Towards a Language and Runtime for Intermittently-Powered Devices

Josiah Hester
Clemson University
School of Computing
jhester@clemson.edu

Kevin Storer
Clemson University
School of Computing
kstorer@clemson.edu

Lanny Sitanayah
Clemson University
School of Computing
lsitana@clemson.edu

Jacob Sorber
Clemson University
School of Computing
jsorber@clemson.edu

ABSTRACT

Tiny batteryless RFID-scale sensing devices promise to transform science and society by enabling long-term, maintenance-free data gathering. These intermittently-powered devices are a key enabling technology for the future Internet-of-Things (IoT): as the economics of disposal and recharging of batteries do not scale to the trillions of devices in the IoT. System designers currently lack the end-to-end solution for designing and deploying these devices en masse. In this paper we describe a potential language and runtime for the tiny, intermittently-powered devices, that will become the fundamental units of the IoT. We consider the many challenges, and outline requirements, for such a language. We then cast the language in the broader scope of an end-to-end toolchain for intermittently-powered devices. Finally, we discuss how languages for these devices should be evaluated–with a focus on usability, and ease of adoption–and outline questions guiding any evaluation of future languages.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Architectures*; • **Human-centered computing →** *Ubiquitous and mobile computing systems and tools*; • **Software and its engineering →** *Context specific languages*;

KEYWORDS

Task language, Time, Batteryless, Intermittent, Energy harvesting

ACM Reference format:
Josiah Hester, Kevin Storer, Lanny Sitanayah, and Jacob Sorber. 2016. Towards a Language and Runtime for Intermittently-Powered Devices. In *Proceedings of Hilariously Low Power Computing Workshop, Atlanta, USA, April 2, 2016 (HLPC),* 6 pages.
DOI: ...

1 INTRODUCTION

Low cost wireless sensors that just work, maintenance-free, for decades—this dream has motivated decades of research, countless papers, and multiple conferences, and it continues to elude us. Why? Because, nearly all sensors depend on batteries, and today's batteries all wear out after a few years, heedless of our efforts to recharge and conserve them. The alternative? Batteryless energy-harvesting sensors offer hope, but irregular power supplies and scant energy storage lead to frequent power failures, and creating software that works reliably in spite of frequent failures is challenging.

A transiently-powered sensor may lose power at any time, with little warning, for unpredictable lengths of time. Time can be difficult to measure, and execution times become difficult to predict. Programs may lose data, corrupt data, or fail to make forward progress on long-running computations. Forward progress can be preserved, if applications checkpoint their state to nonvolatile memory (*e.g.,*Flash or FRAM) before a failure [14]. With programmer defined memory fences [11] programmers can keep non-volatile data structures consistent. These memory fences rollback any operations performed after a fence, so that programmers can be confident that execution before the last power failure does not affect the current state. Developers can even use physical hardware properties to estimate how long a device spends without power [13]; however, responding to dramatically unpredictable execution delays can be complicated.

Data age, priorities change, and applications adapt. Sensor data, once urgent, may only be useful for a few minutes or even seconds. When data expire, after long outages, partial computations may need to be discarded and possibly restarted. After short outages, an application may pick-up where it left off. On power-up, an application's priorities may have changed. A timely task may take precedence over a work-in-progress. A task that has repeatedly failed to complete (due to expired data), might be swapped out for a low-power alternative.

Each of these cases are easy to motivate, easy to explain, easy to understand, and difficult to implement with today's languages and runtime tools. Common imperative programming languages, like C, ignore time and how it relates to data. Programmers expect most tasks to run much faster than necessary and sequential instructions to run close together in time. Application goals and constraints are represented implicitly, if at all, and a C program that discards expired data, or schedules tasks appropriately in spite of power failures, or adapts to changing energy conditions will invariably be full of explicit time checks and cluttered with extensive branching logic.

HLPC, Atlanta, USA
2016. ...…$15.00
DOI: ...

We believe 1) that those who use existing languages to program transiently-powered devices are doomed to frustration and complication, 2) that those complications prevent capable programmers from creating compelling intermittent applications, and 3) that programming for transient power can (and should) be simple.

This paper is a call to rethink how we program for transient power — a call to develop new programming models and runtime techniques as a first step towards simple software development on intermittently-powered devices.

In the following sections, we describe why it is difficult to program intermittent sensors (Section 2) and what requirements a new programming model should have (Section 3). We outline two different possible approaches (Section 4), both of which explicitly capture an application's relationship to time and allow developers to focus on application goals and tasks instead of recovering from power failures and monitoring the passing of time. Finally, we discuss how we should evaluate candidate languages, and recognize the right languages when we see them (Section **??**).

2 BACKGROUND AND CHALLENGES

Language and runtime support for batteryless sensing is central to simplifying the implementation of energy-harvesting devices, and consequently realizing the vision of the IoT. Batteryless sensing applications have traditionally been extremely simple—all tasks must happen in one power cycle. This means that tasks must be uncomplicated, and execute quickly and independently of the successful completion of prior duty cycles. Otherwise, it is unlikely the program will complete before a power loss. In response to this, checkpointing systems like Mementos [14], Hibernus [1], QuickRecall [8], and others have emerged to keep forward progress. Other solutions, like DINO [11], show that even with checkpointing, memory consistency is not maintained. DINO complements checkpointing systems while maintaining memory consistency, and giving idempotent and atomic attributes to sections of code, as defined by the developer. DINO, and checkpointing systems, do not consider how the loss of timekeeping affects the duty cycle, and how the utility of sensed and computed data changes over time. In short, current language techniques do not track or enforce data freshness.

Languages for embedded systems have not progressed much beyond NesC [5], which was neither energy aware, or built for unstable power. Eon [15] took a step forward by considering energy a fundamental part of the language, just as it is a fundamental hardware constraint. Eon was the first energy-aware programming language for sensors. However, Eon is not suitable for intermittent computing devices, because of its high maximum energy draw. Eon assumes that energy costs will be amortized over a long period, so the rare, high energy operations required by the runtime are offset by the low energy steady state. For intermittently-powered devices, the maximum draw matters much more than the average draw. The Chlorophyll [12] programming model and compiler is an example of a new language and runtime built for ultra low power devices. Using syntax-based synthesis techniques, high level programmer intent is translated into low level code. Chlorophyll's unique approach to programming a minimalist, ultra low-power, spatial architecture increased usability and development speed. However, the unique hardware architecture for which Chlorophyll was built makes it difficult to reuse any part of the system for intermittently-powered sensing devices. Chlorophyll assumed a stable (if limited) power supply and did not consider properties of sensor data.

Listing 1: Battery powered greenhouse monitoring program

```
1  void main(){
2    while(1) {
3      int t = temp();
4      int l = light();
5      int m = moisture();
6      int w = wetness(t,l, m);
7      send(w);
8      sleep();
9    }
10 }
```

Current language solutions for simplifying batteryless computing do not effectively maintain time across power failures, are too heavyweight for implementation on constrained devices, or do not actually simplify development. There are significant challenges to implementing a successful language for batteryless devices. In this section, we describe these challenges in the context of a greenhouse monitoring (GHM) application currently in development. Greenhouses waste water by overwatering plants. Especially in large greenhouses, it is difficult for managers to keep track of the status of individual plant beds; forcing them to overwater to ensure plants do not die. This waste can be significant for large greenhouses in terms of economic costs, and sustainability reasons. Currently, commercial plant monitoring systems are simple weather stations with dedicated power supplies that monitor an area with hundreds or thousands of plants. Because these weather stations are large and expensive, they are impractical to deploy densely; worse, they cannot be moved with the plants they monitor. This means that any gathered data is not fine-grained, not localized, and inaccurate in the case of plants that were moved. Greenhouse monitoring offers a unique application for batteryless sensors, where densely-deployed, tiny, cheap, sustainable sensors can be paired with individual plants or trays, to monitor the water usage. Pseudo-code for a simplified battery powered GHM duty cycle is shown in Listing 1. Each sensor is read, a leaf wetness reading is calculated from an embedded model, and then sent to a basestation, then the sensor sleeps before performing the duty-cycle again. This simple application is straightforward to program, test, and deploy, with a stable power supply. However, designing an effective *batteryless* program for this application is daunting, as we detail below.

Challenge #1: Sensor Data

Maintaining relevant sensor data, and discarding old or useless sensor data is key to the efficiency of any embedded application. This task becomes more difficult in the face of intermittent power. Each sensor used in our GHM application has different constraints on how useful its data is or how much is required to make a judgement on water usage. Additionally, semi-complex computation operations on the sensor data are sometimes required. Worse, these may need to span multiple power cycles. However, if the time between resets was too long, the computation needs to be restarted. These

Sensor	Power	Startup Time	Expiration
Temperature	12 µW	1ms	5min
Ambient Light	126 µW	10ms	10sec
Moisture	1800 µW	2000ms	2min

Table 1: Each sensor has different timing and production constraints, as well as data-freshness constraints specified by the application, domain scientists, and the programmer. This table shows the power, turn on times, and the amount of time after generation that the data is fresh, of three common sensors for GHM applications.

constraints make the time that a data point was collected very important. For instance, if a humidity reading is over an hour old, then it is pointless to transmit to the basestation. On the other hand, if it is recent, and previous history shows a sharp change at an alarming rate, then transmission is crucial. By using remanence based timekeepers such as TARDIS [13], the amount of time between power failures can be quantified. These timekeepers charge up an external capacitive component on startup, which slowly degrades when power fails. By measuring the amount of decay, the application can get a coarse understanding of the time the device was off. With these timekeeper, these data management tasks could be performed in embedded-C; however, the high cognitive costs of managing the timekeeping required to keep sensor data relevant and ordered is prohibitive to developing complex applications.

Challenge #2: Scheduling

Intermittently-powered devices are almost universally powered by some form of energy harvesting. This energy is stored in a small capacitor (or capacitors), and used quickly after harvest before leakage drains the energy store. Different sensing tasks require different amounts of energy. Long-running tasks may continually restart and never complete, if not handled carefully. Tasks may depend on other tasks, or on external events. Some tasks may have to be repeated or redone to remain relevant. As sensing schedules become complex, managing energy and allocating it to specific tasks becomes a best guess problem. This scheduling becomes even more complicated in the face of volatile energy environments, and uncertain execution. Matching the available energy with the current tasks that can be executed becomes a balancing act from the language and runtime standpoint, even with Federated Energy storage techqniques [7] that reduce failures.

In our GHM application, each sensor requires a different amount of energy to turn on and take a reading as shown in Table 1; in some cases, the sensor must take multiple readings over a window period, in others, the sensor should only be read if other sensors have returned positive data. With a stable, predictable power supply (such as a battery) this problem is easier to code and visualize, and devolves into a duty cycle calculation for each sensor. Solar energy harvesting, while usually more stable than RF, is unpredictable, even in a greenhouse. Deciding whether to perform a low-cost temperature reading or a high-cost humidity reading is not trivial with limited time, computational resources, and no memory of past events. Avoiding Sisyphean[1] tasks is a key challenge to software development for batteryless devices [11, 14]. Maintaining time through power failures is difficult, but not impossible [13]. Even with these tools, however, the complexity of keeping an intricate sensing schedule while maintaining checkpoints, memory consistency, and time consistency, makes for a difficult programming chore.

Challenge #3: Adaptation

What happens when requirements change? Batteryless sensing devices, by necessity, will not always be able to perform the preferred sensing schedule. This usually happens when the energy environment becomes too sparse to provide the energy to run tasks to completion. However, rarely is the device completely dead; free energy can still be harvested and put to use. Existing tools, like Eon, showed how language could be used to denote different tasks that could be performed in different energy conditions. This adaptive mechanism is even more necessary for intermittently-powered sensors, than sensors with a stable supply—because there is no normal, energy environments change rapidly, and many different energy scenarios will play out over the lifetime of a sensor. This is apparent in the GHM application, where seasonal changes require different duty cycles. Less ambient energy from the sun is harvested during winter and fall. Environmental changes can happen much faster than seasonally; rain, cloudy skies, or other weather events can decrease the available energy and force the greenhouse sensor to adapt its sensor schedule. Adaptation is not always caused by the environment. Rare events, stemming from interesting sensor data, or instructions from a basestation, can change the priorities of a sensor just as quickly as a changing environment. These priorities are difficult to transcribe with conventional languages (embedded-C). Programs tend to turn into absurdly obtuse and large switch-case statements that all interconnect with one another. Confusing the original author, and anyone else unfortunate enough to join the project.

3 REQUIREMENTS

Current languages and operating systems for wireless sensor networks [4, 5, 9], all assume a stable power supply; one associated with a large battery or supercapacitor. A successful language and runtime for batteryless devices must satisfy some core requirements that address the challenges presented previously. Current languages, programming models, and runtimes for sensors do not meet the challenges presented by *batteryless* sensing. Specifically, they cannot effectively maintain time across power failures, are too heavyweight in terms of energy for implementation on energy-harvesting devices, or do not actually simplify development.

Focus on Goals and Tasks: Dealing with power failures in the application code for batteryless devices is inherently complex. Developers must spend a lot of time debugging code, inserting checkpoints and asserts that preserve execution flow, and preserving time. They rely on a variety of methods that, if not properly implemented, can lead to failure and data loss. Approaches like Dewdrop [2] adapt task scheduling to energy conditions, but force the developer to

[1] Sisyphus was a character in Greek mythology doomed to spend eternity pushing a boulder up a hill, only for it to roll back down.

constrain their programs to simple, independant, short programs that run to completion.

This is a distraction; programmers should be focusing on the tasks they want to finish, that map to the goals they want to achieve. Developers should be focused on gathering data, processing data, and delivering data, without worrying about power failures. Requiring the application developer to know how to implement each of the systems that help mask power failure is too burdensome. Even if these subsystems are packaged as libraries, the developer must know how to respond to a low power situation, and adjust the execution in response using all the systems correctly; this requires deep domain specific skill, and application specific knowledge. Application specific information (*e.g.*,the data being collected, task dependencies, and program objectives) gets lost in the details of implementation. For example, a new language for batteryless sensors should allow commands of the type "take five humidity readings evenly distributed over a one hour window" or "only send this temperature dataset, if it was collected within ten minutes and the humidity reading is above 65%." These statements amount to actionable sensing tasks that programmers can easily grasp, without the distraction of dealing with intermittence. Languages suitable for intermittently-powered devices must separate goals and tasks from implementation. The intricacies of managing checkpoints, maintaining memory consistency, keeping time, and managing energy, should be hidden from the programmer. Only once a language has explicitly captured a programmer's goals, constraints, and policies, can a runtime system automatically implement a coherent application from fragmented execution opportunities.

Explicit Execution Flow: In current programming models that deal with intermittence, the dataflow and program execution path are implicit. C-style languages are usually regarded as explicit, however, maping conditional logic statements across power failures challenges this assumption. While existing tools preserve forward progress [11, 14], imperative programming languages, by definition, allow execution to jump through different points in the code. This implicit flow can be difficult to handle even with stable power; exactly why breakpoints, and debuggers like GDB were implemented. When power becomes intermittent, this implicitness becomes very confusing, even with checkpointing. Developers are reduced to piecing together fragments of execution with little confidence and few ways to check their correctness. Instead of putting the onus on the developer to identify execution fragments and arrange them meaningfully, why not change the language such that the execution is explicit? Any language intended for batteryless devices must be explicit in terms of dataflow and execution flow. Tasks must follow one from the other as defined by the developer. This makes applications cleaner, easier to write, and simpler to debug.

Usable: Usability is a key requirement of any language for intermittent devices. If a language, and its associated tools are not significantly easier to work with than the standard (embedded-C) then no one will migrate. Worse, a system that is not usable will hinder new developers from picking up intermittently-powered devices as a solution to their application needs. The current set of tools are difficult to use, and require extreme patience and dedication to implement even the most simplistic applications. However, embedded-C has the advantage of familiarity; any language built specifically for batteryless devices will have to provide benefits that obviously outweigh the benefit of familiarity.

Simple Runtime: Batteryless devices are constrained in terms of memory (often less than 64 KB), size (1 inch), speed (1 MHz), and instruction set. These are low-performance systems that (for now) cannot conduct uninterrupted complex or long running calculations. Because of these constraints, any runtime for batteryless devices must be able to make its decisions quickly, since the time it used to make decisions could have been used for executing user tasks. Therefore, any non-user tasks must have a small energy footprint and use as little non-volatile memory and RAM as possible. This likely precludes the use of any dynamic programming language on these devices, and pushes most of the complexity onto the compiler.

The runtime must also consider the hierarchy of types of memory that modern microcontrollers use. Volatile SRAM, non-volatile FLASH, and FRAM, can all be on the same IC. Any runtime must be able to keep the memory state consistent for both non-volatile and volatile memory, otherwise adding to programmer confusion.

Extensible Core: Even though the language must be focused on the big picture (the application's goals and sensing tasks), the inner working of both the language and runtime must be exposed in such a way to make extensions possible, and straightforward. This is especially important while the field surrounding intermittently-powered devices is in its infancy; as new mote platforms with different sensing capabilities are created, the language must be able to adjust. This could mean simple libraries that can be used inside the language. However, the requirements for extensions must be made in such a way as to preserve the goal-oriented nature. This means that extra constraints are put on extension writers, such as forcing labeling of atomic code segments, and promising no side effects.

4 PROGRAMMING MODELS

Currently the standard for programming intermittently-powered devices is embedded-C. C provides the low-level access and familiar, imperative programming model that is expected for an embedded system. However, the implicit nature of execution path and dataflow of C programs, when run on batteryless devices, is a serious obstacle to development of sophisticated applications. Aside from this, any language for intermittently-powered devices needs to allow programmers to specify their intent, and not worry about writing adaptive code for every energy situation. Describing goals and tasks, instead of spending time handling intermittence allows quicker, simpler development cycles, as well as reusable code. In this section we consider two alternative programming models that could be a good fit for the requirements listed in the previous section.

4.1 Syntax-guided Synthesis

Syntax-guided program synthesis is a language technique where a compiler attempts to automatically synthesize an implementation of a program, that satisfies a specification that determines correctness. This specification is written by the programmer, and refined through multiple attempts, in a hopefully synergistic collaboration between compiler and programmer. This programming model and runtime match the goal-oriented requirement of languages for

batteryless devices; where the correctness specification input to the synthesis compiler allows developers to define the applications dataflow and timing constraints of each task. The partial programs, correctness specifications, and task information, are viewed as a communication medium between the application developer and the synthesizing compiler. This method also puts the onus of finding a solution on the compiler side, allowing the runtime to be slim and fast. The synthesis approach has the advantage that the language can be a superset or subset of embedded-C, allowing quick migration for those already comfortable with current tools. As long as the semantics of the additions are explicit, and the order of execution becomes more apparent with syntactic sugar, the implicitness of data and execution in C programs in intermittent environments can be somewhat (but not entirely) overcome.

Sketching Intermittent Programs: As an example of one synthesis programming language, consider SKETCH [10], which was written for non-experts as a response to the deductive synthesis-based techniques that required expert-level knowledge to use. In Sketch, programmers provide an incomplete program, or a "'Sketch,"' that is a high-level view of the application goals. This sketch expresses structure, with holes for low-level implementation details. The holes are then filled by an SAT-based, inductive synthesis procedure that creates an implementation from a set of test cases. Then a counterexample-guided, inductive synthesis procedure (CEGIS) automatically generates test inputs to validate the implementation, and ensure that the program satisfies the specification. Developers could assert certain conditions about data freshness, and other attributes, and leave placeholders for sensing data. Code for a potential sketch of a greenhouse monitoring application is shown in Listing 2. From the application developer's point of view, the only things necessary to specify are the placeholder variables, the structure, and the assert statements that define the constraints on tasks. For batteryless devices, these specifications (assert statements) could be written as timing and dataflow requirements. Potential synthesis specification for the greenhouse monitoring application would include expiration requirements for data, assertions that certain data is gathered before continuing, and other syntactic sugar that guards against implicit execution, and keeps the developer focused on sensing goals. The synthesizing compiler would take these assert statements and structure and attempt to construct an iteratively-refined program that maintains execution consistency, memory consistency, and manages timing constraints.

Challenges: There are significant challenges to any implementation of a synthesis compiler for batteryless devices. The development of a usable synthesis compiler is daunting, as the combination of machine learning techniques and requirements for instant feedback make for a complex implementation. One of the major concerns with any synthesis compiler for batteryless devices is the verification phase. Verifying functionality would require either a hefty, dynamic proofing system, a simulation technique, or an external hardware emulator to take generated programs and test them. None of these methods currently exist in the literature surrounding batteryless devices, however, some existing systems like Ekho [6] and the Energy Interference-Free Debugger (EDB) [3] could form the basis of a verification solution. Another challenge is the complexity of extracting the control flow from a C program,

Listing 2: Greenhouse program with specifications

```
1  void main(){
2    while(1) {
3      int t = temp();
4      int l = light();
5      int m = moisture();
6      int w = wetness(t,l, m);
7      // Specification
8      assert age(t) < MINUTES(5) &&
9             age(l) < SECONDS(10) &&
10            age(m) < MINUTES(2);
11     send(w);
12     sleep();
13   }
14 }
```

and therefore which points to use to automatically checkpoint and timestamp sensor data and define atomicity bounds. This automatic process could be helped by annotations from the developer.

4.2 Coordination with Graphs

A graph-based coordination language is another model that has appeal for batteryless devices. In this model, programmers create a graph that encapsulates how the application should respond to events, and in what sequence tasks should be executed. This model was used in Eon [15] to provide a language for energy-aware perpetual systems. Allowing programmers to define a graph-based structure for their programs has the obvious advantage of explicit control flow. It also gives bounds on what tasks are dependent, and explicit code boundaries for checkpointing and atomicity bounds. These tasks are similar to Dewdrop style tasks, but allow dependancies, data, and timing constraints. Developers can easily describe sensing goal with graphs, where edges and nodes can be annotated with developer defined constraints (such as timing and data windows), which can be solved for by the runtime. The use of a graph-based structure forces developers to define atomic tasks explicitly, making the runtime decisions much easier when deciding on checkpoint location and how to maintain consistency. The graph-based structure also has the advantage of being very extensible, as the code behind each task node can be in embedded-C, allowing reuse of existing code and libraries.

Building Task Graphs: Application developers using a coordination language would first sketch the high level policy goals, possibly starting with a statement like "'I want to understand water use per plant in a greenhouse."' This statement is further refined and separated into individual tasks that map to embedded-C code. For instance: "'sample temperature, light, and humidity, then perform a moisture calculation."' An example of a high-level view of the greenhouse monitoring application is presented in Figure 1. In this figure, nodes map to tasks, and proceed based on edge constraints that define data, energy, and timing requirements. Edges that generate data have expiration constraints, since developers need to define what data is acceptable for the application (specifically, calculating leaf wetness).

Challenges: A key challenge for this model is where to define task boundaries. While the programmer is expected to define most of

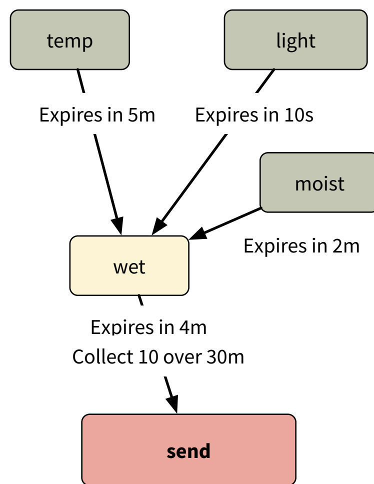

Figure 1: This figure shows how a coordination graph could be used to define tasks, goals, and the constraints on data generation, for a greenhouse monitoring application. It would then be the work of the compiler, and runtime, to determine how best to achieve this sensing schedule.

these, must the runtime trust the developer's intuition about the boundaries? Another challenge is the actual scheduling of tasks: most likely, this schedule will need to be predetermined by the compiler. So, deciding a correct ordering from the developer supplied annotations is crucial. However, a more dynamic task scheduler could provide greater adaptability, as it could change priorities on the flow, based on energy available or new information from a basestation. As with the synthesis-based approach, a form of verification or instant feedback is needed so that developers know how well their program is performing. This could be accomplished in a model-driven simulator, or using a combination of emulation techniques [3, 6].

4.3 Takeaways

Both these techniques provide a promising vision for languages for use in batteryless devices. The focus on usability and simplicity from the application developer's point of view is well met, in both the graph-based coordination language and the synthesis compiler. There are, however, challenges to both approaches in defining task boundaries, preserving atomicity, and maintaining memory and time consistency in the runtime. On the development side, real-time and instant simulation or emulation tools are needed to verify solutions for both models, before deployment. These tools may need to be designed in tandem with the language, otherwise user confidence in the solution will be at risk.

5 CONCLUSIONS

In this paper, we take the position that a new programming model is needed for batteryless, intermittently-powered devices. The lack of toolchain support for these devices, specifically in terms of language, is discouraging new developers and stifling new applications. The complexity of development for these devices is high, and the learning curve steep. However, best practices and techniques can be applied to any language, which will mask the effects of energy management and time volatility. We hope the community will take note of the challenges presented, and the requirements, and begin to consider the possibilities of language for batteryless devices.

REFERENCES

[1] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *Embedded Systems Letters, IEEE* 7, 1 (2015), 15–18.

[2] Michael Buettner, Ben Greenstein, and David Wetherall. 2011. Dewdrop: An Energy-Aware Task Scheduler for Computational RFID. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation.*

[3] Alexei Colin, Alanson P. Sample, and Brandon Lucia. 2015. Energy-interference-free System and Toolchain Support for Energy-harvesting Devices. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '15).* IEEE Press, Piscataway, NJ, USA, 35–36. http://dl.acm.org/citation.cfm?id=2830689.2830695

[4] A. Dunkels, B. Grönvall, and T. Voigt. 2004. Contiki—A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proc. 1st IEEE Workshop on Embedded Networked Sensors (Emnets-I).* IEEE Computer Society, Tampa, Florida, USA.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proc. ACM SIGPLAN 2003 Conf. Programming Language Design and Implementation (PLDI'03).* ACM, San Diego, CA, USA, 1–11.

[6] J. Hester, T. Scott, and J. Sorber. 2014. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-Harvesting Sensors. In *Proc. 12th ACM Conf. Embedded Network Sensor Systems (SenSys'14).* ACM, Memphis, TN, USA, 1–15.

[7] J. Hester, L. Sitanayah, and J. Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *Proc. 13th ACM Conf. Embedded Network Sensor Systems (SenSys'15).* ACM, Seoul, Korea.

[8] Harishankar Jayakumar, Arnab Raha, and Vijay Raghunathan. 2014. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on.* IEEE, 330–335.

[9] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. 2004. TinyOS: An Operating System for Sensor Networks. In *Ambient Intelligence.* Springer Verlag.

[10] A Solar Lezama. 2008. *Program synthesis by sketching.* Ph.D. Dissertation. Citeseer.

[11] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015).* ACM, New York, NY, USA, 575–585. DOI:https://doi.org/10.1145/2737924.2737978

[12] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14).* ACM, New York, NY, USA, 396–407. DOI:https://doi.org/10.1145/2594291.2594339

[13] Amir Rahmati, Mastooreh Salajegheh, Dan Holcomb, Jacob Sorber, Wayne P Burleson, and Kevin Fu. 2012. TARDIS: Time and remanence decay in SRAM to implement secure protocols on embedded devices without clocks. In *Proceedings of the 21st USENIX conference on Security symposium.* USENIX Association, 36–36.

[14] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-Running Computation on RFID-Scale Devices.. In *Proceedings of the 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[15] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. 2007. Eon: A Language and Runtime System for Perpetual Systems. In *Proc. ACM SenSys.*