Energy Harvesting Systems Need an Operating System Too

Samyukta Venkat
University of Virginia
svenkat@virginia.edu

Marshall Clyburn
University of Virginia
mdc5pv@virginia.edu

Bradford Campbell
University of Virginia
bradjc@virginia.edu

Abstract

Software support for intermittent devices has emerged as a key area of research in resource-constrained computing. Work in this area aims to ease application development by providing support for making forward progress in the face of frequent power outages. Typically, systems in prior work provide a runtime or a kernel as the system abstraction and are customized for a small set of hardware. In this paper, we propose our vision for the future of intermittent computing and explore extending a general-purpose embedded operating system to handle intermittent workloads. We show how many common OS abstractions benefit the highly constrained intermittent domain and describe the design extensions required to support intermittent devices. We evaluate the system with respect to memory, time, and developer overhead and argue that full OS support is a promising direction for future intermittent systems.

CCS Concepts: • **Computer systems organization** → **Embedded software**; • **Human-centered computing** → *Ubiquitous and mobile computing systems and tools.*

Keywords: Embedded Software, Energy Harvesting, Operating Systems, Intermittent Computing

ACM Reference Format:
Samyukta Venkat, Marshall Clyburn, and Bradford Campbell. 2020. Energy Harvesting Systems Need an Operating System Too. In *The 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys '20), November 16–19, 2020, Virtual Event, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3417308.3430274

1 Introduction

A wide variety of applications in health, transportation, energy, and other sectors could benefit from devices providing a consistent stream of data from various, often hard-to-reach places [12, 15]. However, these devices will need to be batteryless and capable of harvesting energy from their surroundings, and developing software for energy harvesting devices that experience frequent power outages is challenging [8]. Enabling developers to easily write software for ubiquitous, batteryless computing is paramount.

Generally, existing systems are special-purpose and are tailored to run on a narrow selection of hardware (i.e. WISP RFID) [14]. Developers must conform to the hardware and software stack that is implicitly specified by the system build. Furthermore, there is tight coupling between the applications and underlying software system. This makes switching between systems difficult, and it also leads to application changes requiring recompilation of the whole system.

In this work, we introduce the use of a general-purpose embedded operating system for intermittent computing. This is in contrast to existing, special-purpose systems that run on specific hardware. Given that, over time, most computing paradigms have shifted towards being more general-purpose, intermittent devices may do the same. Using an operating system provides structured abstractions and separation between applications and the kernel. It also grants portability between platforms, flexibility in software development, and the opportunity for updates and longevity.

For this proof-of-concept work, we augmented Tock [10] to support a task-based approach. We tested and evaluated the resulting system with respect to time, memory, and developer overhead. The contributions of this work are 1) proposing our vision for the direction of intermittent computing, 2) integrating support for intermittency into Tock and assessing its overhead, and 3) assessing the constraints of Tock after being extended for this new type of workload.

2 Background & Related Work

Similarly to continuously-powered embedded devices, the applications, system software, and hardware used as the building blocks for intermittent devices are becoming more flexible and complex. This section examines these approaches and how they have changed over time. We will also briefly look at how hardware is trying to push intermittent systems development to be more general-purpose.

ENSsys '20, November 16–19, 2020, Virtual Event, Japan

ACM ISBN 978-1-4503-8129-1/20/11…$15.00
https://doi.org/10.1145/3417308.3430274

2.1 Task-Based vs. Checkpointing

There are trade-offs between a checkpointing and a task-based method to making progress. There is greater developer overhead with task-based systems. The burden is on the developer to break down an application into appropriate chunks that can execute in a power cycle, and porting legacy code from the last 20 years of sensor networking isn't easy with a task-based approach. On the other hand, writes to nonvolatile memory for checkpointing are larger and usually happen more often than in task-based systems which means the former is often more performant [11]. State-of-the-art checkpointing could be closing that gap. However, it usually relies on having nonvolatile main-memory [9].

2.2 Energy Harvesting Platforms Are Becoming General Purpose

Early energy harvesting platforms were generally specialized sensor systems [6, 15]. However, like other computing platforms, energy harvesting needs are becoming more general-purpose. There has been a recent focus on flexibility which allows developers to customize hardware to their use case without needing to build a platform from scratch. Over the last decade, much of the research in intermittent software has used the *WISP* platform or a device that uses an MSP430 MCU with FRAM (same MCU as WISP) [3]. A large fraction of recent work uses RF energy as the target source which has slightly stifled development of platforms around other viable energy sources. *Flicker*, an energy harvesting platform, and *Capybara*, a dynamically-adapting software/hardware platform, have been developed [5, 7] to address this. Both of these platforms highlight a push in the direction of general-purpose platforms for intermittent computing.

As with hardware, several software systems using tasks or checkpointing have been proposed over the years with increasing sophistication. Checkpointing systems have gone from *Mementos*, inserting checkpoints at compile-time depending on instrumentation mode (e.g. after each function call) [13] to *TICS*, which improves on checkpointing by considering time sensitivity and memory consistency [9]. Task-based models have advanced from *Chain*, providing task-level granularity [4], to *Coala* saving state at a sub-task scale and using energy-aware adaptive scheduling [11]. Both approaches are becoming increasingly complex to address gaps in prior systems.

2.3 Tock Operating System

Tock is an open-source, secure operating system for embedded platforms with a kernel written in Rust. Tock provides a process abstraction similar to the abstractions provided by other general-purpose operating systems. One of Tock's main features is sandboxed process execution which maintains separation between each process and the kernel. In this environment, updating an application on a device is easier because only the application binary needs to be updated and not the entire firmware like prior work systems. The design properties of Tock make it a good candidate for this work.

3 Vision for Intermittent Software

We believe that innovations in intermittent computing along with the previously discussed increase in complexity of these systems make the need for a general-purpose, batteryless device inevitable. Here, we will discuss what our future vision for operating systems on intermittent devices looks like.

3.1 Flexibility in Software System Features

We envision a system where programmers can use default settings in an intermittent operating system or can change various parameters to suit their use case. This includes customizations such as employing more optimal scheduling algorithms or being more aggressive in saving state.

With an operating system, processes and their binaries are decoupled from the underlying system. This isolates faulty processes from the rest of the system and also means applications can change without having to alter and re-upload the operating system. Additionally, there will be increased flexibility in using external libraries and other languages. Devices would be easier to multiprogram as applications don't need to be compiled with the operating system.

Given that there are trade-offs between using a task-based approach and a checkpointing approach, a developer should be able to choose between them. In a system that supports both approaches, developers can choose, for example, to port a long-running, legacy program using checkpoints or to write a new, task-based application.

Lastly, porting an operating system to a new platform should not be difficult. A large portion of recent work all runs on the same family of MCUs and are all custom systems. Having an operating system with structured abstractions will make porting to new hardware much easier.

3.2 Updates & Cooperative Operation

If an energy harvesting node is deployed somewhere hard to reach and is intended to operate for decades, it is reasonable to expect that the node's functionality should be able to change over time. A node should be able to perform software updates to improve operation and patch security problems. An operating system will make updates easier by providing the ability to update an application in flash.

We commonly think of embedded devices and energy harvesting devices as single-purpose when their firmware is extremely specific to that purpose. This is distinctly different from other general-purpose platforms. Instead, the device could have several applications in flash that make use of the various on-board peripherals. A subset of these apps can be chosen to run thus determining the node's functionality. During the day, a node may report sensor readings and at night, switch to a different set of sensors or a long running computation. There can also be a notion of changing what

a node needs to do over longer periods of time. Similarly, there can be cooperation between nodes where a node can offload computation to another node if one node is in much better harvesting conditions than another. This adaptability is critical as energy harvesting devices fulfill their main motivation and scale up significantly in number. If this aspect of developing intermittent systems is not taken into consideration, the billions of devices we intend to be operating for decades will not be future-proof.

3.3 Image Recognition to Retrofit Meters & Gauges

Here, we will present an example of an application that can benefit from an operating system. There are many gauges and meters deployed in the world that are not "smart" and require manual checks. This can be highly inconvenient and can make quick response to issues infeasible. There is a desire to digitize and make these gauges wireless, and that desire is being addressed by products like EnergyCam [2]. EnergyCam's website claims that the Munich airport alone has 10,000 analog meters. With the need to retrofit meters and gauges around the world, it becomes clear that these devices will need to be batteryless. But EnergyCam must be battery or wall-powered. Instead, intermittent devices could take low-resolution images and perform analysis to transmit information about these meters and gauges.

This application would work well atop an operating system. It has a machine learning aspect that may require adaptation or updates over time. One could imagine that these devices could come in a form where sensing code could be left unchanged with only updates to the model or analysis code. This is made difficult if the whole system needs to be compiled and re-distributed. There are many such applications for intermittent devices, and the complexity could benefit from operating system.

4 Extending Tock for Intermittency

Tock was designed for continuously-powered systems, so additions and changes (shown in Figure 1) were required to allow it to support task-based execution. This section will lay out the design of features added to Tock.

4.1 Task Scheduling

As in prior work, the programmer must break the application into discrete parts, or tasks. Tock runs tasks based on a directed graph the programmer provides. If a power failure occurs during task execution, the device will next resume operation at the beginning of that task. With this approach, each Tock process is a task. Figure 2 illustrates how a task graph maps to Tock processes and how data is passed between them.

We altered Tock's scheduler to work with the task graph and determine which processes are ready to run. Due to Tock's existing design, even running complex graphs with multiple flows is easy because it can already handle multiple processes being ready. The scheduler checks provided dependencies and marks all processes that are not waiting for the completion of another process. Processes that are ready to run are then executed.

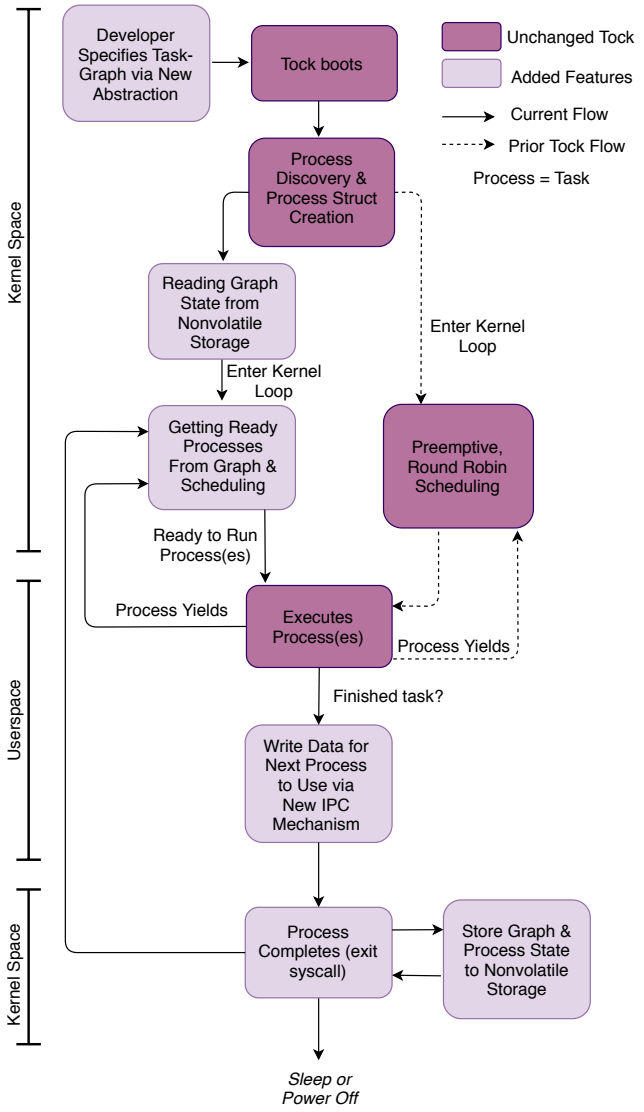

Figure 1. System Diagram. The darker boxes are Tock constructs that were largely left unchanged or were part of the system before. Lighter boxes are extensions for power failure immunity.

4.2 Saving State to Nonvolatile Memory

The kernel is responsible for writing state to nonvolatile memory so the device can restart at an appropriate place in the task graph after recovering from power failure. Before a power failure, the kernel writes information about progress through the task graph and any data that needs to be passed between processes. When Tock boots, it reads the task graph

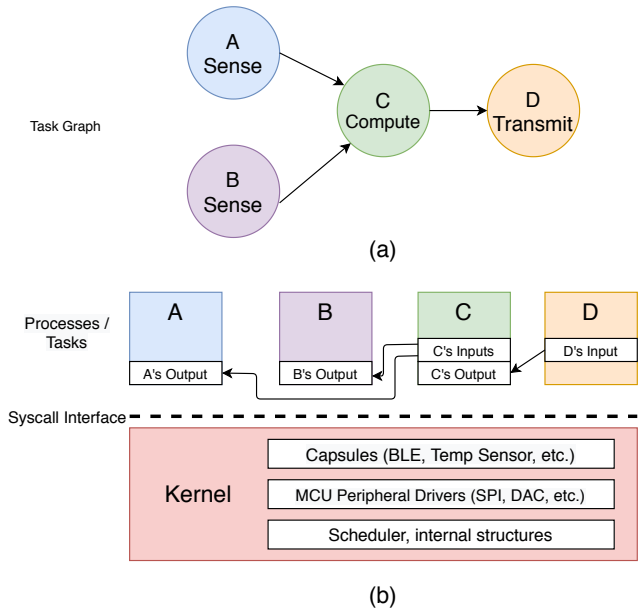

Figure 2. Task-based Tock system overview. (a) is an example task graph. (b) shows how tasks map to Tock's processes. Each task's input is a pointer to the previous task's output buffer.

progress (or initializes it if it's the first boot), computes ready-to-run processes, and begins executing them.

4.3 Inter-Process Communication

Tasks in the task graph need to be able to pass information between each other. Tock has an existing mechanism for inter-process communication, but it is not suitable for task-based execution. It requires two or more processes to coordinate to share data. One process must advertise that it will share data, and the other process must then decide to use the data. This does not map well to the task model.

In the new IPC mechanism, each process is given an output buffer allocated in its own memory space and one or more input buffers from other processes. This allows multiple processes to share data with a single process and vice-versa. At startup, all processes are loaded into main memory from flash. During this time, all input and output buffers are setup using the supplied task graph. Because there are possibly multiple input buffers for a single task, a process provides a specifier unique to the process it is reading from.

5 Implementation

This section will cover the details of the constructs used for augmenting Tock for intermittency. This includes specifics on how the kernel stores state and how information is passed between processes. There were times when existing design choices in Tock placed constraints on this work but also instances of Tock enabling much needed generality.

5.1 Graph Information Struct for Saving State

A struct containing information about processes is used to keep state about task execution. This struct, `graph_info` has four fields. The first is an initialization indicator that lets the kernel know if the struct has been initialized yet. The next three fields are arrays holding process state data. The first is an array of ready-to-run processes. The second is an array of ended processes. These two arrays together contain the information necessary to determine which processes can be executed. The final field is the data that needs to be passed between processes stored in an array. When all processes complete (excluding long-running, independent tasks), the graph is reset and program execution restarts.

5.2 Using Nonvolatile Memory in the Kernel

The kernel is responsible for making calls to read and write the graph state. For this implementation, the nonvolatile storage is flash, therefore writes are slow whereas reads are relatively fast. While flash was used for nonvolatile storage Tock has a unified interface for any type of nonvolatile storage. It could easily be swapped with another storage type, like FRAM.

5.3 Altering The Process Struct for IPC

The new IPC mechanism handles both multiple processes needing to pass data to one process and one process needing to pass input to multiple processes. An output buffer and pointers to input data from other processes are provided in a process' memory space. This works in much the same way stdin and stdout file descriptors do, except there can be multiple inputs. The maximum number of input pointers is limited by the number of processes that can be loaded into memory. The limit on the platform used for implementation is generally four.

5.4 Hardware Platform

This system was implemented on the Imix platform [1] which was developed for use with Tock. Imix uses a SAM4L Cortex-M4 MCU running at 48MHz with 512 kB of flash storage and 64kB of SRAM. The platform has a BLE radio, an accelerometer, and sensors for light, temperature, and humidity. We chose Imix due to Tock's stability on the platform and because it is a general-use device that allows a variety of applications to be developed.

6 Evaluation

We evaluate Tock for energy harvesting platforms by implementing a task-based application. We look at the memory, time, and developer overheads of Tock and discuss each one.

6.1 Experimental Setup

For our experiments, we ran the augmented version of Tock on an Imix board. The time overhead was measured using a Techtronix TBS2000 Series oscilloscope. A GPIO pin was

Table 1. Memory Overhead. Parenthesized values refer to how the Tockloader utility flashes applications onto the board. Tockloader loads processes into flash by padding binary size up to a power of two.

	Flash (w/ padding)	Memory
Kernel Unchanged	154.8KB (155.1KB)	24.6KB
Kernel for Intermittency	157.9KB (158.2KB)	32.8KB
Lighting App Sense	4.3KB (8.2KB)	8.2KB
Lighting App Compute	7.8KB (8.2KB)	8.2KB
Lighting App Transmit	30.7KB (32.8KB)	16.4KB

selectively toggled at runtime to assess the time taken by different parts of the code. In this implementation, any process ready to run will do so, the kernel will write the state to memory, and then the chip will sleep and wait to be rebooted. This made it easy to simulate power outages.

For testing, we used a lighting control application broken down into three tasks. First, it senses ambient light. Then, it performs a computation on that sample to yield a value by which lighting should be adjusted. Lastly, it advertises that value over BLE. The kernel was set up so that it ran a process and a physical button press caused a reset to happen. The metrics we used to assess Tock were the time overhead, the memory overhead, and the developer overhead.

6.2 Memory Overhead

Table 1 lists memory overhead. Tock has much larger overhead in both nonvolatile memory and RAM than prior work. Prior work systems are normally in the range of several KB in non-volatile storage and under 2 KB in memory. In part, this is to be expected as it is a general-purpose operating system and provides abstractions that require a larger memory footprint. Additionally, other systems don't have dedicated kernel or app space.

It's important to observe how changes made to the kernel affected memory overhead. The kernel grew by a few KB which had a significant impact on RAM usage, but there are places in the code that can still be optimized. Copying that takes place in the added code could be done more optimally. Non-trivial functions that were implemented could be optimized to reduce memory footprint. However, they do represent a significant augmentation to the functionality of the kernel.

There is a lot of additional process overhead from mapping the task abstraction to Tock processes. Libraries are compiled into each process separately and could possibly be duplicated. All applications issued `printf` commands, without which would have led to smaller memory footprints. The task abstraction makes sense for certain applications. It's easier to "plug-and-play", and it's often useful to write modular code. However, Tock may also lend itself well to checkpointing so that additional process overhead is limited.

Table 2. Time Overhead. For these measurements, the Tock bootloader, which allows interfacing with the device over USB, was removed.

Tock Function (Startup Cost)	≈ Time (μs)	Percentage
Total Tock Startup Time	8500	
Peripheral Initialization	7600	89.4%
Process Discovery/Creation	390	4.6%
Reading State from Flash	110	1.3%
Getting Ready Processes & Scheduling	400	4.7%
Tock Function (Addt. Overhead)	**≈ Time (μs)**	**Percentage**
Writing to Flash	10800	N/A

It isn't feasible to run this task-based implementation of Tock on platforms seen in much of the related work. However, with some optimizations and the addition of FRAM, it may be feasible to run on some slightly less constrained intermittent devices in the near future.

6.3 Time Overhead

Table 2 shows an itemized list of timings. With the exception of writing to flash, a lot of the time costs of Tock are in line with recent work. However, this comparison is not valid given the difference in the platforms used. For example, the Imix SAM4L's clock speed was set at 48MHz which is at least six times faster than the MCUs in related systems.

Instead, we can use the timings to see how the new code adds to the time overhead. The features added to Tock contribute approximately 6% to the startup time which shows that the new functionality doesn't increase the time overhead very much. The largest startup cost comes from initializing and configuring on-board peripherals. This is necessary since peripheral and capsule state is lost after power failure. This cost could be reduced by only initializing and configuring peripherals that are necessary for the selected applications. The other obvious part to consider is writing to flash, which takes longer than the startup time to complete. The best way to mitigate this is to use a faster storage medium, such as FRAM. There was also 200 ms of time from when the MCU was powered to when Tock started. This could be attributed to the chip performing operations such as clock and register setup, but further investigation is needed.

Process discovery in Tock is different than systems in prior work. By design, Tock inspects storage at runtime for headers containing address information to load processes into memory. This overhead is incurred only once and can be avoided by inserting address information at compile-time.

6.4 Developer Overhead

This system has reasonable usability with a few notable hindrances. New language constructs to learn include how to create task graphs, retrieve input/write output, and signal

task completion. In contrast with other systems, Tock does not need an additional construct for creating tasks since tasks map to processes. Some recent works have more language constructs to learn, however, those provide features like timing constraints which this implementation of Tock does not offer. Aside from the three new constructs, two of which go in application code, applications can be written like normal C code. There is, however, some difficulty in development that arises from the need to know the task's IDs. This is discussed further in Section 7.2.

6.5 Toggling Node Functionality

We implemented a second application to test using two different task graphs to toggle the operation of the node. Multiple task graphs in a task-based system is not new, however, this explores the notion of having a platform ship with apps and easily being able to change its function while deployed. We deployed four tasks to the board: sense temperature, print temperature, sense light, and print light. Based on input from a GPIO pin, the board would either sense and print the temperature or sense and print the light level. The device needed to reboot before a new task graph could be assigned, but given the frequency of rebooting, this is not a problem.

7 Limitations & Future Work

In this section we will address the shortcomings of our system and the next steps to improving the performance of Tock for intermittent computing.

7.1 Tock Inefficiencies

In this implementation, Tock's process discovery and creation model was left relatively untouched. As mentioned previously, Tockloader loads processes into flash by rounding the binary size up to a power of two. Then, based on a configured parameter for the number of processes to load and the space in RAM, Tock loads all of the processes it discovers and has room for from flash at boot. This is problematic for systems where tasks are energy-atomic. Systems with large computations can have many nodes in the task graph. Another challenge is that inputs in Tock are pointers to a previous task's space in RAM. We explored loading only the necessary processes (i.e. processes that will execute and their dependencies), but this feature was not fully implemented.

7.2 Tooling for Graph Processing

Our system relies on the developer to supply a manually constructed task graph that matches processes on-board. Process IDs used to construct the graph must be obtained manually. This is not particularly difficult, but it is sub-optimal. Instead, constructing a graph of task names or other suitable identifiers that is later converted to the correct representation would be a more seamless experience.

7.3 Inter-Process Buffers

A process needs its predecessors in RAM to get valid input pointers. This is an inconvenience, and there are a number of different ways to change process discovery. State and outputs could be saved with each process instead of in the `graph_info` struct. This would allow the kernel to perform a hand-off of the inputs.

Process output buffers were small and could only pass up to four bytes. This was sufficient for a proof-of-concept, but increasing the buffer size should be relatively simple.

7.4 Checkpointing

While this work focuses on extending Tock with a task-based approach, an important step towards enabling the future vision laid out in Section 3.1 is adding checkpointing to Tock. This could be using any of the state-of-the-art checkpointing technologies such as checkpoints assigned at compile-time then altered at runtime. The addition of checkpointing to Tock will provide developers with the ability to port existing code bases to new systems.

7.5 Memory Protection Unit

The memory protection unit was disabled for this work to allow processes to write to their output in the grant region within their own process memory. Other processes then have read access to this one area. For future iterations the memory protection unit will be enabled to ensure that processes cannot do anything malicious or erroneous.

8 Conclusion

Intermittent systems are becoming more general-purpose and, if not now, will be in need of an operating system soon. We cannot expect to deploy billions or trillions of sensors with software tightly coupled with hardware and also expect that they will operate optimally over their lifetime. We present our vision for full OS support and show how a modified version of a particular OS, Tock, is well suited for energy-harvesting devices.

Like previous computing paradigms, we expect application-specific intermittent devices to transition to general-purpose platforms over time. This should enable more sophisticated devices with applications that evolve over a device's lifetime, with proactive security updates, and with suitability for more use cases. Moving away from application-specific and custom intermittent devices will also simplify development and increase the reach of battery-less devices, ultimately helping these devices reach the transformative scale that motivates their existence.

References

[1] 2020. *Imix*. Retrieved July 10, 2020 from https://github.com/helena-project/imix

[2] 2020. *Q-Loud EnergyCam*. Retrieved July 10, 2020 from https://www.q-loud.de/energycam-automatic-mechanical-meter-reading

[3] 2020. *WISP5*. Retrieved July 10, 2020 from http://wisp5.wispsensor.net/
[4] Alexei Colin and Brandon Lucia. 2016. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 514–530.
[5] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 767–781.
[6] Samuel DeBruin, Bradford Campbell, and Prabal Dutta. 2013. Monjolo: An energy-harvesting energy meter architecture. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*. 1–14.
[7] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–13.
[8] Josiah Hester and Jacob Sorber. 2017. The future of sensing is batteryless, intermittent, and awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–6.
[9] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. 2020. Time-sensitive Intermittent Computing Meets Legacy Software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 85–99.
[10] 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*. 234–251.
[11] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. 2020. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Transactions on Sensor Networks (TOSN)* 16, 1 (2020), 1–24.
[12] J Paulo and Pedro Dinis Gaspar. 2010. Review and future trend of energy harvesting methods for portable medical devices. In *Proceedings of the world congress on engineering*, Vol. 2. WCE, 168–196.
[13] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System support for long-running computation on RFID-scale devices. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. 159–170.
[14] Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. 2008. Design of an RFID-based battery-free programmable sensing platform. *IEEE transactions on instrumentation and measurement* 57, 11 (2008), 2608–2615.
[15] Sujesha Sudevalayam and Purushottam Kulkarni. 2010. Energy harvesting sensor nodes: Survey and implications. *IEEE Communications Surveys & Tutorials* 13, 3 (2010), 443–461.