
43         = LogConnectionEvent;
44
45 // Eon Adjustable Timer
46 GPSTimer:[HiPower] = (1 hr,  10 hr);
47 GPSTimer:[*] = 10 hr;
```

Figure 2. A condensed version of Eon source code for the turtle tracking application.

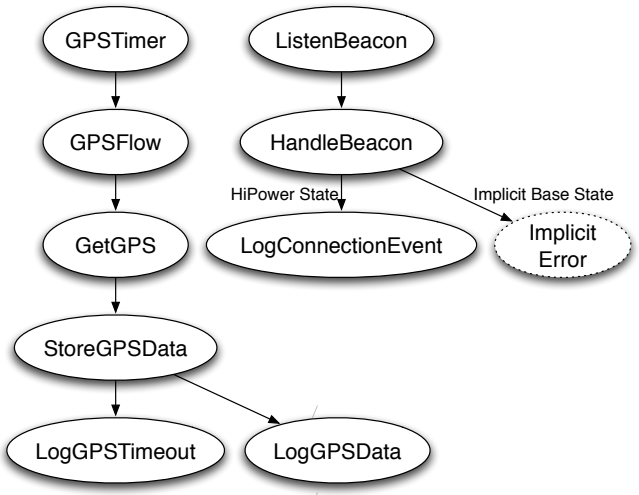


Figure 3. A graph of a simplified turtle tracking application

2.1 Basic Eon Syntax

A coordination language describes the flow of data through different components. We have built Eon on top of an existing coordination language called Flux [3], due to its features, simplicity and available compiler tools. Flux is a declarative language that describes a directed acyclic graph embodying the flow of data through the program. Flux *sources* connect to abstract nodes, which consist of a series of concrete nodes. Concrete nodes correspond to implementations written in a conventional programming language. Flux also allows for conditional flow through a program—a feature that Eon leverages for energy adaptation.

We illustrate Eon's syntax using examples from Figure 2 and the graphical representation of the program in Figure 3. We first describe the parts of the program that are the same as in Flux, and then describe Eon's extensions.

Flux-based syntax: As in Flux, an Eon programmer first declares each *source node* in the program and what types of data it outputs, such as `ListenBeacon` on Line 7, which produces an output of type `msg_t`.

Source nodes feed data into other *concrete nodes*, which correspond to functions implemented in conventional programming languages like C and nesC. Each concrete node takes a set of input arguments and produces an output set of arguments. For instance, *GetGPS* (declared on Line 12) takes no input and produces two output variables: a GpsData_t and a boolean. The Eon compiler checks to ensure that output and input types match in each flow.

Abstract nodes describe the flow of control and data through multiple concrete or other abstract nodes. For instance, GPSFlow (defined on Line 37) is an abstract node that is the combination of two other concrete nodes.

Conditional flows are implemented in Eon using *predicate types*: programmer-defined boolean functions that are applied to a node's output. In Figure 2, the *StoreGPSData* abstract node specifies two possible execution paths on Lines 38 and 39. By applying the *gotfix* predicate to the output of *StoreGPSData*, the Eon program decides which path to take. The test is defined on Line 3.

Each of the concrete nodes and all predicate tests must be implemented by the programmer in a supported conventional programming language (currently C or nesC). The Eon compiler generates a set of stub functions for each node that must be implemented by the programmer.

2.2 Eon Extensions

While the parts of Eon drawn from Flux lets programmers define the sequence of operations that follow from events, they lack any method to express runtime adaptations. In this section, we describe how Eon extends Flux with constructs that describe what runtime adjustments to make as well as the priority with which they should be applied. The Eon application is then mapped to an adaptive runtime system, which continually adjusts the application in order to balance the demands of fidelity and sustainability. We continue to use the application shown in Figure 2 as an example.

Power states

Adaptation policies could be expressed as a set of *utility* functions describing the relative value of flows, and the rate of flows in an Eon program [4, 18]. Our own experience in building adaptive applications as well as anecdotal evidence suggest that general utility functions are difficult for programmers to use or understand.

In contrast to previous approaches, we have found that a simple partial ordering of flows and rates is sufficiently expressive. While a utility function can express a greater number of policies, such as non-monotonic values, and are amenable to a great number of interesting analytical results, their usefulness is questionable while severely complicating life for the programmer.

In an Eon program, a programmer specifies an adaptation policy as a collection of behavior adjustments organized in a *state ordering*. An adjustment is declared simply by listing it in the state ordering, and its priority corresponds to the row in which it appears. All adjustments on a given row are applied together.

Figure 4 shows how the sample application's operating states are derived from the state ordering. An implicit BASE state (S_0) represents the program running without applying any adjustments. Subsequent states are defined recursively by applying an additional level of adjustments to the previous state (i.e. $S_i = S_{i-1} + L_{i-1}$). Also, a higher operating state is assumed to be more desirable and more energy-intensive than all lower states.

The state ordering of an Eon program defines which operating states can be chosen by the runtime system. In addition to declaring adjustments, the system designer must also define what those adjustments are.

Adaptive Timers

One of the most common adjustments used to reduce energy consumption is to periodically turn off energy-hungry com-

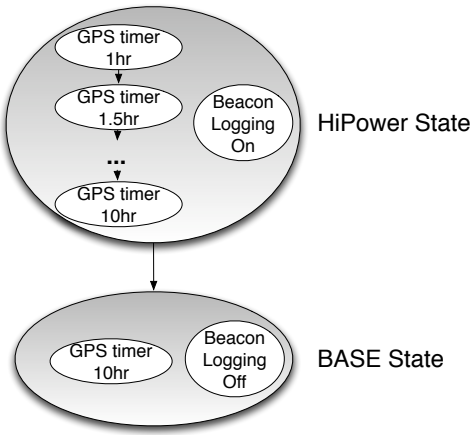


Figure 4. Sample State Order.

ponents, such as radios [1, 26]. In the turtle tracking application, the GPS receiver consumes two orders of magnitude more power than all other components combined. This cost makes the frequency of GPS readings the most important factor in the life of the device. Adaptively adjusting the duty cycle of a component or task represents a trade-off between application fidelity and energy consumption.

Duty-cycle adaptation is implemented in Eon using a special type of event source node called an *adaptive timer*. Adaptive timers differ from other sources in that they are not concrete nodes and are not implemented by the programmer. Instead, the programmer specifies a range of acceptable timer intervals. For example, the *GPSTimer* in the turtle application can fire anywhere from every hour to once every 12 hours. The interval is then set by the runtime system.

Energy-State Based Paths

Another common way to trade value for energy is to change the fidelity of data and the availability of services. Lowering the quality of images, audio, or video reduces the energy a device spends transmitting. Energy can be conserved further by making some services unavailable. For example, a remote camera may store images locally for later querying or only stream metadata, instead of streaming the full images. [16].

Fidelity and availability adaptation is provided in Eon using *energy-state based paths*. This concept is akin to the predicate types used for conditional flows except that instead of choosing paths based on output types, paths are chosen based on the energy state set by the runtime system. In the case of our turtle application, *LogConnectionEvent* is called when *HandleBeacon* produces any type and is in a state labeled *HiPower*. If the node is low on energy, it may enter the implicit BASE state and cease logging beacons from other nodes to save energy. *HandleBeacon* does not take inputs of the BASE state type, so the flow ends in an implicit error that has no side-effects. In this example, Eon lets the programmer express preference for local operations over providing services to other nodes when energy is low.

Implementing Concrete Nodes

Implementing concrete nodes with nodes that block on I/O is straightforward, such as `read()` in a C/Linux system: the programmer merely adds procedures that run until the I/O is finished and then return. If concurrency is a concern, Eon can use Flux's features for the automatic generation of multi-threaded code [3].

Implementing Eon nodes for the nesC/TinyOS environment is less straightforward due to its use of split-phase, event-based semantics. Instead of a single blocking function, a TinyOS concrete node is implemented as a simple nesC component that provides a single "call" command and an asynchronous "done" event that is signaled with the node's return values upon completion. This allows simple nodes that consist of a single function as well as more complex nodes that perform split-phase TinyOS operations.

Discussion

One feature that we considered but rejected during the development of Eon was to implement fine-tuned adjustments in node fidelity. For instance, like timers, we could have provided an explicit adjustment in the fidelity of a node that performs an operation such as video encoding. The runtime system would then have been able to adjust this knob to adapt the fidelity of video encoding in a large number of steps.

However, our experience with adaptive systems has been that only gross levels of adjustment are used. Video is either high-fidelity, low-fidelity, or perhaps a level in between. While Eon's timers are finely adjustable, the semantics of timers and their resulting energy cost are both simple to predict and effectively linear. For instance, firing a timer twice as often will use approximately twice as much energy per unit time.

The energy consumed by a video codec would likely have a non-linear relationship to its resolution. Tuning the fidelity would thus have a corresponding non-linear effect on nodes downstream that transmit the video. Recall that one of our goals is to provide a language that is conducive to well-performing runtime systems. Without an accurate prediction as to what effect an adaptation will have, it is more difficult to select the correct operating point. To find such non-linear, and often noisy, relationships takes a great number of sample points, each of which may be consuming too much or too little energy while the system runs.

Further, there are an unlimited number of power management optimizations that can be made in sensor systems, from wireless duty-cycling, to link-layer power-control, and CPU frequency scaling. Our standpoint has been that anything that can be automatically inferred from the program itself in a general and reasonably efficient manner, should be. Along these lines, we have considered a great number of features to add to the language, but have generally favored simplicity over features instead of building a language that can express every possible energy-management scheme. For instance, instead of providing timers that synchronize to a common time reference for a Synchronized MAC (S-MAC)

duty-cycling [32], we use the low-power listen mode present in many modern sensor radios.

3. The Eon Runtime System

By using the flow descriptions in the program, on-line measurements of the per-task energy costs and workload, and predictions about the amount of incoming energy, Eon's runtime system adapts program execution according to the program's policies. This adaptation is completely automatic, and requires minimal online measurements.

3.1 Design Goals

Two goals inform the design of the Eon runtime system. First, it should support a broad array of low-power platforms, such as Motes [25] and Stargates [30], powered by solar energy. Because microcontroller platforms have relatively small memory sizes, the runtime system must be constrained to perform few measurements on-line.

Second, the runtime system should not require any explicit training, such as measuring the system under simulated load in a lab. Not only is this process painful for programmers, it is also inherently brittle. For example, training might require repeated measurement every time the program is changed or deployed on a new platform with new peripherals and is dependent on having good models of the expected workload. As long as in-situ measurement is sufficiently accurate, and can be done with low-overhead, online measurement is greatly preferable.

3.2 Energy Adaptation Algorithm

The runtime system executes an adaptation algorithm that chooses the ideal power state for the system to use, based on its measurements of energy consumption and production. The adaptation algorithm strives to provide the highest fidelity to the application while avoiding two states: an empty battery and a full battery.

An empty battery prevents the application from executing even high priority flows. In many devices, it also imposes a period of dead time for the system, during which the battery must slowly charge up to a minimal level before the device can turn on again. When the battery is full, any additional environmental energy that the system harvests is wasted and cannot be stored for later use.

From Eon's perspective, any state of the battery between these two states is effectively equivalent: the goal of the system is to consume energy at a rate equal to the rate of energy production. The battery's role is to act as a buffer, riding out periods of low energy production and storing excess energy.

The runtime system periodically makes a decision about the ideal power state for the system by searching the possible adaptations, such as timer frequencies and power states. Eon favors smoothness of adaptation and searches for a single static policy that is sustainable for a long horizon.

Eon can make large adjustments using the energy-state based paths, and smaller adjustments using the adjustable timers. Eon initially assumes that the system runs at the highest energy state with the minimum frequency for all of the timers. It then computes the amount of consumed and produced energy over a short interval T_i. Taking into account the current state of the battery, if this power state would empty the battery, the system lowers the energy state (for instance, Hi-Power to Lo-Power), and then repeats. Once it finds a state that is sustainable over the short interval T_i, it looks further into the future to see if the rate is truly sustainable, examining time horizons $2^n \cdot T_i$ for $n = \{1..N\}$ time intervals.

Once the system finds a sustainable energy state, it performs a binary search on the timers using the same time horizons to discover the exact sustainable policy. This search strategy ensures that the policy is sustainable both over the short and long term, without requiring excessive compute time. More weight is given to the short term, as the runtime system periodically reexamines the policy to adapt to changing workloads and energy dynamics. The entire process runs in just 100 ms on a Mica2 mote for our full tracking program with 31 flows and a horizon of half a year.

Energy Attribution and Consumption

For adaptation, the system must have an accurate model of its energy consumption, including the energy cost and frequency of each independent execution path, or flow, through the program. Each time an Eon flow completes, the runtime system updates an exponentially weighted moving averages (EWMA) of the flow's energy cost. The system also estimates the originating source's firing frequency and the probability of each branch taken by the flow. In the example in Figure 2, there are four possible paths through the program, each with a different energy cost and frequency.

Measuring per-path energy consumption requires careful accounting and hardware support. One option is to use a Fuel-Gauge IC, like those included in many modern laptop, mobile phone, and PDA batteries; two popular example include TI's bq27000 and Maxim's DS2770. These chips measure the capacity of the battery and charge/discharge rates, including corrections for temperature, battery-chemistry, and aging effects. A fuel-gauge chip provides an averaged, coarse-grained view of the energy remaining in the battery and the current rate of charge or discharge. While necessary, this information not sufficient to distinguish between energy consumption and charge, as both occur simultaneously.

The Eon runtime system requires both a fuel-gauge chip and fine-grain current measurement to attribute energy to individual program flows. In our hardware platform, we use an integrated current sensor, which separately measures the rate of consumption. This hardware is accurate to within 0.6mA, sensitive enough to measure differences in current consumption due to radio, flash, or peripheral use by individual flows on a variety of platforms.

The runtime system samples the current once every second, while simultaneously tracking the start and end times of each node in the program graph. Based on the percent-

age of time that nodes from a particular flow were running, the runtime system attributes energy to the flow. The rest of the energy is attributed to the runtime system and to the idle energy consumption of the platform.

Given the amount of energy consumed by the program and runtime system, Eon estimates the energy production rate. Adding the energy consumption for a period of time to the loss or gain in battery capacity yields the energy production over that same period.

Energy Source Model

In addition to knowing how much energy each path consumes, adaptation requires a model of how much energy the system is going to receive in the future. While Eon is not tied to any one energy production method, we concentrate on solar power, which is particularly challenging. The amount of available solar energy is highly variable. It is also unpredictable, since predicting sun intensity is, in essence, predicting the weather.

The model we use in our prototype is an adapted exponential weighted moving average (EWMA) based prediction algorithm from Kansal, et al. [14, 15]. This model essentially predicts that the energy production in the following days will be similar to recent days. Eon measures the energy production over a day, and assigns this value as $E(t)$. It then computes the expected value of $E(t+1)$ as $\alpha E(t) + (1-\alpha)E(t-1)$. This model masks the diurnal cycles inherent to solar energy harvesting and is simple enough for use in small embedded devices.

4. Implementation and Deployment

This section describes the details of the Eon software and its hardware support. In addition, it describes the details of three Eon deployments: a turtle tracker, an automobile tracker, and a remote imaging system. The designs for the hardware, as well as a release of the application code, compiler, and runtime system, are all available from our website (`http://prisms.cs.umass.edu/~sorber/eon`).

4.1 Software

The software implementation of Eon includes a compiler and runtime system, as well as a generator for a trace-based simulator.

Compiler

The Eon compiler is a three-pass compiler implemented in Java, using the JLex Lexer and the CUP LALR parser generator. It is based on the original Flux compiler [3], extended with support for Eon's energy management features. The first two stages of the compiler build a graph representation of the program and then decorate each edge with input and output types. The third stage links this intermediate code with the Eon adaptive runtime system and user-supplied code that contains the program logic.

Eon can be ported to new languages and architectures with minimal effort. Our current implementation has been ported to two different environments: an Intel/CrossBow Stargate [30] XScale Linux system, using nodes written in C, and an Atmel microcontroller-based TinyOS system using nodes written in nesC [8]. In addition, we have ported Eon to a number of hardware platforms, including the Mica2Dot, Mica2, MicaZ motes [25], the TelosB mote [24], and the Shockfish Tinynode [5].

Runtime System

The Eon runtime system measures and adapts to energy usage and production. At the start and end of every flow, the code generated by the compiler invokes a set of functions that interface with the hardware, perform predictions, and calculate a running state. The result then informs the rest of the runtime system which state the system will operate in. The size of the TinyOS runtime is 4850 lines of code, occupying 18 kbytes of program ROM. While running, the runtime system uses 900 bytes of RAM for an empty program, plus approximately 30 bytes of RAM for each independent path in the program, depending on the size of the arguments passed between nodes.

Trace-Based Simulator

The Eon compiler optionally generates a trace-based simulator. By feeding an energy trace and traces for external inputs, an Eon programmer can test different energy predictors, workloads, programs, and adaptation policies. During deployment, an Eon node collects measurements of solar energy, consumed energy, battery state, estimated idle power draw, estimated per-path energy costs, path probabilities, and source frequencies. All of this information is then used as input to the simulator. Additionally, we have found that the information recorded by the runtime system is extremely useful as an energy profiling tool. Although not as accurate as an external measurement tool, it has been crucial in identifying energy bottlenecks in our systems.

4.2 Hardware

Eon's adaptation algorithms require hardware support. We have built a new charging and energy management board that controls the solar charging of lithium ion batteries, measures the capacity of the battery with a Maxim DS2770, and measures the current consumption using a Maxim DS2751. We have fabricated two versions of the board, one that accepts a Mica2DOT mote as a drop in module, and one that attaches to a Shockfish Tinynode via a Molex connector. We adapted some parts of the hardware design from the Heliomote project [17]. The same board can be used with the Stargate, by attaching the board via a mote.

Figure 5 shows the deployment platforms for the Mica2DOT and TinyNode, shown with battery and GPS. This board can handle a wide variety of solar cells, ranging from a small, 25mA peak current cell up to a cell producing 2A. Additionally, Eon requires no program or runtime changes when changing the size or number of solar cells, since it only tracks the amount of energy production.

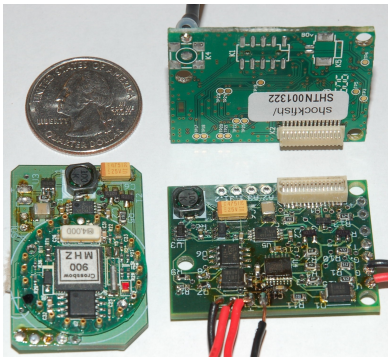


Figure 5. The two implementations of the energy measurement and charging board with a Mica2Dot and a TinyNode.

(a) Photo of an Eon node on a Turtle.

(b) Camera

Figure 6. Photos of two of the test applications, a turtle tracking device, and a remote camera.

5. Deployment

In order to evaluate Eon, we have built several energy adaptive systems: a turtle tracking node, an automobile tracking node, and a remote imaging system. The evaluation section focuses on the automobile tracking system, and we describe all three systems here. We have also constructed a solar-powered WiFi web server on the Stargate platform.

While these deployments are somewhat limited in their scale and duration, we have gathered sufficient data to demonstrate Eon's utility in performing energy adaptation. Perhaps more importantly, these deployments have driven the development of Eon, rather than following as a consequence of it. The applications have informed which features to add to the language, runtime system, and hardware support.

5.1 Turtle Tracking

The first deployment is motivated by the efforts of conservation biologists to protect threatened turtles. The Wood Turtle (*Clemmys insculpta*), is found throughout the Northeast and Great Lakes regions and into Canada. They live primarily in and along streams, and are terrestrial for about 4 months of the year. Wood Turtles are of particular interest since their numbers are rapidly declining. Unfortunately, conservation efforts have been hindered by a general lack of data due to current tracking methods. Researchers currently track turtles manually using radio telemetry and are limited to taking a single location fix every 2-3 days for each animal being studied. The turtles often travel up to 1 kilometer between fixes and practical concerns preclude the collection of location information at night.

Much of the development of Eon is inspired by this particular problem. We have designed and built an Eon node and program to run on the Mica2DOT environment. The turtle node includes a SiRF Star III-based GPS Receiver, an Ultralife UBC581730 250 mAh battery, and one or two 4.2V PowerFilm flexible solar cells. The node is packaged in shrink-wrap tubing and the ends are sealed with a waterproof epoxy. The design of the node is primarily driven by form-factor. The node must weigh less than 50 grams and fit without protruding from the shell. Figure 6(a) shows the Eon node mounted on a turtle's shell.

Unfortunately, our deployment took place at the end of an unusually cool fall. The turtles prepared for hibernation early and spent a large amount of their time immobile and underwater, not emerging until the next summer. We thus collected relatively little data from the turtles: five days of solar traces and a handful of GPS locations.

Despite this small amount of data, we learned new facts about turtle behavior that were useful from a zoological perspective and that have led to improvements in our system. In particular, we discovered that the turtles were underwater 98.5% of the time. Because GPS does not work underwater, we added a water sensor to the node that lets the programmer specify that no GPS readings should take place if the turtle is underwater. In addition, we found that the turtles receive a great deal less energy while underwater, so little that even our upper-bound for the GPS timer was not sufficient to let the node survive. The combination of these two fixes should allow the node to survive long periods of time underwater.

5.2 Automobile Tracking

As a proxy for the turtles, we performed a second deployment using automobiles. We used the same hardware, adaptation policy, program, and runtime system, and collected two weeks of data from five devices mounted on the roofs of cars. The weather for that two weeks was highly variable, with several days of consecutive cloudy weather. These traces can be extended by looping them, which gives us a good idea of how Eon adapts to changing conditions. In addition, this automobile-based deployment has led to bug fixes and other improvements to the runtime system.

While we plan to redeploy the turtle nodes in a large-scale experiment in the summer, the evaluation we present here is based on data gathered from the automobile-based experiment. The complete application, excluding Eon runtime code, is 7900 lines of code. The complete system, including the Eon runtime, compiles to 42 Kbytes of program memory, and runs in 3600 bytes of RAM.

5.3 Remote Camera

Finally, we have built a remote camera application that demonstrates Eon's versatility. This application was inspired by various remote image applications at James Reserve [20], in SensEye [16], and at Virginia Coast Reserve (VCR) [6]. Of note is the fact that VCR researchers programmed their cameras to scale their frame rates to cope with fluctuations of gathered solar power.

To ease their programming burden, we have constructed the video system using a TinyNode [5], a CMUCam low-power camera [28], a 400-mW-peak solar panels, and a 1 Ah battery, shown in Figure 6(b). The Eon application trades off the competing concerns of the frequency of image capture, and image streaming(high power), and image storage (lower power). Using the TinyNode's XE1205 radio, the images can be streamed from the solar-powered node to a base-station up to 1 kilometer away.

Once the CMUCam was connected, building a fully adaptive application took a single developer only three hours to build. No modifications were required to handle the larger solar cells or the energy requirements for the new platform and camera.

6. Evaluation

Our primary goal in Eon was to provide a simple language for building efficient energy-adaptive embedded systems. In this section, we evaluate the Eon language with respect to both usability and system performance.

6.1 User Study

To evaluate Eon's usability, we conducted a user study. Nine programmers were recruited for the study, the majority from a junior-level operating systems course and all having at least 4 years of prior programming experience. None had any prior knowledge of Eon and all were familiar with C. Each subject was initially asked to provide a self evaluation of past experience and programming expertise, which was used to divide them into two balanced groups, one using Eon and the other using C.

Each test subject then completed the user study individually. Participants were first given a 45-minute long tutorial covering the programming tools and computing environment, and an overview of energy-aware embedded systems. Following the tutorial, each subject was asked to write two applications.

The first application was a simple sensor application which periodically samples a sensor, stores the collected sensor readings, and answers simple network requests for past readings. The second application was an extension of the first application to make it adaptive, with the goal of providing the best sampling rate that their device could sustain without running out of energy. After completing these programming assignments, each participant was asked to take a post-experiment survey qualitiatively evaluating their experience.

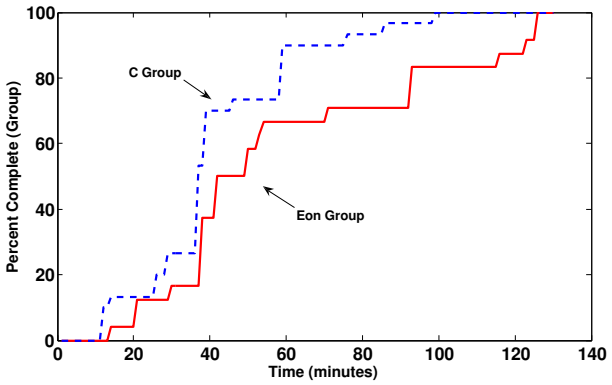
The students performed the study in a simulated environment that included APIs for measuring energy spent using the Flash device, the radio, and for taking sensor readings. Also, in order to provide a fair comparison, we provided the C group with the same solar energy predictor used by the Eon runtime system. The build environment was instrumented in order to collect a snapshot of each participant's code with every successful compile. After the study was complete, we tested each submission. The initial program was tested for correctness to verify that it performed readings and answered queries correctly. The adaptive code was evaluated in terms of how well it was able to adapt to the provided solar trace.

Figures 7(a) and 7(b) show the results of the user study for the first and second applications, respectively. In Figure 7(a), the progress of each group is shown with respect to time spent (in minutes). Progress is measured as the percentage of correctness tests passed, with 100% meaning that all members of the group had passed all test cases.

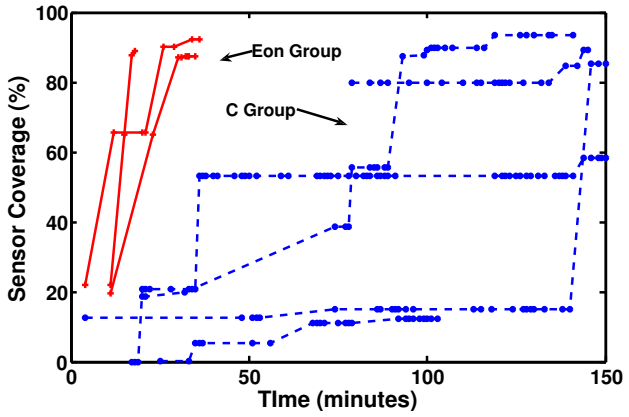
One striking feature is the similarity in progression between experienced C programmers and first-time Eon programmers. Members of the Eon group commented that the primary difficulties were learning Eon's syntax and understanding how Eon sources and flows are executed. Some commented that once these details were overcome, Eon provided a simple and intuitive programming style. We believe that the small difference between groups can be attributed to the Eon group's initial unfamiliarity with the language. We expect that experienced Eon programmers would be able to produce correct code for non-adaptive applications at least as quickly as programmers using a conventional general purpose language.

However, the results from the second application, shown in Figure 7(b), demonstrate the clear advantages of using Eon when building energy-aware software. This figure shows the performance of user submissions over time in terms of *percent coverage*. Every time a sensor reading is taken, an arbitrary amount of time t before and after the reading is considered "covered." The figure shows the percentage of time that was covered by at least one reading. For this experiment, we chose t such that the highest sustainable sampling rate would provide 100% coverage. Choosing a rate that is either too aggressive or too conservative results in reduced coverage. In this figure, we plot an individual line for each study participant. We also plot the *best solution so far* to make the figure easier to understand.

Unlike the results from the first stage, Figure 7(b) shows a substantial difference between the two groups. All members of the Eon group achieved 90% coverage within 40 minutes, while the C group lagged behind both in programming time and coverage. The Eon group's solutions were also uniformly good; this result stems from the effectiveness of Eon's runtime system. Three of the five members of the C group eventually achieved performance comparable to the Eon group, but took between 90 to 140 minutes to develop their solutions. Two of these solutions were inspired



(a) This figure compares experienced C programmers with first-time Eon users programming a simple sensor application. Aggregate group progress is shown over time. Despite language differences, both groups' progression is surprisingly similar. Small differences are likely due to the overhead of learning a new language.



(b) Percent sensor coverage (best so far) is shown comparing C and Eon programmers. By separating adaptation policy from execution, Eon users were able to build high-performing adaptive sensor applications significantly faster than those using C.

Figure 7. User study results

by TCP's exponential backoff. The other two C programmers' best submissions achieved 60% and 12% coverage, respectively. The longer programming times, high variance, and user comments all demonstrate the difficulty of writing adaptive software in conventional programming languages, even on a simplified sensor platform that avoids many common real-world complications.

6.2 Adaptation

One of the primary benefits of Eon is its ability to adapt the rate of flows in a program based on its currently available and predicted energy supply. Here we compare Eon's performance against several other possible systems and across individual devices.

To provide a fair and realistic comparison, we use trace-driven simulations based on data collected during the two week automobile deployment. During this deployment, each of the five nodes collected hourly measurements that we then fed into our trace-based Eon simulator. To avoid measuring transient behavior based on the initial battery state and to show long-term behavior, we loop the measured traces to extend our simulations from two weeks to three months, and report only the results for the last month. Each test was run five times, and the simulator generates the amount of energy used by the GPS drawn from the distribution gathered from each trace.

In each test case, we change the GPS sampling rate according to five energy policies: a *conservative*, static policy based on the minimum sustainable rate across all of the traces; a similar, *greedy*, static policy based on the maximum sustainable rate; a *best sustainable* rate taken for each device individually; *Eon* using the solar predictor algorithm ($T = 24$); and *Eon (Oracle)* that uses a perfect weather predictor that knows the exact amount of solar energy that can be harvested in the future. Note that the conservative policy is an over-provisioning implementation that a system designer may try first: collect traces, find the one that gave the least amount of energy, derive a static policy, and use that on all of the devices.

The results presented in Figure 8 show the average rate of GPS readings. The error bars represent the standard deviation of the rate *within* each trace averaged over the five runs. This demonstrates the variability of the policy over the duration of each run.

The results in Figure 8 show that a conservative policy, unsurprisingly, only performs well for device 5, from which the policy was derived. The rest of the devices pay a large opportunity cost for not using a more aggressive policy. The Eon (oracle) policy, best sustainable, and Eon provide similar average results. However, Eon shows more variance, demonstrating that misprediction in energy harvesting leads to a larger range of rates. It is important to note that neither the best sustainable policy, nor the oracular system can be realized, as both require advance knowledge of future solar trends.

We initially found it surprising that the Eon predictor would do as well as the oracle. However, a closer look reveals that given the size of the battery in the system and the typical rate of consumption, a full battery will last for five days. This lifetime means that the solar power prediction does not need to be extremely accurate day-to-day, as long as it is accurate on average. In systems where the ratio of consumption to battery size is higher, prediction algorithms have more impact. Lastly, the greedy system exhibits a high average rate for most of the traces, but its variation is high. This variation is because the node often ran out of energy, dropping the rate to zero for long periods of time.

Figure 9 shows a more detailed view of the results from the same experiment. The stacked bars show the breakdown of how energy was spent by the different policies. The per-

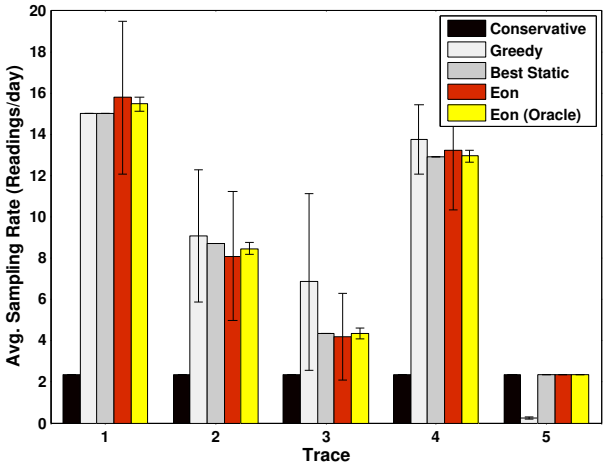


Figure 8. The average number of daily GPS readings taken are shown for different energy policies and energy traces. Despite large variations in energy supply, Eon is able to accurately approximate the best sustainable energy policy.

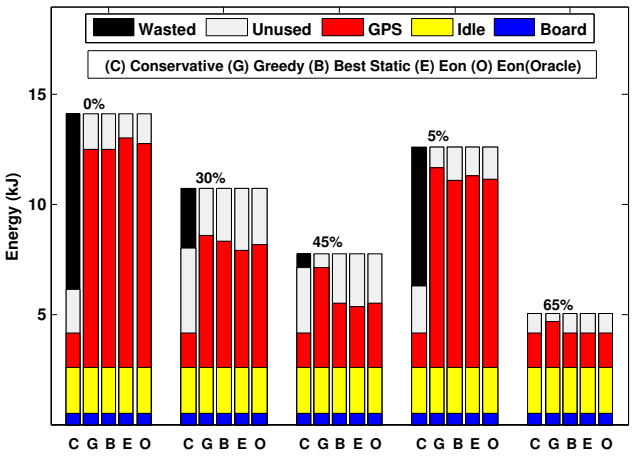


Figure 9. This figure shows the amount of each trace's energy that is consumed by different parts of the system. The percent dead time is also shown for traces that are not sustainable, above the corresponding bar.

centage numbers at the top of the bars show the average amount of time during the trace the device had a dead battery. The board overhead is the energy spent in the measurement board, the idle is the energy spent by the mote while not executing a flow (e.g. the overhead of the runtime system and the cost of an idling mote). The GPS energy was spent on taking samples, unused energy was energy left in the battery, and wasted is any energy that was collected, but could not be stored due to a full battery.

This graph shows the chief shortcoming of the greedy policy: aggressive use of energy leads to large periods of dead time. While sparse and bursty readings are generally undesirable, there is a more serious downside: the inability to run higher-priority flows. As we show in later experiments, when the program contains more than one flow, the dead time caused by one flow's overuse negates any prioritization the program may need.

6.3 Impact of Energy-State Based Paths

To examine the usefulness of energy-state based paths, we conducted a longer experiment using the remote camera application. Rather than conduct a year-long deployment, we used the solar cell to collect adequate solar traces for simulation, and then lengthened those traces using solar intensity data from the US Climate Reference Network, National Climate Data Center, and NOAA. By constructing a model that maps solar intensities to the power produced by the solar cells, we were able to extend the trace backwards for years' worth of data. Note that this process only works for the stationary camera. Long-term simulation of mobile nodes requires information about each node's mobility as well as the weather, to determine its energy budget.

Using the energy profiles collected from a running camera system and generated by our simulator, we compared the behavior of Eon against two systems, one that uses a fixed

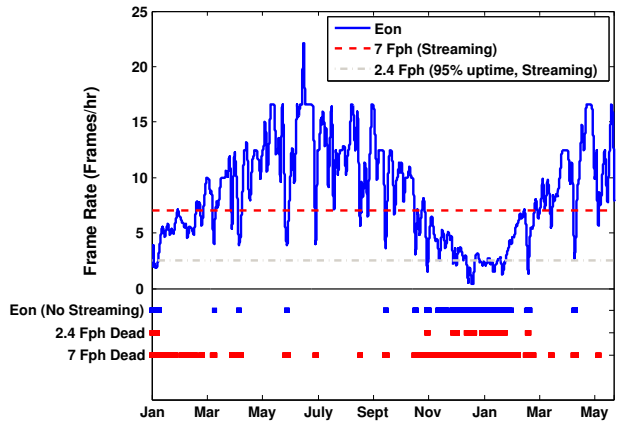


Figure 10. Frame rates for a remote camera application are shown over a 16-month trace, comparing Eon to two static policies. Periods of time when the Eon camera disabled streaming as well as periods of dead time for the static policies are shown across the bottom.

rate of 7 frames per hour (Fph) and one that uses 2.4 Fph. We then determined when each system had a dead battery, and thus could not respond to any queries for old images, and when the Eon system switched into its querying-only mode. The resulting frame rates, dead times, and querying-only times are shown in Figure 10. Note that the Eon system experienced no dead time for the entire trace, so we only plot dead time for the two static policies.

The results show that Eon can completely avoid dead battery times by adaptively switching into a query-only mode, while simultaneously lowering its frame rate. Note that it would be trivial to adjust this policy in Eon, e.g., to pre-

Energy Costs		
Operation	Energy	Time
Path Init	0.6μJ	0.3ms
Edge	1.4μJ	0.8ms
Path Cleanup	5.4μJ	2.1ms
GPS Reading	$1 - 100$J	$20 - 400$s
Evaluate State	$0.5 - 2.0$mJ	$50 - 100$ms

Figure 11. Measurements of Eon overhead in comparison to GPS readings.

fer higher frame rates over streaming. Without the ability to adapt, a fixed frame rate system may remain completely unavailable for months at a time. Eon is also able to scale its frame rate in tune with the seasons and short periods of cloudy weather.

6.4 System Overhead

In this section, we discuss the overhead incurred by using Eon on our turtle/automobile monitoring node.

Since the focus of Eon is energy, the energy overhead of the system must be kept to a minimum. Here, we measure the energy costs of several operations performed by the runtime system. We measure current draw using an Agilent 54621D oscilloscope, measuring the voltage drop across a 1-Ohm sense resistor. We integrate the trace to determine the energy cost of the operation. Figure 11 presents these energy measurements.

Periodically reevaluating the energy state, which presents the largest single energy cost, varies widely depending on the structure of the application graph and the state of the system. If, for example, the battery is low and little energy is expected, the algorithm will quickly rule out higher power states. More complex applications will also take longer than simple applications since they have more flows to consider.

As Figure 11 shows, the turtle tracking application requires up to 2.0mJ in the worst case to choose an energy state. However, since state evaluation happens only once per hour, this cost is easily amortized, resulting in an increase of only 2μW in the average power of the device. There is also a fixed overhead incurred every time a path is executed, which is equal to $(6.0 * 1.4N)\mu$ J where N is the number of edges in the given path. In comparison with the cost of taking a GPS reading, this overhead is insignificant, since it is at least 6 orders of magnitude smaller.

6.5 Measurement Accuracy

The runtime system's ability to accurately estimate the cost of individual paths in the program graph is vital to being able to make accurate adaptation decisions. We evaluate this accuracy by comparing measured task costs with the system's corresponding estimate for tasks that consume different amounts of energy. In this experiment we focus on small tasks (e.g. transmit data, write to Flash, etc) that consume a few mJ and large tasks (e.g. GPS readings) that incur a high energy cost.

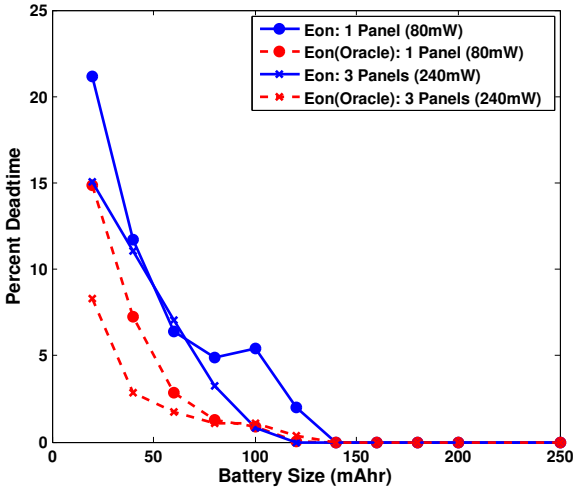


Figure 12. Device dead time is shown for different battery sizes for systems using one and three solar panels. Performance using Eon's EWMA predictor is compared with perfect energy prediction (Oracle). The benefit of better energy prediction is most notable when using a very small battery and the cost of prediction errors is greatest.

For small tasks (<100 mJ) the course grained averaging of our energy measurement board results in large errors in individual estimates (we observed up to 80%); however, averaging six consecutive estimates consistently yields an estimate within 10% of the measured value. For larger tasks (1-10 J) the Eon runtime system estimates the per-task energy cost to within 10% for individual task executions, and six consecutive estimates consistently results the measured cost to within 0.5%. The Eon runtime system benefits from this trend of increased accuracy for high-energy tasks. The penalty for mispredicting a small task is also small, and as these tasks are performed more frequently the system's cost estimate becomes more accurate. Mispredictions of large tasks, on the other hand, can have significant consequences on system lifetime, and cost estimates must be more accurate. Since large tasks are often performed infrequently, it is important to be able to provide accurate estimates with a small number of executions.

6.6 Impact of Battery Capacity

Our final experiment examines the impact that battery capacity has on Eon's ability to adapt, and on the cost of prediction errors. This experiment is set up as described in Section 6.2, except that we vary the size of the battery and the number of solar panels used.

The results of this experiment, shown in Figure 12, demonstrate how prediction errors are magnified as battery size decreases. While a 250mAhr battery is able to mitigate prediction errors, those prediction errors translate into large amounts of dead time when a 50mAhr battery is used. Applications that require a very small battery due to size and weight restrictions should use either more accurate or consistently conservative energy predictors.

We note that an additional benefit of Eon's automatically-generated simulators is the ability to use them to determine what size battery or solar panel to choose for a given deployment.

7. Related Work

Eon derives from a large body of work on energy adaptation in operating systems, as well as dataflow and coordination languages.

Languages: To our knowledge, Eon is the first system that specifically targets energy adaptation at the programming language level. Eon's energy adaptation features are built on a dataflow-based, coordination language [9, 22]. Eon uses this dataflow abstraction to expose just enough structure to make building an adaptive runtime system possible. However, in contrast with many dataflow languages, Eon's goal is to simplify writing energy-adaptive programs, rather than expressing concurrency or real-time ordering constraints [2, 11].

Coordination languages have also been proposed in order to simplify the programming of embedded sensors. SNACK [10] provides language constructs that combine components written in NesC [8], in order to simplify and encourage code reuse. The Flask language [19] has been developed concurrently with Eon [29], and both languages share many properties: both are coordination languages that combine nesC modules together in an acyclic graph. However, unlike Eon, Flask does not provide support for energy adaptation. In addition, Flask is a macroprogramming system, while Eon programs run on a single node. We view the use of dataflow-based coordination languages for embedded sensors as a natural response to the difficulty of writing event-based code without sacrificing the growing base of NesC and TinyOS modules.

Energy Application Adaptation: There has been a wealth of research on building systems that adapt to current conditions, including energy. Odyssey provided the seminal work in application-aware adaptation [23], and later work extended it to account for energy [7]. The Ecosystem project uses application adaptation to share energy fairly between applications, and governs that system's consumption rate [33]. In each case, energy-aware adaptation trades fidelity for energy savings to target a particular device lifetime. Eon builds on this concept by targeting perpetual operation, while expressing the adaptation policy as part of the program. This provides a much tighter integration of resources, programming language, and runtime system.

8. Future Work and Conclusions

We plan to build on this work in several areas. To date we have focused on balancing energy for a single device. We are now working on using Eon to manage energy in a network of devices—including a full-scale deployment on turtles this summer—which introduces new challenges and allows us to better understand how local adaptation decisions impact the network as a whole.

We also hope to improve Eon to incorporate the needs of new applications. For instance, timer sources take very little power, but receiving packets that are later discarded wastes energy. By turning off sources in low-power modes, we can avoid these costs completely.

In conclusion, we have presented Eon: a new language and runtime system for self-adapting perpetual systems. Designed to be both expressive and simple, Eon eases the burden of building energy-adaptive applications. The Eon runtime system effectively manages changing energy availability and demands, while hiding most of the complexity.

9. Acknowledgments

This work was funded by National Science Foundation grants CNS 0519881, CNS 0520729, CNS 0347339, and CNS 0447877. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning wireless network power management. In *Proceedings of the 9th ACM International Conference on Mobile Computing and Networking (MobiCom'03)*, San Diego, CA, September 2003.

[2] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[3] Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D. Berger, and Mark D. Corner. Flux: A language for programming high-performance servers. In *Proceedings of USENIX Annual Technical Conference*, May 2006.

[4] Wei-Peng Chen and Lui Sha. An energy-aware data-centric generic utility based approach in wireless sensor networks. In *Proceedings of the third international symposium on Information processing in sensor networks (IPSN)*, pages 215 – 224, April 2004.

[5] Henri Dubois-Ferriere, Roger Meier, Laurent Fabre, and Pierre Metrailler. TinyNode: A comprehensive platform for wireless sensor network applications. In *Proceedings of the fifth international conference on Information processing in sensor networks (Poster)*, pages 358–365, Nashville, TN, USA, April 2006.

[6] John Porter et. al. Wireless sensor networks for ecology. *BioScience*, 55(7), July 2005.

[7] J. Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. *ACM Transactions on Computer Systems (TOCS)*, 22(2), May 2004.

[8] D. Gay, P. Levis, R. V. Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.

[9] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96, 1992.

[10] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (snack). In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 69–80, New York, NY, USA, 2004. ACM Press.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[12] C. Hartung, S. Holbrook, R. Han, and C. Seielstad. olbrook, r. han, c. seielstad, "firewxnet: A multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In *Fourth International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 28–41, 2006.

[13] P. Juang et al. Energy-Efficient Computing for Wildlife Tracking: design tradeoffs and early experiences with ZebraNet. *SIGOPS Oper. Syst. Rev.*, 36(5):96–107, 2002.

[14] Aman Kansal, Jason Hsu, Mani B Srivastava, and Vijay Raghunathan. Harvesting aware power management for sensor networks. In *43rd Design Automation Conference (DAC)*, July 2006.

[15] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B Srivastava. Power management in energy harvesting sensor networks. *ACM Transactions on Embedded Computing Systems*, May 2006.

[16] P. Kulkarni, D. Ganesan, and P. Shenoy. Senseye: A multi-tier camera sensor network. In *ACM Multimedia*, 2005.

[17] Kris Lin, Jason Hsu, Sadaf Zahedi, David C Lee, Jonathan Friedman, Aman Kansal, Vijay Raghunathan, and Mani B Srivastava. Heliomote: Enabling long-lived sensor networks through solar energy harvesting. In *Proceedings of ACM Sensys*, November 2005.

[18] Geoff Mainland, David C. Parkes, and Matt Welsh. Decentralized, adaptive resource allocation for sensor networks. In *Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005)*, May 2005.

[19] Geoffrey Mainland, Matt Welsh, and Greg Morrisett. Flask: A language for data-driven sensor network programs. Technical Report TR-13-06, Harvard University, Division of Engineering and Applied Sciences, May 2006.

[20] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Workshop on Wireless Sensor Networks and Applications*, Atlanta, GA, September 2002.

[21] S. Meninger, J. O. Mur-Miranda, R. Amirtharajah, A. Chandrakasan, and J. H. Lang. Vibration-to-electric energy conversion. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):64–76, February 2001.

[22] J. Paul Morrison. *Flow-Based Programming: A new approach to application development*. Van Nostrand Reinhold, 1994.

[23] B. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, and K. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, St. Malo, France, October 1997.

[24] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, April 2005.

[25] J. Polastre, R. Szewczyk, C. Sharp, , and D. Culler. The mote revolution: Low power wireless sensor networks. In *Proceedings of the 16th Symposium on High Performance Chips (HotChips)*, August 2004.

[26] Joseph Polastre, Jason Hill, and David E. Culler. Versatile low power media access for wireless sensor networks. In *SenSys*, pages 95–107, 2004.

[27] Shashank Priya, Chih-Ta Chen, Darren Fye, and Jeff Zahnd. Piezoelectric windmill: A novel solution to remote sensing. *Japanese Journal of Applied Physics*, 44(3):104–107, 2005.

[28] Anthony Rowe, Charles Rosenberg, and Illah Nourbakhsh. A low cost embedded color vision system. In *Proceedings of Intelligent Robots and System*, pages 208–213, EPFL Switzerland s, September 2002.

[29] J. Sorber, A. Kostadinov, M. Brennan, M. Corner, and E. Berger. eFlux: Simple automatic adaptation for environmentally powered devices. (poster/demo). In *Proc. IEEE workshop on Mobile Computing Systems and Applications (HotMobile/WMCSA)*, April 2006.

[30] R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar, and J. Light. The Personal Server - Changing the way we think about ubiquitous computing. In *Proceedings of Ubicomp 2002: 4th International Conference on Ubiquitous Computing*, Goteborg, Sweden, September 2002.

[31] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, Seattle, WA, November 2006.

[32] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient mac protocol for wireless sensor networks. In *21st International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1567–1576, New York, NY, USA, June 2002.

[33] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proceedings of the Tenth international conference on architectural support for programming languages and operating systems*, San Jose, CA, October 2002.