eFlux: 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, MA
$\{$*sorber, akostadi, mgarber, mbrennan, mcorner, emery*$\}$*@cs.umass.edu*

Abstract

A key goal of mobile computing is untethering devices from wires, making them truly portable. While mobile devices can make use of wireless communication for network connectivity, they are still dependent on an electrical connection for continued operation. This need for tethering to available electricity significantly limits their range, usefulness, and manageability. Environmental energy harvesting—collecting energy from the sun, wind, heat differentials, and motion—offers the prospect of unprecedented, large-scale deployments of *perpetual mobile systems* that never need to be recharged. However, programming these systems presents new challenges: perpetual systems must adapt dynamically to available energy, delivering higher service levels when energy is plentiful, while consuming less energy when energy is scarce.

This paper presents eFlux, a high-level *energy-aware* programming language and associated runtime system that specifically targets perpetual mobile systems. eFlux programmers build programs from components written in C or NesC and label *flows* through the program with different *energy-states*. The deployed program then adapts to current energy levels by changing energy states, turning flows on and off and adjusting their rates. We demonstrate eFlux's utility and portability with two perpetual applications deployed on widely different hardware platforms: a solar-powered web server for remote, ad-hoc deployments, and a GPS-based location tracking sensor that we have deployed on a threatened species of turtle as well as on automobiles.

1 Introduction

A key goal of mobile computing is untethering devices from wires, making them truly portable. As computing moves from a handful of mobile devices per user to hundreds of specialized devices, fully wireless operation will become essential. While wireless networking has eliminated the need for wires for network communication, nearly all devices depend on frequent wired connections to electricity in order to recharge their batteries. The continual need for human involvement in power management significantly limits the feasibility of having users manage large numbers of mobile devices. The assumption that a person will recharge batteries has also led to a narrow class of solutions to mobile energy management. Whether the system is implemented at the hardware, operating system, or application layer, the goal has been the same: maximize battery lifetime or target a particular lifetime, while minimizing the impact on user-perceived performance.

Environmental energy harvesting fundamentally changes these assumptions. Solar, wind, heat differential, and motion-derived energy sources provide the opportunity to build mobile systems that do not require regular human intervention to charge batteries, thus enabling large-scale deployments of mobile systems that are completely untethered.

As two examples of perpetual systems, consider the following: deploying remote outdoor access points and tracking wildlife. In the first case, remote deployments on volcanoes [23] and in forests, solar energy offers the opportunity to cheaply deploy access points that require no maintenance. The case for tracking wildlife is even stronger: systems such as ZebraNet [11] have shown that using solar cells to power portable sensors makes perpetual wildlife tracking possible.

However, environmental energy sources also present some unique challenges. The amount of available energy is often difficult to predict, and may change dramatically with location, time of day, time of year, and current weather. Traditional energy management, that merely tries to minimize energy, does not consider these factors. As an example of these variations consider two traces shown in Figure 1. Each bar corresponds to the amount of energy gathered by one of two mobile, solar-powered devices over a two week period. Although both devices show elements of the same general weather trend, the two devices show significant variation in the amount of gathered energy. Similarly, Figure 2 shows the amount of energy that one device required to take a GPS reading over that same two week period. Due to mobility, and consequently the amount of time required to synchronize with satellites, the amount of energy varies over an order of magnitude. Even worse, comparing these two graphs demonstrates times of plentiful energy do not necessarily coincide with times of plentiful need.

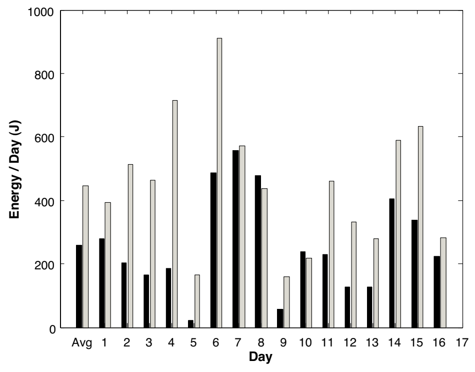

Figure 1: A histogram of the average amount of daily energy gathered by two devices over a two week period.

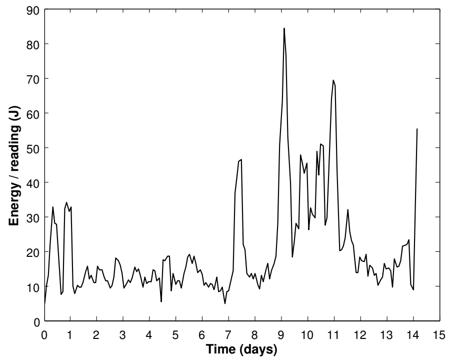

Figure 2: The amount of energy needed to take a GPS reading over the same period shown in Figure 1.

It is crucial not only to avoid running out of energy, but also to take advantage of available energy in order to benefit the device's users. For example, a solar-powered tracking device could increase its sampling rate when energy is plentiful. A mobile data server could augment text data with images, audio, and video as its energy budget allows. Development of accurate measurement and prediction of incoming and outgoing energy is essential in implementing those policies. In any application where adaptation is beneficial, managing harvested energy requires an ongoing process of finding the best operating state that the energy source will allow. The expression of these adaptation policies, in conjunction with the logic of the program, is difficult for programmers to implement without extensive experimentation and data harvesting.

This paper presents the design and implementation of a new language and runtime system, *eFlux*, for programming perpetual computing systems. eFlux allows programmers to build programs from code written in a variety of different languages (e.g., NesC and C). By breaking eFlux *flows* into *energy state-based paths*, a programmer can easily write programs which provide different service levels based on current available energy. The eFlux runtime system then adapts to available energy on the fly. Because eFlux operates at a high level of abstraction, it is portable: eFlux can generate code for a variety of embedded platforms, including Linux and TinyOS.

To demonstrate eFlux's utility and portability, we have built and deployed two perpetual applications on embedded platforms with very different hardware characteristics. The first application is a solar-powered adaptive web server capable of providing text, videos, and images in a untethered deployment. The second, and more ambitious system, is a preliminary deployment of perpetual location-tracking sensors on threatened turtles in Western Massachusetts. Current attempts to protect these animals have been limited by an inadequate understanding of how they use their habitat. Researchers currently use radio telemetry to track these turtles, which requires up to three man-hours per location: this severely limits the number of animals that can be studied. A tracking device with a GPS receiver running eFlux records the turtle's location over time, providing a much finer granularity of data. We demonstrate the efficacy of our perpetual location tracking system both with an initial deployment on turtles, and a large-scale deployment on automobiles.

2 eFlux Language Description

eFlux is a domain-specific language intended to support perpetual systems. These include a broad range of 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. eFlux 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.

One method of building a language to support perpetual systems is to start from scratch, incorporating energy as a first-class concern, while trying to balance energy adaptation with the features found in any programming language. Such an approach would require programmers to learn a new language while muddling basic constructs such as loops and conditionals with policy. An alternative approach is to build a set of annotations to an existing language, thus easing the learning-curve for new programmers, and leveraging existing analysis tools and compilers. While this approach is frequently used to help support legacy code bases, it does not translate well, as the annotation syntax would have to be adapted to each new language. More importantly, the resulting system would still muddle the issues of adaptation with logic.

Instead, we have built eFlux as a *coordination language* [7] that ties together code written in a conventional programming language, like Java, C, or nesC [6]. This approach provides programmers with a high level of abstraction that separates the concerns of energy adaptation from program logic.

In addition to eFlux's system for energy adaptation, it is inherently portable between different languages (Java/C/NesC) and operating systems (Linux/TinyOS/-SoS). Porting an eFlux program from an XScale-based device to a mote-class device requires only a modification of any platform specific code used to implement the program logic. This makes eFlux a natural candidate for use in mobile devices, given the wide variety of platforms, operating systems, and languages currently in use.

2.1 Flux

We have built eFlux using the Flux [4] programming language as a starting point. While Flux was developed for high-performance servers and to separate the concerns of concurrency and logic, its programming model is also a natural fit for many embedded devices. Like servers, which respond to asynchronous requests, embedded systems typically respond to events like timers, sensed events, or packet arrivals. Before describing eFlux, we provide a brief overview of the Flux language.

Flux is a declarative language that describes the flow of program control from event sources through event handlers. A Flux program is a directed acyclic graph (DAG) consisting of the following elements: *concrete nodes, abstract nodes, predicate types, error handlers*, and *atomicity constraints*. For compactness, in explaining Flux, we use examples from the eFlux program shown in Figure 3. The additions for eFlux, explained later in this section, are denoted in the source code comments.

Concrete nodes correspond to code written in a conventional programming language which performs a specific task. In Figure 3, *GetGPS*, *LogGPSData*, *LogConnectionEvent* are all concrete nodes, and are implemented by the programmer. Each concrete node takes a set of input arguments and produces an output set of arguments. For instance, *GetGPS* takes no input and produces two output variables: a GpsData_t and a boolean. A distinguished concrete node called a *source node* produces only output and initiates the execution of other nodes. The Flux compiler combines these concrete nodes into a single program targeted to a specific architecture.

Abstract nodes describe the flow of control and data through multiple concrete or other abstract nodes. They thus provide a modular way to build Flux programs. In Figure 3, *GPSFlow* is an abstract node.

Conditional flows are implemented in Flux using *predicate types*, programmer-defined Boolean functions that are applied to a node's output. In Figure 3, the *StoreGPSData* abstract node specifies multiple possible execution paths. By applying the *full* predicate to the output of *StoreGPSData*, the Flux program decides the appropriate path to take. Multiple paths in a Flux program have semantics similar to those of `switch` statements in C. Paths are tested in the order that they are listed in the code, and the first matching path is chosen. Flux also supports exception handling via error handlers, and allows programmers to control concurrency with atomicity constraints.

2.2 eFlux

The eFlux language comprises a set of extensions to the basic Flux language. While Flux lets programmers define the sequence of operations that follow from events, it lacks any method to express runtime adaptations. In eFlux, we add constructs which let a programmer describe what runtime adjustments to make, as well as the priority with which they should be applied. This application is then mapped to an adaptive runtime system, which continually adjusts the application in order to balance the demands of fidelity and sustainability.

The keys to adaptation in an energy-limited system are to understand: (i) the sequence of operations that follow from external events, (ii) the probable costs of those operations, (iii) the probable workload in the system, and (iv) the probable amount of energy the system will acquire, and (v) the adaptation policy. In each of these cases, the eFlux programmer provides just enough information to allow an adaptive runtime system to measure the rest on-line. All five of these concerns are addressed in two layers: the programming language and the runtime system. Matters particular to a program, such as the sequence of operations and the adaptation policy, are relegated to the eFlux language, and matters that can be automatically measured, or are particular to underlying hardware and energy sources, are relegated to runtime systems.

In providing language support for adaptation, our goal is to balance the opposing goals of simplicity and expressibility. An overly simple mechanism would limit the ability to express real adaptation policies. On the other hand, the ability to express all possible adaptation policies would likely negate the benefits of using Flux by significantly increasing the complexity of the language and underlying runtime system. Our goal is to allow most real adaptation policies to be expressed with a minimal set of features.

In this section, we describe eFlux's energy adaptation features. We continue to use the application shown in

```
// Flux Concrete Source Node Declaration
ListenBeacon() => (msg_t msg);

// eFlux Timer Declaration
GPSTimer() => ();

// Flux Concrete Node Declarations
GetGPS() =>
   (GpsData_t data, bool valid);
LogGPSData(GpsData_t data bool valid)
        => ();
LogGPSTimeout(GpsData_t data bool valid)
        => ();
LogConnectionEvent(msg_t msg) => ();

// Flux Abstract Node Declarations
HandleBeacon(msg_t msg) => ();
GPSFlow() => ();
StoreGPSData(GpsData_t data, bool gotfix)
        => ();

// eFlux States
// there is always an implicit BASE state
stateorder {HiPower};

// Flux Sources
source ListenBeacon => HandleBeacon;
source timer GPSTimer => GPSFlow;

// eFlux Adjustable Timer Limits
GPSTimer:[HiPower] = (1 hr,  10 hr);
GPSTimer:[*] = 10 hr;

// Flux Flows
GPSFlow = GetGPS -> StoreGPSData;
HandleBeacon:[*,*][HiPower]
        = LogConnectionEvent;
StoreGPSData:[*,gotfix][*] = LogGPSData;
StoreGPSData:[*,*][*] = LogGPSTimeout;
```

Figure 3: A reduced version of eFlux source for the turtle tracking application.

Figure 3 as an example.

Power states

Adaptation policies are often expressed as a set of *utility* functions that describe the relative value of different operations in a system. Both 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. A particular difficulty is that of finding a common unit of measure between different components.

In contrast with previous approaches, we have found that a simple partial ordering of service levels 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.

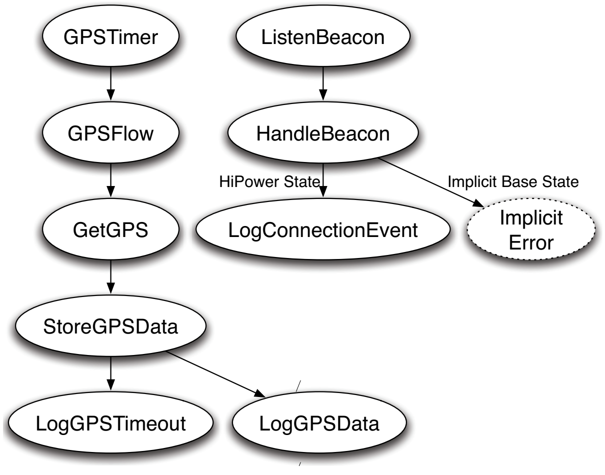

Figure 4: A graph of a simplified turtle tracking application

In an eFlux program, a programmer specifies an adaptation policy as a collection of behavior adjustments organized in a partial order, or lattice, called a *state ordering*. In a state order, adjustments are both declared and assigned priority. 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 5 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 than all lower states.

The state ordering of an eFlux program defines which operating states can be chosen by the runtime system. In addition to declaring adjustments, the system designer also must 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 components, such as radios [20, 1] and storage devices [9]. In the turtle tracking application described in the introduction, 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

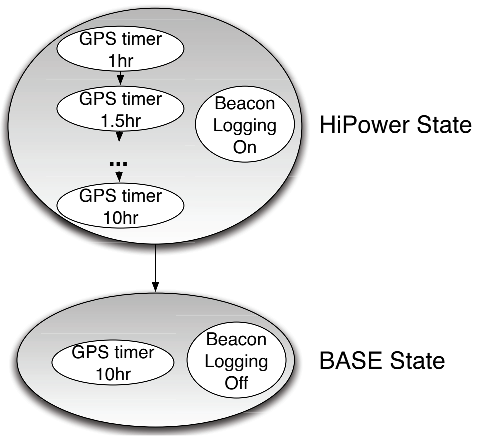

Figure 5: Sample State Order.

or activity represents a trade-off between application fidelity and energy consumption.

Duty-cycle adaptation is implemented in eFlux using a special type of event source node called an *adaptive timer*. Adaptive timers differ from traditional Flux 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 10 hours. The interval is then set by the runtime system.

Energy-Based Service Levels

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 device that answers queries for different kinds of data might respond only to requests for text data, while ignoring more expensive requests.

Fidelity and availability adaptation is provided in eFlux using *energy-state based paths*. This concept is akin to conditional execution in Flux, 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, *LogConnection* is called when *HandleBeacon* produces any type, and is in a state labeled *HiPower*. If the node runs low on energy, it may enter the implicit BASE state, and cease logging beacons form 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, eFlux lets the programmer express this preference for local operations over providing services to other nodes when energy is low.

Discussion

One feature that we considered but rejected during the development of Flux 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 one more level in between. While this is not true of eFlux's timers—they are finely adjustable—the semantics of a timer, and the resulting energy cost is 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. However, 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 that 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.

3 Runtime System and Hardware Support

Mobile systems that depend on harvested energy are unusual in several ways. Harvesting energy presents the prospect of very long-lived mobile systems that are limited primarily by the aging and decay of physical materials, rather than by energy consumption. However, in order to successfully operate with harvested energy, it is essential that these mobile systems gracefully adapt to changes in energy. When running, an eFlux runtime system must make predictions about the balance of incoming and outgoing energy. By using the flow descriptions in the program, on-line measurements of the cost of operations and workload, and predictions about the amount of incoming energy, the runtime system can adapt according to the program's policies.

One design goal in developing the runtime system was to avoid any explicit training, such as measuring the system under load in a lab. Not only is this process painful for programmers, it is inherently fragile, possibly requiring repeated measurement every time the program is changed. Similarly, the system must be adaptive to different battery sizes and peripherals such as radios.

The runtime energy adaptation, takes as input a set of measurements and predictions about consumption and production, and makes a decision about the ideal state of the system. Such a broad mandate leads to many possible runtime system designs. We have focused on building a single runtime system that is suitable for a broad array of low-power platforms, such as Motes [19] and Stargates [22], powered by solar energy. The Mote platform in particular has a relatively small memory size, constraining the runtime system to a small set of online measurements. The use of solar energy leads to a particular set of energy prediction algorithms.

3.1 Energy Consumption Model

The energy consumption model of a program includes: (i) the energy cost of each path through the program, (ii) the rate at which each source node produces an event, and (iii) the predicate probabilities in the system. Each time an eFlux flow completes, the runtime system track an exponentially weighted moving average (EWMA) of the energy cost of the flow, the average frequency with which the source fired, and the exact flow that the request followed. For instance, in the example in Figure 3, there are three possible flows through the program, each with a different energy cost and frequency. Measuring the consumption of each path requires hardware support, described later in the section.

3.2 Energy Source Model

Adapting to changing energy sources also requires a model of how much energy the system is going to receive in the future. While eFlux is not tied to any particular energy production method, we have chosen to concentrate on solar power. In the case of solar, the amount of energy is highly variable, and is only semi-predictable—predicting sun intensity is similar to predicting the weather. However, two EWMA models have been previously proposed.

In the Environmental Energy Harvesting Framework, the authors use an EWMA filter to predict future solar energy production [12, 13]. This EWMA essentially predicts that the energy production in the next day will be largely similar to recent days. Further, the predictor makes the same prediction for subsequent days as it does for the next day.

While their work uses the prediction algorithm as part of a larger cost function, we have extracted the prediction filter for use in eFlux. The algorithm is as follows. The system measures the energy production in a time period, t, that is T_e in duration, and assigns this value as $E(t)$. It then applies an EWMA to predict the expected value of $E(t+1)$ as follows:

$$E(t+1) = \alpha E(t) + (1-\alpha)E(t-1) \qquad (1)$$

The time period T_e is taken as a whole day [12]. An alternate method is to split the predictor across an array of 48, half-hour, slots that takes time of day effects into account [13]. The second method consumes more memory, but is useful in certain scenarios—the evaluation demonstrates the effective differences in these predictors.

3.3 Energy Adaptation

Using the measurements of consumption and production, the adaptation system chooses the ideal state for the system to use. The overriding goal in adaptation is to avoid two states: an empty battery and a full battery, while providing the highest fidelity to the application. In the first case, an empty battery removes the ability to make adaptation decisions altogether, even to run high priority flows. In many devices, it also imposes a period of dead time for the system to recover the battery from a fully-discharged state—the battery must slowly charge up to a minimal level before the device can turn on again. In the case of a full battery, any additional energy that the system gets is wasted, and could have been used for increasing the fidelity of the application. Any state of the battery between these two states is equivalent, and the system makes adjustments to avoid an empty and full battery. In an ideal sense, the particular size of the battery is irrelevant, as the system must consume energy at a rate equal to the rate of energy production—-the battery only acts as a buffer to ride out periods of low energy production and to store excess energy.

Periodically, the runtime system makes a decision about the ideal state for the system. The design of the algorithm is primarily driven by simplicity, and we have targeted eFlux to run on a variety of platforms, including embedded, memory and CPU-limited microcontrollers. For less constrained platforms, more complex algorithms can be used.

To find the target state, eFlux starts by assuming the system runs at the highest energy state in the lattice, with the minimum frequency for all of the timers. It then reads the capacity of the battery, and computes the state of the battery at the end of a short interval of time T_i, using the estimates of consumption and production over that interval. If at the end of an interval, the battery is not empty, eFlux computes the battery state at time $2^n \cdot T_i$ for a horizon of $n = \{1..N\}$ time intervals. If at any of the intervals, the node exhausts its battery, the runtime system starts over with a lower power state, and minimum timers. Once it finds a sustainable state, it performs a binary search of the space of possible frequencies, to find a set of timers that is sustainable and maximal in frequency. This process consumes just 100 ms of computational time for our turtle tracking program with 31 flows and a time horizon of half a year.

Although our runtime system tracks path costs by EWMAs, it could capture more complete information with histograms. A histogram yields adaptation solutions that are tunable by the *probability* of running out of energy, rather than by just the expected result. We leave this possibility as future work.

3.4 Hardware Requirements

As mobile computing hardware devices have proliferated, the need for accurate energy information has increased. Most modern batteries sold with laptops and PDAs contain an IC to measure and model the discharge and charge characteristics. These are frequently referred to as Gas-Gauge, or Fuel-Gauge chips; two popular example include TI's bq27000 and Maxim's DS2770. These chips include corrections for temperature, battery-chemistry, and for aging and memory in batteries. However, they typically provide only a subset of information needed for building effective runtime systems. The hardware provides an averaged, large-grained view of the remaining energy in the battery, and the current rate of charge or discharge—this is insufficient to tell the individual rates of consumption and charge as both occur simultaneously.

Thus, in addition to a fuel-gauge chip, the runtime system requires fine-grain current measurement to attribute energy to individual program flows. For this purpose, we have added a current sensor, a Maxim DS2751, to our system, which separately measures the rate of consumption. The current sensor measures current to within 0.6mA, which is accurate 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. To attribute energy to individual flows, we measure the fraction of time that the flow was running during the previous second and assign it that portion of the previous second's energy consumption. The rest of the energy is attributed to the runtime system and to the idle energy consumption of the platform. Adaptation can only affect the consumption of flows, not the overhead consumption. Any errors in the system do not *accumulate* as the fuel-gauge chip corrects eFlux's notion of the battery capacity. We quantify eFlux's measurement accuracy in the evaluation section.

Given the amount of energy consumed by the program and runtime system, we can also estimate the energy production rate. Adding the energy consumption over a period to the loss or gain in battery capacity yields the energy production over that period. This quantity is then fed into the energy prediction model as described previously.

4 Implementation and Deployment

eFlux brings together new hardware elements such as charging control and solar power, a new compiler, two runtime systems, as well as deployment. The designs for the hardware, as well as a release of the application code, compiler, and runtime system, are all available from our website (`prisms.cs.umass.edu`).

4.1 Hardware

One of our goals for eFlux is portability to a wide variety of hardware platforms and energy sources. Thus far, we have implemented eFlux on two different hardware platforms, a Mica2Dot mote, and an Intel/CrossBow Stargate, both powered by solar cells.

To support eFlux's adaptation algorithms, we have built a new charging and energy management board. This board 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. This board was designed to accept a Mica2DOT mote as a drop-in module to the board. We adapted some parts of the hardware design from the Heliomote project [14]. As the hardware needs of a Stargate-based system are similar, we connect the board and Mica2DOT to the Stargate via serial to manage and measure the power consumption of the system. The deployment platform for a turtle device, with board, Mica2Dot, battery, and GPS is shown in Figure 6. This board can handle a wide variety of solar cells, ranging from a small, 25mA peak current cell up to an array of 20-100mA cells, depending on component selection. Additionally, eFlux requires no runtime changes when changing the number of cells, since it only tracks the amount of energy production, not how it was produced.

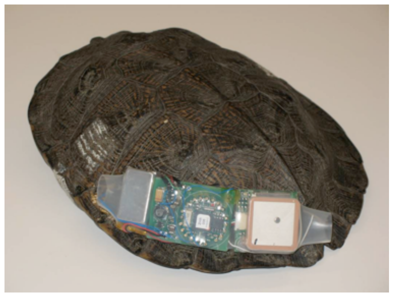
Figure 6: The energy measurement and charging board with a Mica2Dot, GPS receiver, and battery.

4.2 Compiler

The eFlux 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 [4], extended with support for energy adaptation. The first two stages of the compiler are the same as Flux: it builds a graph representation of the program and then decorates each edge with input and output types. The third stage links this intermediate code with the eFlux adaptive runtime system and user-supplied code that contains the program logic.

eFlux can be ported to new languages and architectures with minimal effort. Our current implementation targets two different environments: (i) an XScale-based Linux system, using nodes written in C, and (ii) an Atmel microcontroller-based TinyOS system using nodes written in nesC [6]. There are some key differences between the two environments. In a C/Linux environment, the programmer supplies C functions that accept the same types as described in the eFlux program. The runtime system then invokes these functions as blocking calls, and waits for the return value to pass to the next node in the graph. For programs that require concurrency, any of the concurrency mechanisms, such as threads and events, work just as they do in the Flux runtime system.

The implementation for the nesC/TinyOS environment is less straightforward. As TinyOS is based on split-phase, event-based operations, the programmer-supplied function does not follow the blocking semantics that eFlux depends on—the function that implements a node returns before completion. In this case, programmers are required to make a small change to their code to make the modules work with eFlux. The start of each node is called with a nesC command, which is akin to a function call, and the node signals its completion with an asynchronous event containing the return values.

One feature of Flux that we have not incorporated fully into eFlux is *atomicity constraints*. Flux gives the programmer the ability to encode the atomic constraints into the Flux program, making it easier to build highly concurrent runtime systems. As our embedded programs are not highly concurrent, eFlux supports minimal atomicity constraints by guaranteeing that a particular node will not be run more than once simultaneously.

4.3 Runtime System

The eFlux 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.

4.4 Trace-Based Simulator

We also modified the compiler to automatically generate a trace-based simulator at compile time. By feeding an energy trace and traces for external inputs, we can test different solar predictors, workloads, programs, and adaptation policies. During deployment, the eFlux 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.

4.5 Deployment

In order to evaluate eFlux, we have completed three different deployments that span a range of power constraints. While these deployments are somewhat limited in their scale and duration, we have gathered sufficient data to demonstrate eFlux's utility in performing energy adaptation. Perhaps more importantly, these deployments have driven the development of eFlux, rather than following as a consequence of it—the applications inform which features to add to the language, runtime system, and hardware support. We plan to significantly lengthen and increase the range of the experimentation with the language in the future.

Turtle Tracking

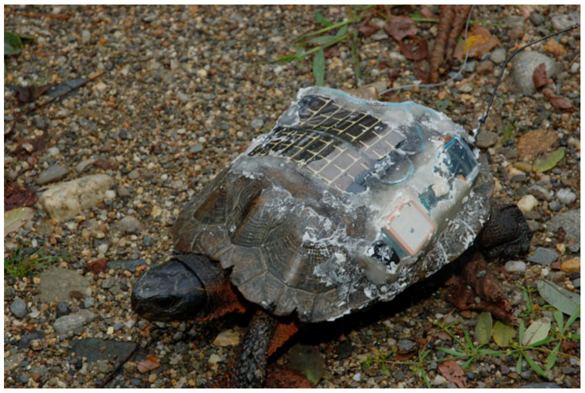

Figure 8: Photo of an eFlux node on a Turtle.

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; however, they are also 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

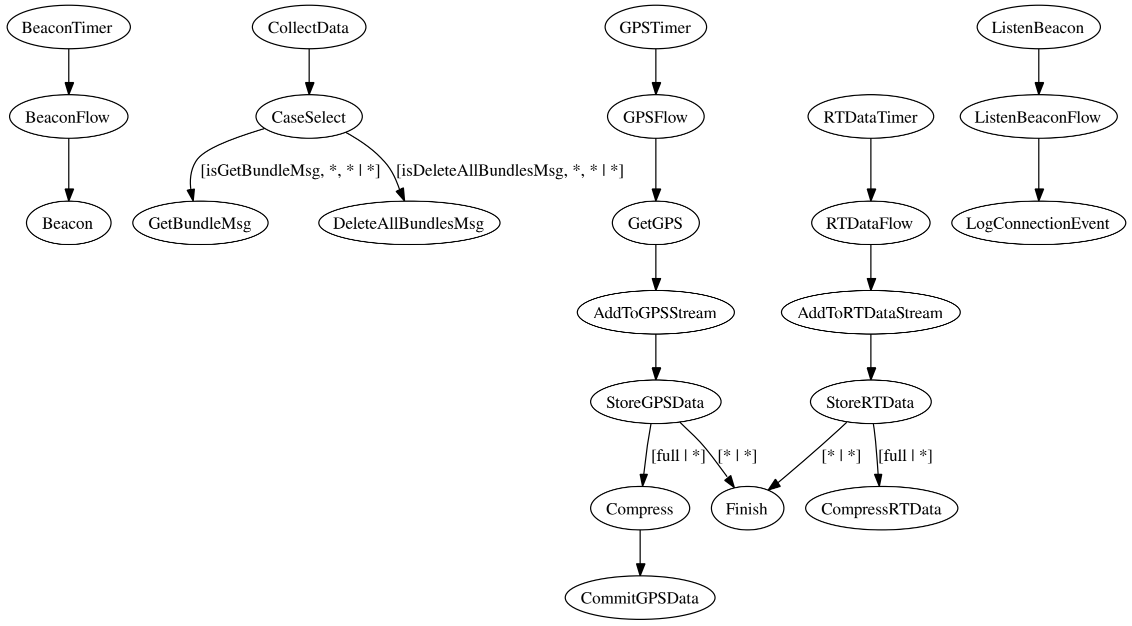

Figure 7: Graph representation of the full turtle-tracking application.

collection of location information at night. In order to accurately understand how these turtles behave and use their habitat, new tracking methods are required to collect data at finer granularity.

Much of the development of eFlux is inspired by this particular problem. We have designed and built an eFlux node and program to run on the Mica2DOT environment. The graph and code listing for the full eFlux program are shown in Figures 7 and 9. 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 8 shows the eFlux node mounted on a turtle's shell.

Unfortunately, as our deployment took place at the very end of an unusually cool fall, the turtles prepared for hibernation early, and spent a large amount of their time immobile and underwater. 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.

Automobile Tracking

As a proxy for the turtles, we performed a second deployment using automobiles. We used the same hardware, adaptation policy, 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

```
typedef full IsStreamFull;
typedef isGetBundleMsg IsGetBundleMsg;
typedef isDeleteBundleMsg IsDeleteBundleMsg;
typedef isDeleteAllBundlesMsg IsDeleteAllBundlesMsg;

platform MICA2DOT:

//Node definitions
//Beacon flow
   BeaconTimer() => ();
   BeaconFlow() => ();
   Beacon() => ();
//Listen flow
   ListenBeacon() => (uint16_t addr, uint8_t version);
   ListenBeaconFlow(uint16_t addr, uint8_t version) => ();
   LogConnectionEvent(uint16_t addr, uint8_t version) => ();
//Runtime data collection flow
   RTDataTimer () => ();
   RTDataFlow () => ();
   AddToRTDataStream() => (uint32_t num);
   StoreRTData(uint32_t num) => ();
   CompressRTData(uint32_t num) => ();
//GPS Timer flow
   GPSTimer() => ();
   GPSFlow() => ();
   GetGPS() => (GpsData_t data);
   Compress(uint32_t num) => (stream_t stream);
   CommitGPSData(stream_t stream) => ();
   AddToGPSStream(GpsData_t data) => (uint32_t num);
   StoreGPSData(uint32_t num) => ();
   Finish(uint32_t num) => ();
// Data Collection flow
   CollectData() => (int action, uint16_t bundle, uint16_t src_addr);
   CaseSelect (int action, uint16_t bundle, uint16_t src_addr) => ();
   GetBundleMsg(int action, uint16_t bundle, uint16_t src_addr) => ();
   DeleteAllBundlesMsg(int action, uint16_t bundle, uint16_t src_addr) => ();

//eFlux states (implicit BASE state)
stateorder {HiPower};

//Sources
source ListenBeacon => ListenBeaconFlow;
source CollectData => CaseSelect;
source timer BeaconTimer => BeaconFlow;
source timer GPSTimer => GPSFlow;
source timer RTDataTimer => RTDataFlow;

//Flows
BeaconFlow = Beacon;
ListenBeaconFlow = LogConnectionEvent;
GPSFlow = GetGPS -> AddToGPSStream -> StoreGPSData;
RTDataFlow = AddToRTDataStream -> StoreRTData;

StoreRTData:[full][*] = CompressRTData;
StoreRTData:[*][*] =  Finish;

CaseSelect:[isGetBundleMsg,*,*][HiPower] = GetBundleMsg;
CaseSelect:[isDeleteAllBundlesMsg,*,*][HiPower] = DeleteAllBundlesMsg;

StoreGPSData:[full][*] = Compress -> CommitGPSData;
StoreGPSData:[*][*] = Finish;

//Adjustable Timer Limits
BeaconTimer:[*] = (25 min, 25 min);
RTDataTimer:[*] = (1 hr, 1 hr);
GPSTimer:[*] = (1 hr,  10 hr);
```

Figure 9: The Full eFlux source for the turtle tracking application.

gives us a good idea of how eFlux 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 spring, the evaluation we present here is based on data gathered from the automobile-based experiment. The introduction includes two sample graphs of incoming solar energy and the energy required to take a GPS reading. Also, a sample map of GPS locations is shown in Figure 10.

Solar-Powered Web Server

For a third deployment we built a solar-powered webserver in eFlux. We adapted this program from a webserver written for Flux, and extended it with an energy adaptation policy. The eFlux webserver serves text, images, audio, and video. When energy runs low it prioritizes text over images, images over audio, and audio over video. The full eFlux program graph and source listing is shown in Figures 11 and 12. This deployment allowed us to test several different elements of eFlux. First, it demonstrates the portability of eFlux between platforms of different characteristics. Second, it highlights eFlux's ability to adapt to a much larger solar cell—in this case, one 80 times larger in area than the one deployed on the turtles. Finally, it demonstrates the ease with which an existing program can be adapted to eFlux.

The webserver consists of the same charge board and Mica2Dot assembly used in the turtle deployment, an Intel/CrossBow Stargate with a LinkSys WCF12 802.11 card, and an array of 20 thin film solar cells. Since PSM is not supported in ad-hoc or master mode when using the hostap drivers, we reduce the idle power consumption by operating at a 25% duty cycle—25% of the time the node is active and accepting requests, and 75% of the time is

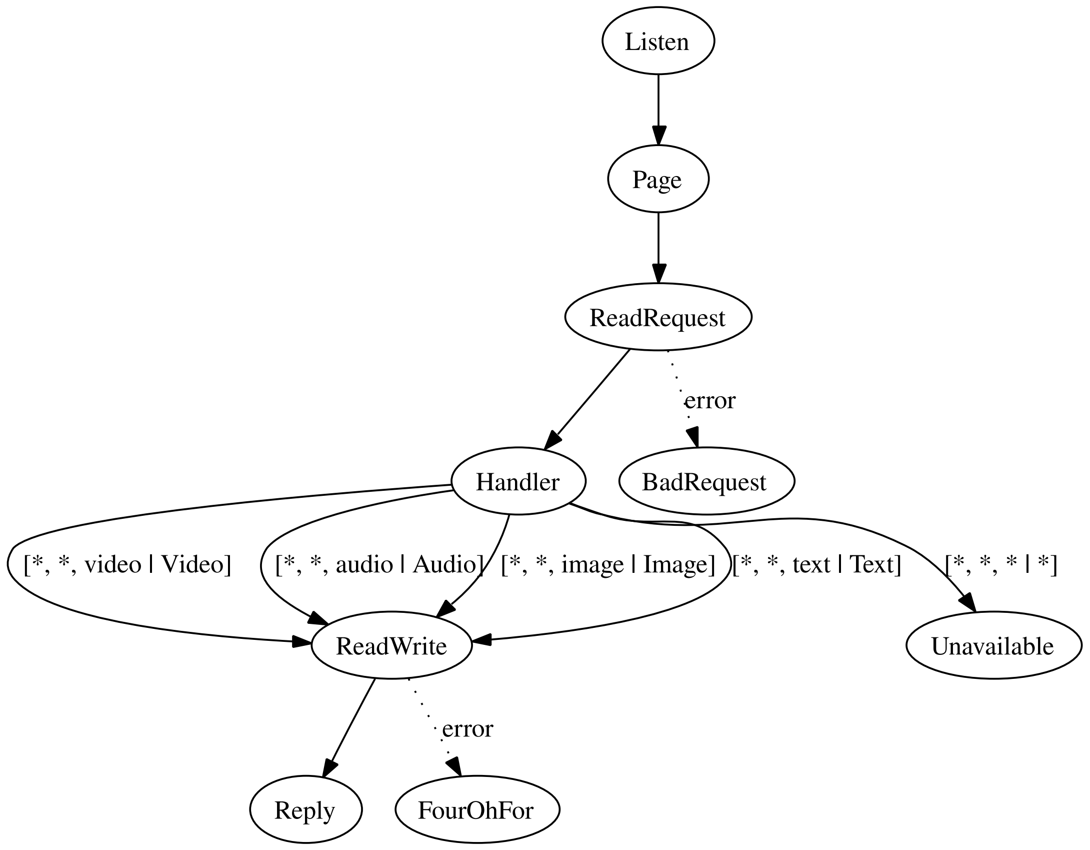

Figure 11: Graph representation of the full adaptive webserver application.

remains suspended. If there are any clients associated with the base-station it remains active until they disassociate. This duty-cycle brings the idle power consumption down to 400mW, from the active power consumption of 1600mW. We deployed the solar cell array on a few different days 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 computing a model that maps solar intensities to the current produced by the solar cells, we can extend the trace backwards for years worth of data. Note that this process only works for the stationary webserver; because the turtles and cars are mobile, we cannot use a straightforward model to quantify the greater variations in solar intensity that would result.

One improvement that we hope to make in the webserver is to incorporate techniques from Triage [2] to significantly lower the idle power cost of the webserver. The Stargate platform does not have as great of dynamic range, limiting the range that adaptation has to work with.

5 Evaluation

Our primary goal in eFlux is to provide the maximum sustainable service level without sacrificing availability and with minimal overhead to the program. In this section we evaluate the effectiveness of eFlux against this goal.

5.1 Adaptation

The experiments described in this section demonstrate the differences between different adaptation policies. In order to providing a fair comparison between approaches, we use trace-driven simulations, based on data collected during the two week automobile deployment described previously. During this deployment, each of the five nodes collected hourly measurements that we then feed into the simulator described in the previous section.

In these experiments we compare static policies, which do not adapt but try to take GPS readings at a fixed frequency, and eFlux adaptive policies. We use ten static policies which vary from 0.1-1.0 readings per hour—the

```
typedef text TestText;
typedef image TestImage;
typedef audio TestAudio;
typedef video TestVideo;

platform STARGATE:

//Node definitions

Listen () => (int socket);
Page (int socket) => ();
ReadRequest (int socket) => (int socket, bool close, char* file);
Handler (int socket, bool close, char* file) => ();
Reply (int socket, bool close, int length, char* content, char* output) => ();
ReadWrite (int socket, bool close, char* file)
        => (int socket, bool close, int length, char* content, char* output);
Unavailable(int socket, bool close, char* file) => ();

//eFlux Power States
stateorder  {Video,
        Audio,
        Image,
        Text
        };

//eFlux Sources
source Listen => Page;

//Flows
Page = ReadRequest -> Handler;

Handler:[*,*,video][Video] = ReadWrite -> Reply;
Handler:[*,*,audio][Audio] = ReadWrite -> Reply;
Handler:[*,*,image][Image] = ReadWrite -> Reply;
Handler:[*,*,text][Text] = ReadWrite -> Reply;
Handler:[*,*,*][*] = Unavailable;

//Error Handlers
handle error ReadWrite => FourOhFor;
handle error ReadRequest => BadRequest;
```

Figure 12: Full eFlux source for the adaptive webserver application.

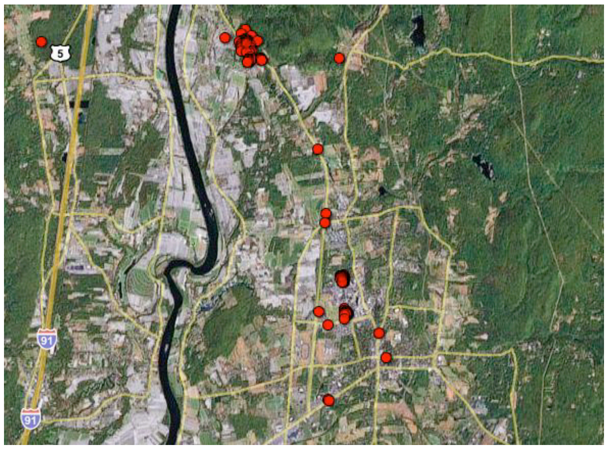

Figure 10: A sample GPS trace from an automobile mounted node.

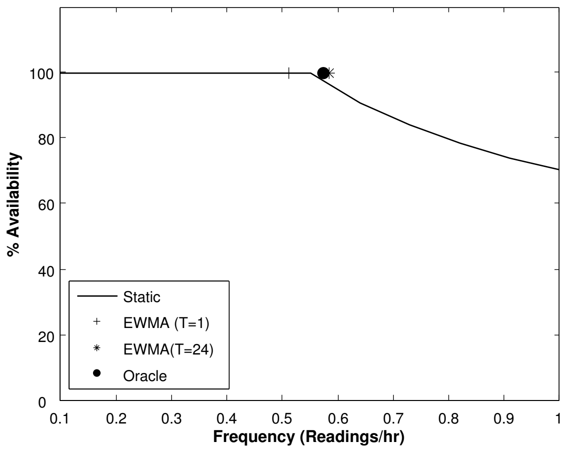

Figure 13: System availability is shown with respect to energy policy.

same dynamic range available to the adaptive policies. We also evaluate eFlux using three different energy predictors: the default predictor which uses a EWMA filter to predict daily energy(T=1), an hourly EWMA predictor(T=24), and a hypothetical oracle predictor, which represents system behavior given perfect energy prediction. We simulate each of these 13 policies for all 5 collected traces. For the static policies we show them as continuous lines, as intermediate values are also valid static policies.

In order to avoid measuring transient behavior based on initial battery state and to show long-term behavior, we loop the measured traces to extend our simulations from two weeks to two months, and report only the results for the last month. These results are discussed in the following paragraphs.

Figure 13 shows the resulting availability for a single trace with respect to the frequency of taking GPS readings; adaptive policies are shown with respect to their average frequency. In the static policies when the frequency exceeds what the energy source can sustain, the device availability drops off sharply, resulting in large gaps in the collected data. All three adaptive approaches

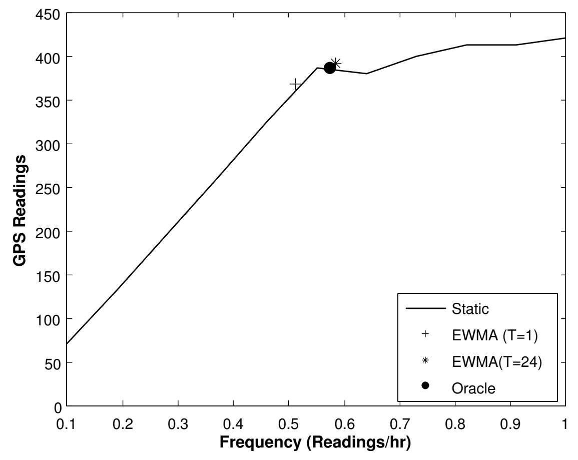

Figure 14: The number of GPS readings taken are shown with respect to the sampling frequency.

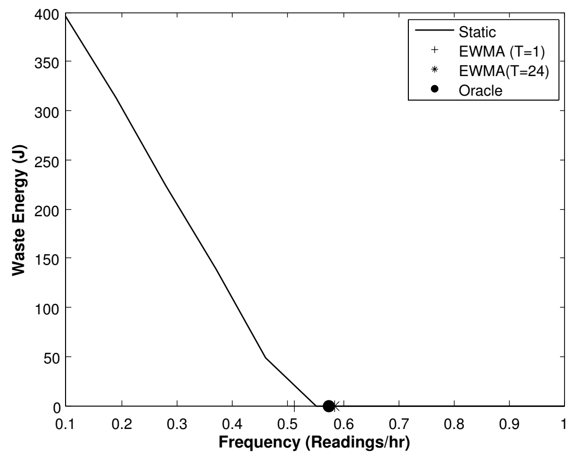

Figure 15: The amount of wasted energy is shown with respect to the frequency of taking GPS readings. Even with simple energy source predictors, the eFlux runtime system is able to effectively use all available energy.

closely approximate the best sustainable static policy. In addition, both EWMA-based systems achieve performance comparable to using an oracle, in spite of their simplicity. In our deployment, none of the five devices depleted their batteries at any time.

At first, we found that it surprising that the predictors 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 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, the prediction algorithms will have more impact.

Similar results are shown in Figures 14 and 15 with respect to total GPS readings and wasted energy. In Figure 14, the frequency of GPS readings increases linearly until the rate is no longer sustainable. Figure 15 shows the energy wasted by overly conservative policies–in this case the battery fills and it can no longer store solar energy as reserves. The figure demonstrates that the adaptive systems successfully approximate the best sustainable static policy with respect to both of these metrics.

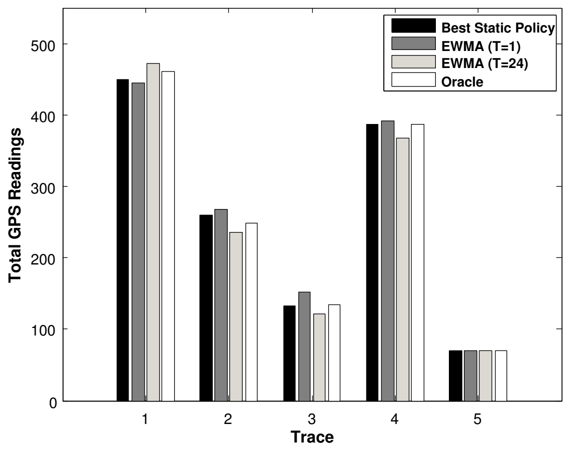

Figure 16: The number of GPS readings taken are shown for different energy policies and energy traces. Despite large variations in energy supply, eFlux is able to accurately approximate the best sustainable energy policy.

In the interest of space, Figures 13, 14, and 15 show these results for a single trace. The corresponding figures for the other four traces are almost identical except for a difference in scale. The bar graph in Figure 16 shows how the number of total GPS readings varies between traces. Even though all five traces were collected concurrently, their individual mobility patterns result in very different sustainable operating states. This further strengthens our case for energy-adaptive systems, as using any one of the ”best” static policies for all of the devices will perform much worse than the adaptive systems.

5.2 System Overhead

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

Since the focus of eFlux 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. Since the node expends a small amount of energy when triggering task boundaries, these are, in fact, conservative estimates of the actual task costs. These energy measurements are shown in Figure 17.

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 17: Measurements of eFlux overhead in comparison to GPS readings.

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 shown in Figure 17 the turtle tracking application requires up to 2.0mJ, in the worst case, to choose a energy state; however, since state evaluation happens only once per hour, this cost is easily amortized, resulting in an increase of only 2μW to the average power of the device. Also, there is 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—differing by at least 6 orders of magnitude.

The memory and code overhead of our eFlux runtime system is also reasonable. On the Mica2DOT it requires only 750 bytes of RAM—including a 300-byte heap which stores flow variables—and 8 bytes of additional space for each path in the program graph. In the applications we have tested, the runtime system also increases the ROM footprint by 25KB-35KB, which also depends on the size of the eFlux program. This fits easily into the Mica2DOT's severely constrained memory (4KB RAM, 128KB ROM).

5.3 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. We vary the energy consumed by the tasks by changing the duration of an operation with roughly constant power draw—in this case, we turn the GPS unit on for a period of time which we vary from 100ms to 30s.

Figure 18 shows the results of this experiment. The line represents perfect estimation accuracy—when the measured and estimated costs are equivalent—and a point's distance above or below the line represents the estimation error. Note that the system accurately estimates the cost of heavyweight tasks (>1J) such as GPS readings, but underestimates smaller tasks. As a result of averaging performed by the fuel gauge chips, much of the energy consumed by these short tasks is attributed by the system to its idle consumption.

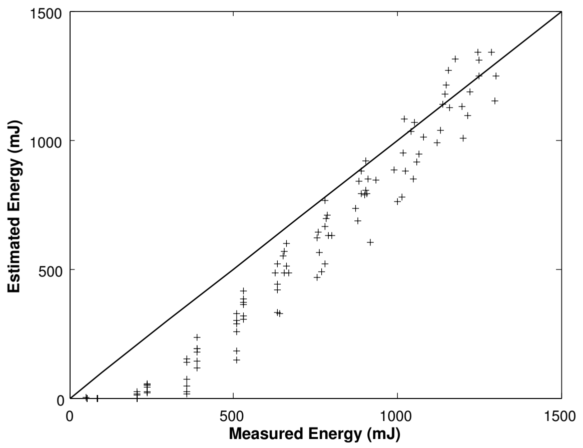

Figure 18: The dots represent the node's energy measurements of a path, while the line represents the true energy cost. The system is accurate for larger energy flows.

In applications, such as turtle tracking, where heavyweight tasks largely determine the system's lifetime, inaccurate estimates of small tasks has no noticeable effect. In an application with only low-energy tasks the system's estimate of its idle power consumption will likely change due to adaptation decisions, causing its adaptation choices to oscillate. We are currently working on both on improving cost estimation for small tasks and exploring the consequences of inaccuracy for small tasks.

5.4 Impact of Battery Capacity

Our final experiment examines the impact that battery capacity has on the ability to adapt and the cost of prediction errors using our adaptive webserver. Recall that the webserver reduces its energy consumption by denying requests for certain types of data, such as video and images, that come at a high energy cost. For this experiment we simulate application behavior for a 2-month solar energy trace using measured per-path energy costs for battery capacities varied from 1-4 Ahr. We compare a webserver that answers all requires, Full Service, with the three adaptive policies discussed earlier.

The results of this experiment are shown in Figure 19. The non-adapting server is unable to sustain operation with the given workload regardless of how large the battery is. Also, the ability to adapt and the impact of prediction errors are also highly dependent on battery size.

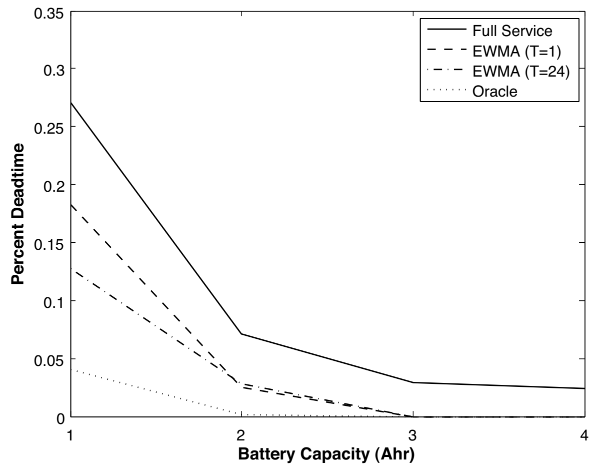

Figure 19: Webserver dead time is shown for different energy policies and battery sizes. The ability to effectively adapt is limited for small batteries; however, without adaptation, no battery is large enough.

With a small battery both EWMA-based policies perform significantly worse than the ideal; however, as battery capacity increases this difference is no longer apparent. With a small battery, prediction errors are magnified.

An additional benefit of the automatically generated simulator is the ability to use it as a design tool when determining what size battery or solar panel to choose for a given deployment.

5.5 Programmer Experience

Our primary goal is to make writing adaptive software simple and straightforward. While we have not yet conducted usability studies to quantify this aspect of eFlux, our experience has been positive. For example, an undergraduate student, with no prior Flux or eFlux experience, took only 2 hours to convert the Flux web server—written for the x86—into a working adaptive web server which runs on the PXA-based Stargate platform. By distributing the compiler, runtime system, and node designs, we hope to build a user-base to further develop the eFlux code and to gather feedback on the language design.

6 Related Work

eFlux derives from a large body of work on energy adaptation in operating systems, as well as data-flow and coordination languages.

Languages: To our knowledge, eFlux is the first system that specifically targets energy adaptation at the programming language level. Domain-specific languages are a powerful technique to address the difficulty in expressing the same logic in a general-purpose language.

Coordination languages [7] typically allow separate pieces of code to communicate with one another in order to link code written in a language together in a new way, or to marry two or more disparate languages. However, a coordination language, such as eFlux, can also be used to incorporate modules from a second language to implement the logic of the program.

eFlux is also a dataflow, or flow-based language [17, 10]. eFlux uses this dataflow abstraction to expose just enough structure to make building an adaptive runtime system possible. However, in contrast with many dataflow languages, the focus is not on providing a visual programming language, although eFlux programs are easily understood in their graphical form.

The closest language to eFlux is the recent Flask programming language for sensors [16], which has been developed concurrently with eFlux [21]. Flask shares many properties of eFlux: both are coordination languages that can tie NesC [6] modules together in a graph. Flask is based on the OCaml language, and at a high-level describes a directed acyclic data-flow graph, just as eFlux does. The use of dataflow-based coordination languages for embedded sensors is a natural response to the difficulty of programming event-based NesC code [6] that takes advantage of the growing base of NesC and TinyOS modules. However, Flask does not provide support for energy adaptation, the primary contribution of the eFlux language. Similarly, eFlux does not provide support for the distribution of programs over multiple nodes in a network, as Flask does.

eFlux and Flask are not the first languages to apply dataflow concepts to embedded programming. Synchronous languages, such as Esterel [3] and Lustre [8], use data flow to describe a completely deterministic ordering of events in real-time systems for use in safety-critical systems. The environments we are targeting for eFlux have much looser timing requirements, enabling a much more expressive programming language to describe the logic of the program.

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 [18], while later work extended the system to account for energy [5]. The Ecosystem project used application adaptation to fairly attribute energy costs to applications and then enforce a policy that contains applications to using the battery at a certain rate [25]. In each of these cases, energy-aware adaptation trades application fidelity for energy savings to target a particular lifetime in a device. eFlux builds on this concept by targeting perpetual operation, while expressing the adaptation policy as part of the program. eFlux also targets a much tighter integration of resources, programming language, and runtime system. eFlux concentrates on measuring energy in-situ and making adaptation decisions in concert with a predicted affect on consumption.

Application-Driven Energy Management: Instead of relying on the operating system to provide hints of resource availability, new systems have emerged that take a more cooperative approach to energy management. By providing hints from the application, such as in GraceOS [24], the application can provide enough information for the operating system to provide statistical guarantees on performance while minimizing energy usage in the processor. In an even more direct approach, we have opened interfaces to allow the application to *directly* control to power management of the processor for the duration of the application's scheduling quantum [15]. Such an approach necessarily provides more control, at the cost of programming effort.

7 Future Work and Conclusions

We plan to build up on this work in several key areas. First, we plan to improve the measurement and modeling of energy both in production and consumption. In particular, we would like to improve the attribution of energy for small flows to separate idle costs from adaptable costs. Second, we hope to improve the language to incorporate the needs of new applications. For instance timer sources take very little power, but receiving packets that we later discard wastes energy. By turning off sources in low-power modes, we will be able to avoid these costs all together. Third, we plan to complete a full-scale deployment on turtles next spring. Recent advances in GPS receivers have yielded much more efficient chip sets that should quadruple the number of GPS readings.

In conclusion, we have presented a new language and run-time system for self-adapting perpetual systems. We have designed eFlux to be both expressive and simple, easing the burden in building energy-adaptive applications. We have shown that we can build a well-performing runtime system that bests static policies and performs on-par with an oracular energy-harvesting system that knows the future weather. As all sides of the energy equation improve, the mobile community will find new uses for devices that operate continuously and autonomously.

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] N. Banerjee, J. Sorber, M. D. Corner, S. Rollins, and D. Ganesan. Triage: A Power-Aware Software Architecture for Tiered Microservers. Technical Report 05-22, University of Massachusetts Amherst, Amherst, MA, April 2005.

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

[4] 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.

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

[6] 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.

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

[8] 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.

[9] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the Second ACM International Conference on Mobile Computing and Networking (MobiCom'96)*, Rye, NY, November 1996.

[10] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[11] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with ZebraNet. In *ASPLOS*, October 2002.

[12] 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.

[13] 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.

[14] 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.

[15] Xiaotao Liu, Prashant Shenoy, and Mark D. Corner. Chameleon: Application controlled power management with performance isolation. In *Proceedings of ACM Multimedia 2005*, Singapore, November 2005.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of the Symposium on Operating Systems Principles*, pages 149–163, Bolton Landing, NY, October 2003.

[25] 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.