A Portable Interface for Runtime Energy Monitoring: Extended Analysis

Connor Imes - University of Chicago  
ckimes@cs.uchicago.edu

Lars Bergstrom - Mozilla Research  
larsberg@mozilla.com

Henry Hoffmann - University of Chicago  
hankhoffmann@cs.uchicago.edu

ABSTRACT

As energy consumption becomes a first class concern for computing systems, there is an increasing need for application-level access to runtime power/energy measurements. To support this need, a growing number of power and energy monitors are being developed, each with their own interfaces. In fact, the approaches are extremely diverse, and porting energy-aware code to new platforms with new hardware can involve significant rewriting effort. To reduce this effort and support portable, application-level energy monitoring, a common interface is needed. In this paper, we propose EnergyMon, a portable application interface that is independent of underlying power/energy data sources. We demonstrate EnergyMon’s flexibility with multiple case studies, including energy-aware profiling and self-adaptive systems, which require monitoring energy across a range of hardware from different manufacturers. We release the EnergyMon interface, implementations, utilities, and Java and Rust bindings and abstractions as open source.

1. INTRODUCTION

Power and energy consumption are becoming increasingly important measures of software quality, especially as battery-driven mobile platforms become ubiquitous. Even small improvements in power/energy management can save significant amounts of battery and extend a system’s usable runtime by hours. As a result, there is a growing need for software to access power/energy data in-situ. However, accessing that data requires writing system-specific code which is challenging for engineers and results in applications that are not portable across platforms.

The diversity in sensor properties and current ad-hoc approaches for using them presents a significant challenge in accessing runtime power/energy measurements. The problem is more complex than simply locating and reading the correct files, hardware registers, or memory addresses. Other factors must be considered. Some sensors report energy while others report power. Energy counters eventually overflow. Some sensors, particularly external devices, require exclusive access. Each may use different units of measurement with various levels of precision. All sensors have refresh intervals. Sensors which only report power data must be polled at their refresh interval so as to not lose data after the sensor refreshes. Power/energy readings may need to be extracted from other data or transformed before they are meaningful [32, 35, 42]. Developers may even have to implement different protocols and data formats [32, 42]. Furthermore, even a single system equipped with power or energy monitoring can offer multiple approaches to obtain it [10, 21, 35]. Correctly and accurately collecting power and energy data requires understanding and properly managing sensor properties and behavior.

Despite these difficulties, software increasingly relies on power and energy metrics. In general, software systems that use this data fall into two categories: profilers that characterize an application’s power and energy usage, and self-adaptive applications that actively modify their behavior to manage power or energy consumption. Power/energy profiling is important for applications ranging from mobile/embedded platforms [2, 13, 29, 33, 36, 46] to more traditional end user systems and High Performance Computing environments [9, 25, 43]. Other approaches create self-adaptive applications [6, 24, 37] that change their runtime behavior to achieve better energy efficiency or respect runtime power/energy constraints [1, 12, 14, 17, 20, 40]. However, all this prior work uses ad-hoc power/energy monitoring approaches. None have proposed a common, portable approach for exposing the required energy consumption data to software.

A common application interface for portable and practical access to power and energy monitoring hardware is clearly needed. To this end, we propose EnergyMon, the first application-level interface designed specifically for exposing energy metrics from diverse sources in a portable manner. This paper makes the following contributions:

- Describes the challenges and importance of capturing runtime energy metrics in software, motivating the need for a common interface.
- Proposes EnergyMon, a portable API for accessing energy data from diverse sources.
- Demonstrates EnergyMon’s versatility in usage, with performance/power application and system characterization, power/energy-aware profiling, and self-adaptive applications as case studies, executed on various platforms with different power/energy data sources and distinct performance/energy behavior.
- Open-source release of the EnergyMon API, current implementations, bindings and abstractions in Java and Rust, and additional utilities.

Furthermore, EnergyMon is publicly available in Servo, Mozilla’s parallel web browsing engine [3] (see Section 5).

2. BACKGROUND AND MOTIVATION

This section motivates the need for a portable, application-level energy monitoring interface. We first discuss the challenges pre-

1 Available at https://github.com/energymon

This work was partially conducted while Connor Imes was at Mozilla Research.

The University of Chicago’s effort on this project is funded by the U.S. Government under the DARPA BRASS program, by the Dept. of Energy under DOE DE-AC02-06CH11357, by the NSF under CCF 1439156, and by a DOE Early Career Award.
sent by the array of different power and energy sensors available across modern hardware platforms. We then demonstrate the importance of collecting runtime energy data by discussing prior work in building software systems for both energy profiling and self-adaptive power/energy management.

### 2.1 Challenges

As power and energy consumption become ever more crucial measures of software quality, hardware power and energy sensors are becoming more prevalent in computing systems. The diversity of available sensors complicates the process of collecting and interpreting the data they produce. Any portable interface for energy monitoring must account for the following differences:

- **Data Type**: Whether sensors expose power or energy data.
- **Access**: Access privileges/constraints, e.g., exclusiveness.
- **Units of measurement**: The power or energy units that sensors expose measurements in, and their precision.
- **Overflow**: Maximum values for sensor counters.
- **Refresh Interval**: How frequently sensors update.
- **Interface**: System abstractions, protocols, and data formats.

Table 1 demonstrates these differences for the data sources used in this work. The first column are the nicknames we use to refer to the sensor access implementations, the second describes the underlying data source, and the rest present the details for properties listed above. The fact that refresh intervals span three orders of magnitude, from the MSR’s 1 ms to the WattsUp? 1 second, exemplifies how diverse sensors can be. In some cases there are even multiple implementations for the same data source because they expose multiple data types or interfaces.

One property that necessitates further discussion is the data type — whether sensors expose power or energy data. Assume a sensor exposes cumulative energy data. Software that requires energy data simply takes the difference in readings at the beginning and end of the event being evaluated. Software interested in power does the same, but divides the difference in energy by the elapsed time to get an average power. Power sensors expose a value that is the average power over an interval (e.g., one second), and are more complicated to use. To record energy, software must read the power in each interval, multiply that value by the refresh interval to compute energy, and keep a running total. Algorithm 1 demonstrates this approach. If the power sensor refreshes before its value is read, the energy consumption over that interval is lost since the next power value may be different. Software interested in power must also poll at the refresh interval and keep a running average of the readings during the desired time interval.

Software that reads from a power or energy data source must understand its behavior in order to use it properly. Such a task may involve managing sensor polling threads, tracking elapsed time, limiting the frequency of sensor accesses, performing I/O, parsing and transforming data readings to consistent units, and sharing the data with other software components. For example, our energy monitors for the data sources listed in Table 1 require roughly 250-300 lines of code on average. Without a portable interface, software that

### Algorithm 1 Computing Energy from Power Sensors.

```plaintext
Require: PowerSensor ≫ The power sensor
Require: RefreshInterval ≫ The sensor’s refresh interval

Energy ← 0 ≫ Energy counter, initialized to 0

loop

function POLL_PowerSensor ≫ Local function

Wait(RefreshInterval)

Power ← ReadPower(PowerSensor)

Energy ← Energy + Power × RefreshInterval

end loop

end function

function ReadEnergy ≫ Externally visible function

return Energy

end function
```

relies on power or energy data must contain all of this platform-specific code to account for the diversity in data sources.

### 2.2 Power and Energy-Aware Software

Software requires runtime access to energy data for various purposes including profiling and self-adaptation.

**Power and Energy-Aware Profiling**: With the proliferation of energy-constrained systems like smartphones, tablets, and now the Internet of Things, power and energy-aware profiling has become increasingly important for developers who previously have not been concerned with this aspect of software behavior. PowerScope is one of the early works that recognized the importance of profiling energy usage for mobile applications [13]. More recently, Banerjee et al. used Android device power profiles to detect energy hotspots and bugs in applications [2]. Other research has proposed new power modeling techniques, investigated the behavior of SoCs or components like GPS and cameras, and explored power behavior that is unique to smartphones like tail power states and wakelocks [30, 33, 34, 36, 46]. Power and energy-aware profiling is not limited to mobile applications and devices. Cui et al. instrument PC hardware to capture power with fine-grained time granularity as low as 20 microseconds [9]. PAPI 5 provides access to hardware performance counters including those that measure power/energy data exposed by common hardware in HPC environments [43].

**Self-Adapting Applications**: Applications that monitor and adjust their behavior at runtime to obey performance, power, and energy constraints are becoming more common [37]. Software can even exhibit contrasting power/energy behavior on different hardware [18]. To address these problems, some applications access power/energy data in deployment to automatically adapt their behavior. The Green language creates adaptive applications that minimize energy given an acceptable accuracy [1], while the Eon language and runtime adapt embedded applications to the availability of harvested energy [40]. The Odyssey [14], GRACE [41, 44], CoAdapt [16], JouleGuard[17] and xTune [22] projects automatically coordinate applications and systems to meet real-time goals.
with reduced energy in mobile and embedded systems, while Mohapatra et al. provide a similar framework for distributed systems [31]. The POET library enables applications to tune their own resource usage to meet soft real-time constraints with minimal energy [20]. Similar projects automatically tune server applications to meet power goals with maximum performance [7, 19, 45]. Klein et al. recently proposed a technique to alter webpage content based on available energy and power resources [23].

2.3 The Need for a Common Interface

Prior works have used an assortment of sensors, but none have proposed a common interface specifically for making energy data available to software in a portable manner. Tools like PAPI are useful in some cases, but do not present data in a consistent format and instead just provide direct access to the hardware sensors. Thus, PAPI and similar low-level tools require making application code changes to support new power/energy data sources [43]. This requirement might be reasonable for the HPC domain it is designed for, but fast becoming impractical for software like mobile applications that are deployed on a wide variety of continually evolving systems. In POET, the authors created a primitive interface that was internal to their tool in an attempt to access energy data in a more portable manner than had been done previously [20]. POET is representative of many self-adaptive systems in that it is not designed to report energy usage, but simply use that data internally. Such systems do not expose implementation properties like refresh interval, precision, or exclusiveness, nor do they enable sensible bindings to other programming languages. Additionally, their implementations do not manage practical concerns like counter overflows or preservation of error codes.

As software matures, its shelf life increases. That software will continue to be deployed on new hardware for years to come, with few upfront guarantees about the hardware’s power/energy properties or how it will expose such data. Additionally, it is infeasible for most software to manage the challenges in correctly using power/energy sensors for more than a handful of data sources. Much of the information needed to effectively use power/energy sensors like the access restrictions and operating system abstractions are not important to the software that needs the data – they are peripheral concerns. An interface can hide these details while exposing important properties in a common manner, like the actual energy values, the refresh interval, and the sensor’s precision, while requiring only a few lines of platform-independent code. We address the challenges described in Section 2.1 and aforementioned problems in prior work with EnergyMon, a portable interface for accessing energy data in-situ that is independent of the data source. It can easily be used by current and future power and energy-aware software without the need to change code for different deployment scenarios or even know the available sensors in advance.

3. DESIGN AND IMPLEMENTATIONS

3.1 Interface Design

The first important design decision is which data type to expose in a common interface – we choose energy. Unlike power, energy is not explicitly a function of time and can be recorded cumulatively, simplifying the interface. With an energy provider that records total energy consumed since some time \( t = 0 \), a power-aware application can simply record energy \( E \) at the beginning and end of a time interval and compute the average power as \( \Delta E / \Delta t \). In this way, the interface easily supports applications interested in energy or power.

Next, a common interface can expose energy metrics in a standard and sufficiently precise unit of measurement at a large enough data length to avoid overflow. Applications therefore do not need to track various unit types and do conversions. A deceptively simple but important insight is not our choice of units or data length, but rather that counter overflow is one of the diverse sensor properties that applications should not have to worry about. We choose microjoules (\( \mu J \)) as 64-bit unsigned integers\(^2\), which are precise enough for the use cases we anticipate and avoid the need for floating point data types. In our experience, modern sensors are too imprecise and models too inaccurate to justify smaller units.

The **energymon** C interface defines a struct that contains seven function pointers and a state variable:

- `init`: Initialize the energy monitor.
- `fread`: Get the cumulative energy in microjoules.
- `finish`: Destroy the energy monitor.
- `fsource`: Get the name for the energy monitoring source.
- `finterval`: Get the refresh interval in microseconds.
- `fprecision`: Get the best possible precision in microjoules.
- `fexclusive`: Get if exclusive access is required.
- `state`: Pointer to a struct that the implementation uses to maintain internal state, e.g., file descriptors or thread data.

This interface exposes some of the diverse properties described previously, like refresh interval and sensor precision, while hiding other properties that are not important to most software, like the underlying interfaces, device protocols, and data formats.

A user typically calls `energymon_get_default` provided by the **energymon-default** interface to populate an `energymon` struct. If the implementation is known in advance, its getter function can be called directly. The default interface enables maximum portability since application code does not require any modification when changing data sources. Using the function pointers in the `energymon` struct provides portability for all other operations. The decision to use function pointers instead of explicitly declaring common header functions was made for practical reasons – in C, only one implementation of an interface can be implemented in a runtime. Reducing the number of common header functions minimizes the need for conditionally including extra wrapper functions at compile time (`energymon_get_default` is the only one). Therefore, implementations conditionally include the default getter function, allowing multiple EnergyMon implementations to be used simultaneously when there are multiple power/energy data sources known in advance.

EnergyMon implementations are initialized by calling the `finit` function pointer. If the implementation reads from a power sensor, `finit` starts a separate thread to poll the sensor at its refresh interval and update the cumulative energy value after each sensor reading. The application reads from the energy monitor by invoking `fread`. For any time \( \tau \), where \( \tau = 0 \) is the energy monitor’s initialization, the following invariants hold for energy readings \( E(\tau) \):

\[
\forall \tau \geq 0 \quad \tau_i \leq \tau_j \implies 0 \leq E(\tau_i) \leq E(\tau_j)
\]

\[
\tau = 0 \implies E(\tau) = 0
\]

Eqn. 1 states that for any time at or after initialization, energy is always non-negative and monotonically increasing. The only exception should be in case of a sensor read failure – in C, a 0 value is returned and `errno` set appropriately; abstractions in other languages may use their own error handling mechanisms, like exceptions. Eqn. 2 declares that the energy counter does not have to begin at 0 \( \mu J \). This flexibility facilitates sharing energy data with multiple applications and avoids the need to reset hardware counters, which is often a privileged operation. When the energy monitor is no longer needed, `finish` is called to cleanup resources like files.

\(^2\)At 100 W of power, it would take about 5,850 years to overflow.
or threads. The following example of the EnergyMon C interface demonstrates computing the average power of a worker function:

```c
energymon em;
uint64_t start_uj, end_uj, start_usec, end_usec;
// get the energymon instance and initialize
energymon_get_default(&em);
em.finish(&em);
// get start time/energy
start_usec = gettime_us(); // e.g. wrap clock_gettime
start_uj = em.read_uj(&em);
// application-specific processing
do_work();
// get end time/energy
end_usec = gettime_us();
end_uj = em.read_uj(&em);

printf("Average power of do_work() in Watts: \%f\n",
    (end_uj-start_uj) / (double) (end_usec-start_usec));
// destroy the instance
em.ffinish(&em);
```

Listing 1: EnergyMon C example.

Other functions provide additional information about the implementation that software can use to make runtime decisions. For example, applications can use the name provided by `fsource` for debugging or logging. `finterval` can be used to determine a lower bound on sleep time for polling the energy monitor at regular intervals. `fprecision` might inform a self-adaptive system how much noise to expect in power/energy data. `fexclusive` could be used in deciding to share energy data with other software components.

### 3.2 Implementations

We provide EnergyMon implementations for all the data sources detailed in Table 1, which access various power and energy sensors, both embedded and external. The table describes their differences in more detail. While not strictly a requirement, the interface is best leveraged when the underlying energy source makes data available to userspace. This allows sandboxed or other unprivileged applications to measure their own behavior, or even to adjust their behavior at runtime, e.g., to meet power cap requirements.

Some systems expose power or energy data for multiple sources. For example, the Intel MSR provides energy data for core and uncore components, as well as DRAM. Both the `msr` and `rapl` EnergyMon implementations use the MSR as their data source. Similarly, the ARM big.LITTLE system used in this paper exposes power data for the SOC’s big cluster, LITTLE cluster, GPU, and DRAM.

The `osp`, `osp-polling`, and `wattsup` implementations read from external sensors. The first two read from an ODROID Smart Power meter [32]—`osp` reads energy data in Watt-hours, accurate to three decimal places; `osp-polling` polls the sensor’s power readings in Watts, also accurate to three decimal places. The polling implementation provides more accurate energy data given that Watt-hours are so imprecise (1 Wh = 3600 J). The third implementation reads from a WattsUp? power meter, which provides power in deciwatts at coarse-grained one second intervals [42].

The `shmem` implementation reads from a shared memory location. Sensors that require exclusive access and/or elevated privileges are ideal candidates to run in separate processes and expose their data to multiple unprivileged applications over shared memory. This is especially true for external meters, so we include shared memory providers for the `osp-polling` and `wattsup` implementations. The `dummy` implementation is used when no other sources are available.

### 3.3 Bindings and Abstractions

Current EnergyMon implementations are written in C, but we also provide interfaces in Java and Rust. Additionally, we make bindings and abstractions over the `energymon-default` C interface available for those languages. Bindings provide as close to direct access to the underlying interfaces and implementations as possible. Abstractions provide the safety guarantees these languages demand and the RAI3 idioms they commonly use.

In Java, we provide the `EnergyMon interface`, which implementations written in pure Java may implement. Its methods throw exceptions in error cases. The `DefaultEnergyMon` abstraction manages a JNI boundary, encapsulates pointers to heap allocations, and cleans up automatically. Users may also safely destroy an instance themselves to immediately clean up resources, which might include background threads, if they do not wish to wait for garbage collection. For example, to use the default Java abstraction:

```java
// get the EnergyMon instance and initialize
EnergyMon em = DefaultEnergyMon.create();
// get start time/energy
long startMs = System.currentTimeMillis();
long startUj = em.readTotal();
// application-specific processing
doWork();
// get end time/energy
long endMs = System.currentTimeMillis();
long endUj = em.readTotal();
System.out.println("Average power of doWork() in Watts: ",
    (endUj-startUj) / (1000.0 * (endMs - startMs)));
// destroy the instance (optional, but recommended)
em.finish();
```

Listing 2: EnergyMon Java interface and abstraction.

In Rust, we provide the `EnergyMonitor trait` where functions that are allowed to fail return a `Result`. EnergyMon implementations written in Rust may implement this trait. The default `EnergyMon` abstraction encapsulates the strict bindings and prevents accidental or malicious destruction of an instance. For example, to use the default Rust abstraction:

```rust
// get the EnergyMon instance and initialize
// EnergyMon implements the EnergyMonitor trait
let em: EnergyMon = EnergyMon::new().unwrap();
// get start time/energy
let start_time: SystemTime = SystemTime::now();
let start_uj: u64 = em.read_uj().unwrap();
// application-specific processing
do_work();
// get end time/energy
let end_time: SystemTime = SystemTime::now();
let end_uj: u64 = em.read_uj().unwrap();
let duration: Duration = end_time.duration_since( start_time).unwrap();
println!("Average power of do_work() in Watts: {:.2f}",
    (end_uj - start_uj) as f64 /
    (duration.as_secs() * 1000000 +
    duration.subsec_nanos() as u64 / 1000) as f64);
// instance dropped automatically
// (when it falls out of scope or is consumed)
```

Listing 3: EnergyMon Rust trait and abstraction.

---

3Resource Acquisition is Initialization
4. EXPERIMENTAL DESIGN

4.1 Evaluation Platforms

We use three very different hardware systems for our analysis. The first is a Lenovo Thinkpad X1 Carbon laptop (3rd gen) with an Intel i7-5600U dual-core processor with Hyperthreading [11], running Ubuntu 15.10 with Linux kernel 4.2.0. The second is a Hardkernel ODROID-XU3 mobile development platform with an Exynos5 Octa 5422 SoC with quad core ARM Cortex-A15 “big” and Cortex-A7 “LITTLE” clusters [38]. The ODROID runs either Ubuntu 14.04 LTS with Hardkernel’s modified Linux kernel 3.10.58+ or Android 4.4 “KitKat”, depending on the case study. The third system is a Sony VAIO SVT11226CXB tablet with a dual-core Intel Haswell mobile processor with Hyperthreading [35]. It runs Ubuntu 14.04 LTS with Linux kernel 3.13.0. Prior work has shown that the Haswell processor and the Exynos big.LITTLE SoC exhibit contrasting performance/energy behavior [18].

4.2 Case Studies

Evaluating and exploiting performance and power tradeoffs are common research topics, particularly in the embedded and mobile systems domain. Our first case study demonstrates diversity in both application and system performance/power tradeoffs, motivating the need for energy awareness in applications. We capture coarse-grained timing and energy behavior for two distinct, unmodifying applications using the Vaio and the ODROID, both running Linux. It is common for engineers to use external instrumentation when evaluating software behavior on different systems, so we use a WattsUp? PRO meter with the wattsup EnergyMon implementation on the Vaio, and an ODROID Smart Power meter with the osp-polling implementation on the ODROID. Since video encoding is a common task for embedded systems like mobile phones and tablets that have on-board cameras, we use a compute-bound parallel video encoder for the first application [4]. We use the STREAM benchmark as the second example to represent memory-intensive applications [28].

The second case study profiles Servo, Mozilla’s parallel web browsing engine written in Rust [3, 27], at a fine-grained level. This experiment uses the Thinkpad and the ODROID. The Thinkpad runs Linux and uses the msr EnergyMon implementation to read from the Intel MSR [35]. The ODROID now runs Android and uses the adroid-ioctl implementation to poll embedded TI INA-231 power sensors [21]. We use a real, industrial-strength application to demonstrate that EnergyMon provides in-situ runtime energy data on different hardware and software platforms suitable for profiling in a practical setting.

Our third case study returns to the parallel video encoding application used in the first experiment [4], but now uses a version that was previously modified to configure resource allocation at runtime to obey soft power constraints. Again we use the Vaio tablet and the ODROID, both running Linux. The modified application’s feedback controller was originally designed for the systems’ embedded sensors, but we substitute the energymon-default interface and use external power meters. This provides an additional challenge and further demonstrates EnergyMon’s utility and portability. We once again connect the Vaio to an external WattsUp? PRO power meter [42] and the ODROID to an external ODROID Smart Power meter [32]. Because these meters require elevated privileges and exclusive access, we now launch the wattsup and osp-polling EnergyMon implementations in a separate process and expose energy data to the application using shared memory. This makes the energy data available to any application that wants it, which we access with the shmem EnergyMon implementation during application runtime. This experiment not only demonstrates EnergyMon’s potential for self-adaptive applications, but demonstrates its flexibility and portability by using sensors the self-adapting controller was not originally designed for.

In summary, our case studies use Linux and Android and run on three different hardware platforms – a tablet with an Intel Haswell processor, an ARM big.LITTLE development platform, and a laptop with an Intel i7 processor. We use four distinct power/energy data sources – the Intel MSR, embedded TI INA-231 power sensors, an external WattsUp? PRO meter, and an external ODROID Smart Power meter.

5. EXPERIMENTAL EVALUATION

This section presents our analysis of EnergyMon with case studies that motivate the importance of energy awareness in software and demonstrate EnergyMon’s utility. We also analyze EnergyMon’s runtime overhead, then discuss limitations and future work.

5.1 Performance/Power Tradeoffs

Our first case study demonstrates diversity in performance/power tradeoffs for systems and applications, motivating the importance of energy awareness in software. To properly manage system resource allocation, engineers must understand their application’s behavior, even if only at a coarse-grained level. This is especially true for applications that run on energy-constrained platforms like embedded and mobile systems. Software exhibits different performance/power tradeoffs depending on the system it is executed on, and behavior can even vary dramatically for different applications running on the same system. Performance/power tradeoffs are a function of both the application and the system. We first demonstrate tradeoff diversity between systems with a parallel video encoding application [4]. We then show diversity across applications on the same system by comparing these results with those produced by the STREAM benchmark, which represents memory-intensive applications [28].

We connect the tablet with the Intel Haswell mobile processor and the ARM big.LITTLE development system to external WattsUp? PRO and ODROID Smart Power meters, respectively. We allocate unique combinations of DVFS frequency and core assignment – 44 configurations for the Intel Haswell and 128 configurations for the ARM big.LITTLE. We record the total application runtime and system energy consumption for each configuration, which we use to compute normalized average performance and power values.

Figure 1 is the resulting tradeoff space for the video encoder, normalized to the maximum performance and power achieved on each system. System idle power is included for reference. The Intel Haswell has a mostly linear tradeoff space, with maximum perfor-
5.2 Servo

This case study uses Servo, Mozilla’s parallel web browsing engine. We add energy monitoring to Servo’s existing fine-grained time profiling interface and add a background profiler called ApplicationHeartbeat to record runtime power behavior. We profile Servo on the Thinkpad with an Intel i7 processor running Linux, and the ODROID with an ARM big.LITTLE SoC running Android. We capture energy readings for each platform’s processors rather than using external equipment. The experiment loads Servo, fetches the page mozilla.org from the remote server, processes the page, and displays it. We instruct Servo to use four layout threads on each evaluation system (the Intel i7 has four virtual cores, and each of the ARM big.LITTLE clusters has four cores).

Figures 3a and 3b break down by profiler category where time and energy are spent within Servo while loading on the two platforms, averaged over four trials. Profiler category names are trimmed to save space, and unused categories are not shown. Engineers use timing and energy data to decide where to focus their optimization efforts. Developers will recognize that tasks running in parallel, like PaintingPerTile, record their own elapsed time which results in total time values that exceed actual runtime. Energy is measured for shared system components, so some energy consumption is assigned to multiple profiler categories and is therefore counted more than once (the sum of the energy values exceeds the actual energy consumed). More advanced profiling and analysis techniques beyond the scope of this paper are needed to better assign time and energy consumption to parallel tasks. ApplicationHeartbeat provides the true wall clock time and total energy consumption for the execution. Layout operations require a significant portion of the total time and energy on the Intel i7 in Linux, but painting consumes a larger percentage of the runtime on the ARM big.LITTLE in Android. This means developers need to concentrate their efforts in different places depending on the system they are trying to optimize for.

Figures 4a and 4b present the time series for a single execution, overlayed with system power derived from ApplicationHeartbeat’s energy readings. The ApplicationHeartbeat polling interval is limited by the refresh rate of the sensors underlying the EnergyMon implementation, as exposed by the finterval function, hence the more coarse-grained power data on the ARM big.LITTLE system (its power sensors refresh about 4x per second). The Intel i7’s MSR refreshes every 1 ms, but we limit the profiler’s polling interval to 50 ms to avoid unnecessary overhead.

The LayoutPerform profiler category is an umbrella event for all layout operations. LayoutStyleRecalc determines which CSS styles...
apply to each of the nodes in the Document Object Model (DOM) tree and creates a structure that layout will use to determine element positioning. This is a time-consuming operation given that it must traverse the DOM tree. LayoutMain then determines where to position elements when rendering on the screen. The time series makes it clear that layout tasks, especially LayoutStyleRecalc, consume more power than average on the Intel i7. The power data captured on the ARM big.LITTLE in Android is more coarse-grained, but accounting for the delays also indicates the same trend. On ARM, there is a large power spike about halfway through the execution – something we saw in all of our trials. The current profiling does not point to an obvious reason for this, indicating that the application developers should investigate further. More importantly, this behavior would not be detected by time profiling alone, yet is critical on power and energy-constrained systems.

Despite the differences in the two platforms, we are able to capture useful profiling energy data from both using the common EnergyMon interface. This energy profiling capability is now publicly available as part of Servo via the energy-profiling feature.

5.3 Self-Adaptive Application

This experiment uses an energy-aware parallel video encoding application that adjusts its resource usage at runtime to meet soft power constraints [19]. Given a power target and set of configurable system resources, it adapts to meet the power target while maximizing performance. We substitute EnergyMon for the controller’s custom energy monitoring code. To further challenge EnergyMon and demonstrate its versatility, we again use external instruments instead of the platforms’ embedded sensors that the feedback controller originally intended to use. The experiment is par-

Figure 4: Timing and power behavior while loading mozilla.org in Servo.
ticularly challenging because of the relatively slow refresh rate, coarse-grained accuracy, and additional I/O overhead of communicating with the devices over USB. We select a window period of 50 frames—the work interval at which the application adjusts its resource usage. For each system, we set the power target to be the mean of the minimum and maximum processing power. We then launch the application with an input that is the combination of three videos with distinct levels of encoding difficulty.

Figure 5 is a time series of the resulting behavior. The top portion of the figure is the window performance (frames encoded per second), normalized to highest for each system; the bottom portion shows the window power, normalized to the target. The vertical lines at frames 500 and 1000 indicate the boundary between the input video’s phases. The system power hovers around the target as the application adjusts resource usage while the performance fluctuates depending on the level of encoding difficulty for each frame.

Maintaining the desired power on the Intel Haswell is more difficult than on the ARM big.LITTLE. The slower refresh rate and lower precision of the WattsUp? power meter, compounded by the Haswell’s faster processing speed and the variable difficulty of encoding the input video (even within a single phase) make the task especially challenging. During the first window period (50 frames), the application observes system behavior, then makes its first resource allocation adjustment. For frames after the first period of resource adjustment (frame 100), the mean window power per frame is 0.14% above the target on the Intel Haswell and 0.027% below the target on the ARM big.LITTLE, with 4.4% and 2.0% coefficient of variation, respectively. Of these frames, 44.6% on the Haswell and 43.4% on the big.LITTLE had an average window power above the target. This is expected when meeting soft power targets for an application that exhibits as much variability as a video encoder [20]. The application could be tuned further to keep more frames below the power target, but that is beyond the scope of this work. What is important for this case study is that EnergyMon successfully provided the necessary energy data to the application’s feedback controller. Despite the challenges, EnergyMon’s portable interface allows the application to use data from external power meters it was not originally designed for to meet its power constraints.

5.4 Overhead Analysis

Runtime overhead is an important consideration when collecting power/energy data, whether it be for profiling, a feedback control system, or something else. We took simple in-situ timing measurements for the finit, fread, and ffinish functions on current EnergyMon implementations and averaged the results of 100 trials. We have not attempted to optimize our implementations’ performance beyond the obvious design decisions required to achieve accurate results, like caching file descriptors as part of the state variable (see Section 3.1).

Figure 6 shows the average latency for these three functions—note the log scale. The odroid, odroid-ioctl, osp, and osp-polling implementations were tested on the ODROID, the rest on the Thinkpad. Initialization and destruction of energy monitors are the most time-consuming tasks (finit and ffinish), but only need to be done once. Implementations that read from external devices must perform I/O communications with the devices and may need to wait while they reset internally. For example, the wattsap implementation must sleep for a whole second after clearing the device memory before it starts capturing, so it took an average of about 1.5 seconds to completely initialize. On the other hand, the msr and rapl implementations initialized in about 50 µs. The odroid implementation took 1.6 µs since it scans the sysfs filesystem for the sensors; odroid-ioctl took 384 µs to open the four device files and enable the sensors. If the overhead of finit and ffinish truly matter to an application, the energy monitor can be run in a background process and expose data using shared memory. Applications can then read energy data using the shmem implementation, like in Section 5.3, which initialized and finished in 9.0 µs and 2.3 µs, respectively.

The fread function provides the core functionality and is naturally executed the most. The msr implementation took less than 1 µs on average and rapl took 15 µs. The osp implementation reads from an external ODROID Smart Power meter over USB, which caused relatively high latency—about 12 ms. However, the refresh interval of the device is 200 ms, so the implementation is not very useful for applications that need to poll more frequently. Implementations that read from power sensors have background threads to poll the sensors, and account for the I/O and runtime latency when computing energy from power. In these cases, reading the energy data from an application only involves accessing 8 bytes of memory. Their response times were typically around 1 µs, at which point the precision, refresh rate, and overhead of reading the system clock affects the accuracy of the measurement.

5.5 Limitations and Future Work

As with any power/energy monitoring, whether performed in situ or externally, results are limited by the precision, accuracy, and refresh rate of the sensors. A sensor that only updates once per second may not be as useful as one that updates more frequently. In cases of relatively slow refresh intervals, tasks sometimes complete too quickly to capture an accurate change in energy values. Most of the time no change in energy will be recorded, but occasionally a task execution will overlap with a sensor refresh and record an inaccurately large change in energy for a short-lived task. With a statistically large enough sampling, the total energy recorded for relatively short-lived tasks will average to reasonably accurate values. Servo’s PaintingPerTile profiler category in Figures 3 and 4 is a good example of this behavior. The most informative results
we observed were captured using the Intel MSR, which refreshes at millisecond intervals.

Some power/energy sensors require exclusive access, particularly external meters. If multiple parties are interested in using the sensor simultaneously, a reader can run in a separate process and expose the data over shared memory or other IPC mechanisms. For example, we wrote shared memory providers for the osp-polling and wattsup EnergyMon implementations. Applications then just link with the shmem implementation.

Automatic discovery of power/energy sensors is an important next step for hardware and operating systems designers to address. A standard operating system interface would offer greater flexibility to power and energy-aware applications. Advances in operating system support would not eliminate the need for a portable application interface like EnergyMon, but rather expand its utility.

Unfortunately, not all systems have power/energy sensors or are connected to external meters. EnergyMon implementations can instead use models based on other hardware counters or sensors to provide energy estimates. There is prior work on building power/energy models based on other hardware counters, but is beyond the scope of this work [8, 26, 34, 39, 46]. However, future work includes building implementations that can interpret models in a common format, perhaps using tools like PAPI [5] to read performance counters as needed to drive the model.

6. CONCLUSION

This paper describes the challenges arising from the diversity in power and energy data sources, motivating the need for a common interface to expose energy data to software. In response, we propose EnergyMon, a portable interface that is independent of underlying power/energy sensors. We demonstrate the interface’s flexibility and practicality with three use cases. We first show diversity in performance/power tradeoffs across platforms, and for applications running on the same platform, motivating the need for energy awareness in software. Next, we instrument Servo, Mozilla’s parallel web browsing engine, with energy profiling and describe the insight it provides. This feature is now publicly available in Servo. We then substitute EnergyMon for an energy-aware video encoder’s custom energy monitoring code and show that it continues to meet imposed power constraints, even using different sensors than it was originally designed for. Our experiments use both internal and external power and energy sensors with widely varying properties on platforms with different, sometimes contrasting, performance and energy characteristics. We release the EnergyMon interface, implementations, Java and Rust bindings, and other utilities as open source.

References