TET: A Proof-of-Concept Tock Energy Tracker

Bili Dong*
b2dong@ucsd.edu
UC San Diego
USA

John Hwang*
jfhwang@ucsd.edu
UC San Diego
USA

Abstract

We present TET (Tock Energy Tracker), a proof-of-concept energy tracker for the Tock operating system, that monitors embedded system energy consumption both at hardware component level and at application level. TET works even when there are concurrently running applications and contention in usage of hardware components. We demonstrate that our prototype implementation works to some extent, and evaluate its performance on a real hardware board. Although our current implementation is not quite production-ready, we hope to provide a useful reference for future exploration.

1 Introduction

In the field of embedded systems, energy management has long been a important issue. Unlike traditional computers, the space of embedded systems is encompassed by devices that are often not attached to power sources and have to run off of finite amount of energy such as with batteries, or run on harvesters that can reap only limited amounts of energy. In order for such devices to function with minimal maintenance and downtime, there arises a need to keep them energy efficient and capable of running for long periods with limited energy.

One of many ways to achieve this is through energy efficient applications. To facilitate this type of development, developers must have some way of evaluating the energy requirements of their applications. With the recent increase of multi-tenancy in embedded systems, this process becomes complex as users must disentangle or isolate the costs of multiple applications running on the same platform. Additionally, obtaining energy measurements can often be unwieldy and require the use of digital oscilloscopes and other testing hardware if software support is not provided.

In our work, we develop a software tool to provide an estimate of such values for the Tock OS, a relatively new but popular embedded operating system which supports multi-tenancy. As a first step towards an efficient energy management subsystem in multi-tenant embedded operating systems, we first need to have a system to attribute power consumption to different concurrently running applications. This application power attribution system would also be useful on its own, as a performance monitoring tool, which could be utilized to inform system designers and application developers to improve the performance of the system or application.

For the purpose of better energy monitoring, several power tracking systems have been developed for embedded OSes[2, 4] in the past. Although different in their overall designs, they share the same underlying technique called *power state tracking*. The main idea of power state tracking is to model the power consumption of the whole system as a linear combination of its components. Each component could be operating in different power states. Corresponding device drivers need to be modified to track the power state information of each component. Eventually the power consumption of each component and the whole system can be estimated using a power model.

Tock[5][1] is a more recent embedded operating system featuring concurrently running mutually distrustful applications. Being a relatively new embedded OS, it currently does not have any accompanying power tracking tools. In this course project, we aim to implement a prototype energy tracking tool for Tock, based on the aforementioned power state tracking technique, that can monitor energy consumption of individual components and attribute them to concurrently running applications reasonably well.

2 Related Work

2.1 Previous Energy Tracking Systems

Energy tracking systems have been developed for popular embedded OSes prior to Tock. The most notable ones are Quanto[4] for TinyOS[2], and Powertrace[2] for Contiki[3]. They share the same underlying technique called *power state tracking*, but differ in many other ways.

Quanto tracks component power state and activity attribution at run-time, and it also requires the special iCount energy meter[3] to measure the total energy consumption continuously. This information is logged at run-time, and analyzed off-line later to estimate the energy consumption per component and per activity. *Activity* is a high level abstraction Quanto uses to group together any logically related operations to attribute energy consumption to.

Powertrace, unlike Quanto, is a purely software-based system. It provides power-profiling data continuously at runtime. Powertrace uses an abstraction called *energy capsule* for energy attribution. As far as we read from the papers, it's

*Both authors contributed equally to this research.

[1] https://www.tockos.org/
[2] http://www.tinyos.net/
[3] https://www.contiki-os.org/

pretty similar to the activity abstraction in Quanto, at least as a high-level concept.

TET uses the same power state tracking technique (details to be found in Section 3.2) as both Quanto and Powertrace. At the lower level of power state tracking, we are closer to Powertrace than Quanto in that we continuously process power state updates do not rely on a special hardware-based energy measurement device at run-time. At the higher energy attribution level, we do not develop any generalized energy attribution abstraction like "activity" in Quanto, or "energy capsule" in Powertrace. Instead, we simply attribute energy to applications/processes. This is mainly for the simplification of implementation. More discussion could be found in Section 6.3.

2.2 Energy Accounting Policy

In contrast to TinyOS and Contiki, on which Powertrace and Quanto are built respectively, Tock is an operating system that supports multi-tenancy. This means TET has to handle tracking energy in the presence of contention over peripherals. In Tock, there are no existing abstractions to manage contention over control of the LEDs. Instead, processes are free to modify LED state without consideration for other processes. This makes the problem that we face different from the virtual battery paper [1] where they prioritize maintaining higher power states when two processes attempt to set conflicting states. In contrast, we build our tool on a system that prioritizes the most recent state.

Another factor that accounts for differences in our designs is our differing objectives. Unlike the virtual battery paper [1] which has the objective of balancing and dividing energy usage, the objective of our design is to provide users with accurate energy profiles.

3 System Design

TET is designed to track energy both at hardware component level and at application level. The hardware component level energy tracking is necessary to provide a faithful view of the actual energy consumption. The application level energy tracking only provides a virtualized view of energy consumption. Each application's energy consumption is accounted as if it is the only application running. This might lead to a situation where the same piece of component energy consumption could be attributed to multiple applications at once.

3.1 Power Model

The basic assumptions we make about hardware power consumption are:

- The total energy consumption of the board is a linear combination of its individual components.
- Each component can operate in several different power states, corresponding to a different but constant power consumption.

The assumptions can be summarized mathematically as:

$$P_{\text{system}}(t) = \sum_{c,s} P_{c,s}\alpha_{c,s}(t) \tag{1}$$

Here $P_{\text{system}}(t)$ is the total power consumption of the system at time t, $P_{c,s}$ is the power consumption of component c in power state s, $\alpha_{c,s}$ is a single bool bit to indicate whether or not component c is in power state s at time t (1 for yes, and 0 for no). Integrating this over time and we get the expression for the total energy consumption:

$$E_{\text{system}} = \sum_{c,s} P_{c,s}T_{c,s} \tag{2}$$

Here $T_{c,s}$ is the total time component c spends in power state s. The power coefficients $P_{c,s}$ comprise the *power model* we need for converting time tracking into energy accounting.

3.2 Component Power State Tracking

As we could see from Equation 2, to track the component energy consumption, essentially we just need to track the time the component spends in different power states. This technique is called *power state tracking*, which are widely used in previous energy tracking systems[2, 4]. To track time for different component power states, we need to instrument corresponding device drivers to signal any power state changes. In this work, we focus on the LED component, which simply has two power states: on and off. And the instrumentation is easy. The implementation details are described in Section 4.2.

3.3 App Energy Attribution and Accounting

Although power state tracking helps us account energy spent per component, it doesn't help us account energy spent per application. For the latter, we also need to know which application causes the time spent by a component in a power state, as could be expressed by:

$$E_a = \sum_{c,s} P_{c,s}T_{c,s,a} \tag{3}$$

Here E_a is the total energy consumption of application a, $T_{c,s,a}$ is the total time component c is in power state s as *caused by application a*.

But how do we know which application causes which component power state? We achieved this by intercepting system calls. In Tock, a system call always pass along a process ID.

The need for an application-specific accounting policy arises when multiple processes contend for control over the same component. Due to the nature of the system we build on, the state of a component is based on the most recent state modification. As such, it becomes possible for processes to interfere with one another's operations. For example,

- App1 turns LED ON at time T1.
- App2 turns LED OFF at time T2.
- App1 turns LED OFF at time T3.

Due to App2's interference, App1 had the LED turned off at T2 instead of the expected T3. In this situation, when App2 modifies the LED state at T2, we add App1's energy costs from T1 to T2 to App1's profile, and set App1's LED state to None. The None state has no energy accounting and is not modified until the process issues another state change.

This implementation is sufficient for most cases. However, there is one case where further improvements are possible, and that is when multiple processes request the same state on the same peripheral. For example, let's consider a board with a single LED, two applications running concurrently on it, and the following event sequence:

- App1 turns LED ON at time T1.
- App2 turns LED ON at time T2.
- App2 turns LED OFF at time T3.
- App1 turns LED OFF at time T4.

It's assumed that $T1 < T2 < T3 < T4$. Then the following are the LED's power state changes and what would happen in the energy tracker:

- T1: LED enters power state ON, as caused by App1.
- T2: LED stays in power state ON, as caused by App1 *and App2*.
- T3: LED enters power state OFF, attribute energy to App1 and App2.
- T4: LED stays in power state OFF, do nothing.

At T2, there is ambiguity in how to attribute the energy costs of the LED. Which application causes the LED power state ON? App1 or App2? Our policy says both, because both App1 and App2 request the LED to be ON at this moment, so logically they are both the causes for the corresponding power consumption. Following this logic, we add the full energy costs of utilizing the LED to the energy profiles of both App1 and App2. At T3, energy spent from T1 to T3 are attributed to App1, and energy spent from T2 to T3 are attributed to App2. Note again that the energy spent from T2 to T3 are accounted for twice.

4 Implementation

Our TET implementation is publicly available on GitHub[4].

4.1 Energy Tracker Capsule

In the Tock architecture, the kernel is divided into two categories. First is a small trusted **core kernel** which implements essential functionalities such as hardware abstraction layer (HAL), the scheduler, and platform-specific configurations. The less trusted and less essential software components are introduced into the kernel as **capsules**. The core kernel is protected from any possibly malicious capsules via the Rust language's safety guarantees. Meanwhile, capsules can freely interact with one another.

The majority of TET's software modification came in the form of a new Tock capsule, "energy_tracker". Although energy tracker is not essential enough to be part of the core kernel, we also need our tool to be able to intercept any desired communication between processes and driver capsules. Thus, a capsule is the most appropriate place to add the core functionality of our implementation. Within the energy tracker capsule, we maintain two groups of information: ones used for per component energy tracking, and ones used for per application energy tracking.

For per component energy tracking, we maintain a global static table that is proportional to the number of components. For each component, we record: the total energy it has consumed, the current power state it is in, and the start time for this power state.

For per application energy tracking, we maintain the same type of tables previously discussed on a per process basis. We need these additional tables to maintain a virtualized view of energy consumption for each application and each process' power state table will likely differ.

One thing to note is that capsules are not allowed to allocate dynamic memory since kernel memory is very limited and a capsule with poor memory management could potentially exhaust it. Since the number of processes is dynamic, we have to allocate process-specific tables inside **grants**. A grant in Tock is a section of process memory which is reserved for the kernel's usage. As such it scales with the number of processes, which is exactly the kind of memory we need to maintain per application states.

4.2 Device Driver Instrumentation

In addition to adding a capsule, for our tool to track component power states, we need to instrument the device driver to notify TET whenever its power state changes. The notification interface is described by a simple function:

```
fn set_power_state(app_id, component_id, state);
```

This function basically says that the specified app requests the specified component to enter a specified power state. Using these information, TET can update its internal states to keep track of energy consumption per component and per app. To instrument the device drivers, we just need to insert this function whenever the device's power state is supposed to change.

4.3 Debugging Interface

For the convenience of interacting with the energy tracker, we added a debugging interface. We build our debugging interface as modification of the existing process console tool which was already present in Tock. After connecting the board to a computer using the debugging USB port, a user can use the process console to query or control processes on

[4] https://github.com/JFHwang/tock/tree/energy-tracker

the board like a shell. We expanded the process console tool so the user can query and control the energy tracker too.

5 Evaluation

We implemented TET for the nRF52840 DK board and evaluated it on the same hardware. As we only implemented power state tracking for the LED component, our evaluations focus on the 4 LEDs on this board. The raw data for these experiments, and the processing code are publicly available on GitHub[5] for anyone interested to reproduce.

5.1 Power Model

A full power model would consider power consumption of all components on the board. Limited by our current implementation of TET, we use a simplified power model where everything other than the LED are considered a background. Since the applications used for evaluation are all simple LED toggling applications, we think it is reasonable to assume the background power consumption as constant. We can express this simplified power model as:

$$P_{\text{board}}(t) = \sum_i P_{\text{LED}_i} \alpha_{\text{LED}_i}(t) + P_{\text{background}} \tag{4}$$

5.2 Power Measurement Setup

Some of the following experiments require measuring the total power consumption of the board accurately. So we set up a hardware-based measurement as shown in Figure 1, 2.

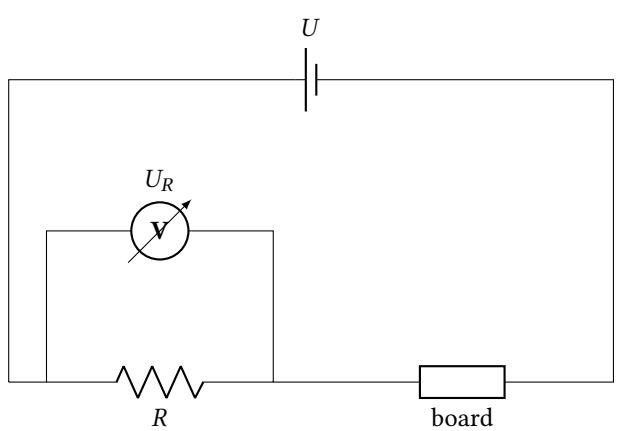

Figure 1. Circuit diagram for the power measurement setup.

The board is connected in series with a resistor R. The whole circuit is powered by a power supply providing a constant voltage U. A logic analyzer is used to measure the voltage U_R across the resistor R. Eventually we can calculate the total power consumption of the board as:

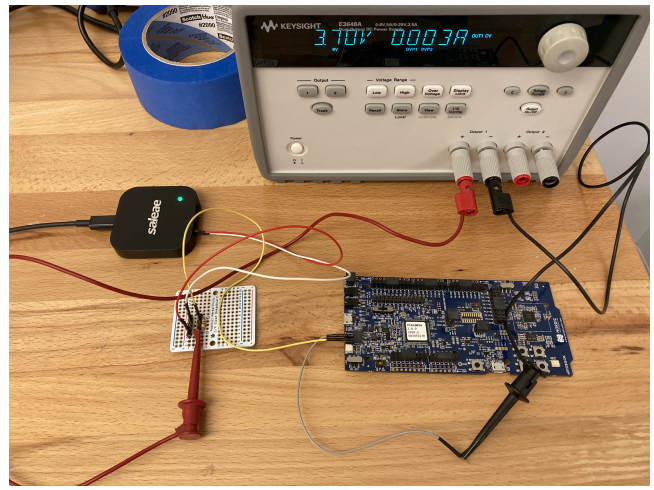

Figure 2. A photo of the actual power measurement setup.

$$P_{\text{board}} = \frac{U_R}{R} \times (U - U_R) \tag{5}$$

In our setup, $U = 3.7V$, $R = 46.87\Omega$, $U_R < 0.7V$ (U_R reaches its maximum when all 4 LEDs are on at the same time). The resistor is chosen such that $U - U_R > 3V$ always holds, where $3V$ is the minimum voltage the board requires. The logic analyzer is operating with a sampling rate of 250Hz, which is good enough for our applications that toggle the LEDs at most every hundreds of milliseconds.

5.3 Calibration

To use the power model as expressed in Equation 4, we first need to determine the constants P_{LED_i} and $P_{\text{background}}$. This is done in a calibration step. We run an application that take turns toggling each LED on and off every few seconds. In the end, all 4 LEDs are turned on simultaneously. The power measurement result is shown in Figure 3.

The background power is measured to be $P_{\text{background}} = 11.18\text{mW}$. When individual LEDs are on, each consumes approximately the same amount of power, and is measured to be $P_{\text{LED}} = 12.19\text{mW}$. When all 4 LEDs are on simultaneously, the power is measured to be $P_{\text{4LED}} = 32.73\text{mW}$. We notice that $4 \times P_{\text{LED}} \neq P_{\text{4LED}}$, which basically says that our linear power model assumption does not hold. This is a strange result for us, because previous research with experiments on LEDs did see a pretty good linear power model[4]. We suspect this has something to do with our power measurement setup, but we do not have any concrete explanation yet.

5.4 Accuracy

Next, we evaluate the accuracy of TET by comparing the power profiles of the same applications as measured by our power measurement setup, and as estimated by TET. We run 2 applications concurrently with contended uses of one

[5] https://github.com/qobilidop/tock-energy-tracker

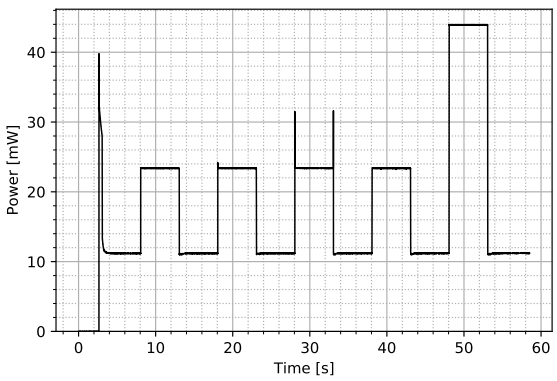

Figure 3. Calibration measurement. The first peak corresponds to switching on. The next 4 peaks correspond to the 4 LEDs individually on. The last higher peak correspond to all 4 LEDs on at the same time.

LED, to demonstrate some non-trivial energy accounting cases at the same time. Given that the linear power model assumption does not hold in our setup, we do not expect to see the power profiles match in values, but we do expect the 2 profiles match in trends. The results are shown in Figure 4.

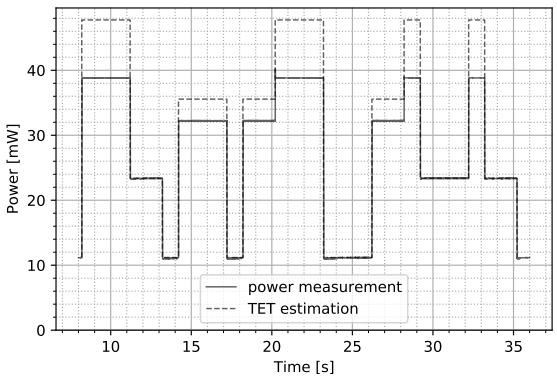

Figure 4. Power profile comparison between power measurement and TET estimation.

As we expected, the power profile trends do match between our software estimation and the hardware measurement, which means TET could track correct component power states with relatively accurate timing, even when there are concurrently running applications and contended uses of the same component.

5.5 Overhead

Lastly, we evaluate the overhead of TET in terms of memory, running time, and energy. The evaluation is done by comparing those performance metrics with the same application running on the baseline Tock implementation (referred to as tock-base), and our modified Tock with TET (referred to as tock-tet). In order to have enough space for TET, we removed the BLE radio capsule from the board definition in tock-tet. So for a fairer comparison, tock-base is original Tock with the BLE radio capsule removed too. For easier timing, we only run a single application on the board. The application we use is the classic blink app. The overhead measurement results are summarized in Table 1.

Table 1. Overhead measurement results.

	tock-base	tock-tet	Overhead	Relative
Memory [B]	151553	180225	28672	19%
Time [s]	19.5656	19.5829	0.0173	0.09%
Energy [mJ]	609.84	611.39	1.55	0.25%

The memory overhead is significant. We interpret this as our implementation needs some optimization. We think the design itself should not consume this much memory. Since both authors are new to Rust and Tock development, it is very likely this significant memory overhead is due to some poor programming practice. This optimization is left for future work.

The running time and energy overheads are ignore-able, which is good. But we have to point out that care must be taken in interpreting these numbers. Since the TET service is triggered only when there is power state change event, the run-time performance overhead shall scale with the frequency of these events. Our experiment application toggles 1 to 4 LEDs once every 250 ms, which must be taken into consideration. A more thorough experimental examination on this scaling is left for future work.

6 Discussion

6.1 Design Tradeoffs

Energy Accounting Policies. For our application-specific energy accounting policy in the case of multiple requests for the same state, we considered 3 different designs.

- Maintain the previous policy and pre-emptively end the earlier process' energy charges. This is treating the same-state case like other cases.
- Charge every process using the LED for the full amount. This is the implementation we selected and the implementation in the Virtual Battery paper[1].
- Split the usage costs across contending processes evenly.

The first and third approaches have the benefit of creating application energy profiles that add up to the total energy cost and can be beneficial if the user is attempting to profile the energy consumption of the entire system. Meanwhile, the second policy often results in situations where the sum of application-specific energy costs will exceed the actual total energy cost.

The benefit of the second policy is that it ensures more stable application-specific energy profiles in the presence of increased contention. This approach bases the energy profile off of how much utilization of the peripheral achieved was achieved. Meanwhile, the other policies don't maintain this relation between utilization and energy costs. In fact, this relationship become increasing distorted with additional contention which could lead to misleading energy profiles at an application level. In the end, we decided on using the second policy for this reason. However, in order to supplement its weakness of inaccurate total energy estimates, we introduce a peripheral-specific profiling system which captures actual energy costs of each peripheral.

6.2 Limitations

Even given the improvements we could make to make the implementation more production-ready, there are some fundamental limitations to our system design.

Linear power model. A linear power model is a basic assumption we made, but we know in reality this is only an approximation. When this assumption breaks, our software-based estimation could be far off. A more accurate approach would be combining a software system like ours with a hardware current meter to calibrate the power model at run-time. However, this might bring in other issues which are beyond the scope of this discussion.

System modification. Our system requires instrumentation of system code like device drivers to work. Although the changes are not extensive, they are not as minimal as similar tools seen in other embedded OS's due to Tock being written in Rust.

6.3 Future Work

Our current implementation are only proof-of-concept. There are much more to be done to make this system production-ready. We have considered some of those, but haven't got the time to flesh them out. So we list them here as future work.

Power state tracking for more components. In this work, we have only implemented power state tracking for LEDs. LEDs' power states are simply ON or OFF, but there are more complicated components like radios or the CPU that have different modes of operating corresponding to different power states. Implementing power state tracking requires more careful instrumentation of the corresponding device drivers to thoroughly and correctly track any power state changes.

Hardware interrupts handling. Tock kernel scheduler is event-driven and has two sources of events: hardware interrupts and process system calls. In our current design, only process system calls are handled. This is sufficient for the simple LED component, and simplifies the app attribution process, because each process system call includes the caller process' ID. But for more complicated components, hardware interrupts might be involved in its operation, and we need to come up with a method to attribute these interrupts to applications properly.

Userspace interface. It remains a question whether we want to provider an interface to userland apps to query and control the energy tracker. And if we do, how? The major concern here is security, as individual app's energy consumption should be regarded as private to that app to avoid side-channel attacks. Maybe we could provide an interface to let an app query its own energy consumption alone. It requires further research to get any concrete conclusion.

Device driver virtualization. We have adopted a virtualized energy accounting policy for applications, but the LED device drivers themselves are not virtualized (no proper handling of concurrent use of the same LED). It would give our system much more design flexibility if both the energy accounting policy and the device driver are virtualized like in the Virtual Battery paper[1].

Energy attribution abstraction. In this work, we simply attribute energy consumption to processes. There might be more natural units of resource attribution for certain cases. For example, we might be interested in studying the energy consumption of a certain network protocol. So it's worth extending the notion of resource attribution unit from a process to a more general notion of *activity*. These are explored in previous works[2, 4], and might be worth reproducing in our system too.

7 Conclusion

This paper presents our proof of concept implementation of an energy tracker in Tock, aptly named TET (Tock Energy Tracker). We show that such a tool is very much possible to implement in the Tock architecture. In our exploration of this project, we have found the majority of the complexity of the project originated from picking up many new tools and interfaces as well as learning Rust and Tock's software architecture. In its current state, our implementation have set up sufficient infrastructure to facilitate instrumenting additional peripherals which should not be too difficult given the proper identification of power state transitions. We hope this report could serve as a useful reference for future researchers or developers aiming to implement a production-ready energy tracker for Tock.

Acknowledgments

To Pat Pannuto, for being a great instructor introducing us to the wonderful world of embedded systems and IoT. To Nishant Bhaskar, for being a fantastic TA and helping us set up all the hardware-based measurements. To our classmates, for

the various discussions we had. To the anonymous reviewers of our draft, for giving us valuable suggestions.

References

[1] Qing Cao, Debessay Fesehaye, Nam Pham, Yusuf Sarwar, and Tarek Abdelzaher. 2008. Virtual Battery: An Energy Reserve Abstraction for Embedded Sensor Networks. In *2008 Real-Time Systems Symposium*. 123–133. https://doi.org/10.1109/RTSS.2008.41

[2] Adam Dunkels, Joakim Eriksson, Niclas Finne, and Nicolas Tsiftes. 2011. *Powertrace: Network-level power profiling for low-power wireless networks*. Technical Report. Swedish Institute of Computer Science.

[3] Prabal Dutta, Mark Feldmeier, Joseph Paradiso, and David Culler. 2008. Energy Metering for Free: Augmenting Switching Regulators for Real-Time Monitoring. In *2008 International Conference on Information Processing in Sensor Networks (ISPN'08)*. 283–294. https://doi.org/10.1109/IPSN.2008.58

[4] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. 2008. Quanto: Tracking Energy in Networked Embedded Systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 323–338.

[5] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 234–251. https://doi.org/10.1145/3132747.3132786