Feasibility of Multi-Tenancy on Intermittent Power

Dimitris Patoukas
Delft University of Technology
d.patoukas@student.tudelft.nl

Kasım Sinan Yıldırım
Delft University of Technology
k.s.yildirim@tudelft.nl

Amjad Yousef Majid
Delft University of Technology
a.y.majid@tudelft.nl

Josiah Hester
Northwestern University
josiah@northwestern.edu

Przemysław Pawełczak
Delft University of Technology
p.pawelczak@tudelft.nl

ABSTRACT

Energy harvesting and battery-free sensing devices show great promise for revolutionizing computing in the home, in the wild, and on the body. The promise of cheap, dense, and ubiquitous sensing technology brings new applications for the Internet of Things. However, the future programming model is blurry and complex. With a potential for trillions of devices, and thousands of devices per person on earth, programming languages and associated operating systems must be usable, flexible, and resource efficient. Because of the thousands of applications and fine grained differences in requirements, multi-tenancy may be a part of the solution to solving this programming model crisis. This paper explores the energy and resources costs, feasibility, and motivation for multi-tenancy on these tiniest of computing devices—namely the difficulties in scheduling tasks fairly, efficiently, and simply. Because of intermittent power, resources and energy must be mostly devoted towards user tasks, we implement a rudimentary operating system with low overhead to conduct experiments and test time-sharing and scheduling protocols. We close with a discussion on challenges to implementing a multi-tenant run-time on battery-free tags, and proposals for future work.

CCS CONCEPTS

• **Computer systems organization → Sensor networks**; **Embedded software**; Real-time operating systems;

KEYWORDS

transiently powered computers, wireless sensor networks, scheduling

ACM Reference Format:
Dimitris Patoukas, Kasım Sinan Yıldırım, Amjad Yousef Majid, Josiah Hester, and Przemysław Pawełczak. 2018. Feasibility of Multi-Tenancy on Intermittent Power. In *ENSsys '18: International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems, November 4, 2018, Shenzhen, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3279755.3279761

ENSsys '18, November 4, 2018, Shenzhen, China

ACM ISBN 978-1-4503-6047-0/18/11…$15.00
https://doi.org/10.1145/3279755.3279761

1 INTRODUCTION

Ubiquitous computing and ambient intelligence demand un-tethered nodes that can sense and compute in deeply embedded applications. However, the major constraint of un-tethered nodes is their finite battery capacity. This bounds the lifetime of the deployed device, not only hindering deployment flexibility, but also introducing additional cost and complexity of recharging, replacing and maintaining batteries. Although low power processors and new battery materials are increasing the lifespan of embedded systems, energy density does not scale exponentially like computing power traditionally has, but linearly [11] bounding the future of battery deployed nodes. Luckily, research progress on energy harvesting [13] (converting ambient energy into electrical current powering embedded devices) and transient computing [9] (executing computing tasks on embedded device that frequently loses power due to unpredictable energy harvesting source) show the path to a new era of energy-neutral battery-less computers.

An ample body of work is currently tackling the intricacies of intermittent computing [1, 2, 8]. A number of solutions have been proposed that ensure forward progress of program execution and maintain data consistency on systems that experience frequent power outages. Each system takes a different approach providing mixed results on performance, features and design constraints. These systems focus on designing *monolithic software systems* that are being deployed under intermittent conditions with no guarantee over their performance or longevity.

The trend of software development mandates that hardware acts as a hosting node where multiple actors develop and operate software to different ends, leaving monolithic designs behind and embracing multi-tenancy. This is demonstrated in maturity of computing systems from sensors, phones, desktops, and datacenters. Much like other compute paradigms, the process and resource management *as well as scheduling* have a significant impact on the behavior, efficiency, and quality of application of deployed battery-less embedded systems. However, the design space for enabling multi-tenancy on intermittent computers has not been explored.

This paper will shed a light onto the problem of task (or generally speaking: applications) scheduling under intermittent power. Drawing inspiration from task-based intermittent systems [8] we design an intermittent micro-kernel aimed towards hosting multiple applications on transiently powered nodes. We use this deployed kernel to evaluate the performance of common scheduling scenarios under uncertain energy conditions. In contrast to relevant work which tries to solve scheduling under uncertainty, either considering a clairvoyant approach on energy availability [10] or a probabilistic

prediction of an optimal schedule [5], we take a more simple approach. We investigate simple approaches because of our device target: a very constrained system with available execution time less than ≈100 ms before energy outage, with execution tightly coupled to energy availability. Thus spending computation on either computing an elaborate algorithm or predicting energy availability seems unreasonable [3].

Our preliminary results show that scheduling algorithms yield different outcomes than expected from continuously-powered embedded systems: a simplistic round robin implementation yields comparatively good response times and slightly better execution times to an first come first serve algorithm, while at the same time wasted computation due to power failures is worst by a factor of two. This hints that prioritizing fairness over other heuristics might be the key for allocating resources in non-predictive energy scarce environments. Additionally throughput remains relatively stable across all algorithms and no significant gains seem to emerge when employing a predictive technique.

2 SCHEDULING UNDER INTERMITTENCY

Intermittent systems suffer from lack of guarantees over their performance. The hardware hosting the system is able to execute only when energy collected from a harvester reaches a certain level. Taking into account that energy availability is not constant over time and recharging times vary, it becomes clear that the host will actively compute for uncertain amounts of time while frequently, multiple times per second, experiencing loss of power deaths. As compared to the general purpose embedded systems, intermittently-powered systems must overcome two vital challenges caused by the intermittent operation:

- maintaining useful forward progress of computation; and
- ensuring memory consistency.

In order to guarantee forward progress of computation and guarantee memory consistency a number of different of different techniques has been proposed, e.g. [1, 2, 8]. In this implementation we chose a task based approach where we consider tasks to be atomic in the context of intermittency. In order for a task to be considered completed it must successfully complete within an execution interval, thus scheduled tasks should have all-or-nothing semantics; i.e. their re-execution upon power failures should not introduce any side-effects. This side-effect can be described as follows. Upon power failures, a device loses its volatile state, e.g. the contents of the CPU registers and the volatile memory, but the contents of the non-volatile memory, such as FRAM, or peripheral state, persists. An inconsistency might cause intermittent execution to deviate from continuously-powered behavior, often leading to an unrecoverable application failure. Therefore, the scheduler should keep track of the finalized tasks as well as keep the journaled volatile state consistent with the volatile state.

Naturally, extensive profiling and accurately sizing the implementation to the harvesting environment can provide some performance guarantees. However, systems that experience frequent power outages will operate only under a best effort policy. Consequently, if we want to operate complex software in transient computers we should provide some assurance over the expected performance. This is especially critical for systems that serve multiple applications. Since computational availability becomes a highly valued scarce resource, allocating it should be done in a way that best serves the need of the deployed system.

2.1 A Case Study in Multi-Tenancy

We consider an on-body wearable sensor network for health purposes as a demonstrative thought exercise. Energy harvesting devices show the greatest potential in deeply embedded sensor networks and wearable technology, where form factor, lifetime, and cost are of huge importance. We can speculate on the need for multi-tenant applications in wearable healthcare networks. Such a network operates using tens of tiny sensors that are used by a multitude of applications with different requirements and expected outcomes. For instance stress measurement could be achieved by acquiring heart rate R-R interval, electrodermal activity, activity level, noise level etc. Each of the sensory node can host a 'stress app' that does preliminary processing before combining the data with the rest of the network. Guaranteeing a successful stress measurement means that each individual node application successfully run under its timing and operational constraints alongside other applications hosted in the same node.

On the deeply embedded domain a plethora of IoT devices is planned to be installed in access-constrained areas to enable ubiquitous monitoring in the deployed environment. We consider a building monitor sensory network with a number of transient nodes deployed in various locations in the buildings structure. Pipes, ventilation ducts, cabling canals as well as walls and flooring can be enriched with small transient nodes that are able to host multiple applications, provide diverse sensory information for their near environment and report on a main node only if an event of interest has occurred without the need of power connection or battery replacement. As an example a sensor in a cabling canal can host multiple applications that monitor for fire (smoke and temperature sensing), monitor for a rodent infestation (motion detection), monitor for security risks (sound, motion and light detection), etc. Additionally, by allowing multiple applications to be hosted we enable for the system to be more modular and maintainable as new applications can be added in the future without the need to replace the existing node.

Under that context we could define multi-tenancy as the ability of the intermittent kernel to provide every tenant (application in this instance) a dedicated portion of the available resources to the end that every tenant can adequately fulfill its requirement: from a performance standpoint, this on-device pre-processing saves significant energy instead of broadcasting raw data on another medium (backscatter or active radios).

2.2 Scheduling Challenges

Transiently powered systems that experience frequent power outages are prone to starvation problems as a consequence of the operating principle. For example a (computationally) large task can be scheduled for execution without being able to complete after numerous power failures. Since this task restarts every time the system powers up each performed computation is wasted since it has to be repeated, consequently every other application in the

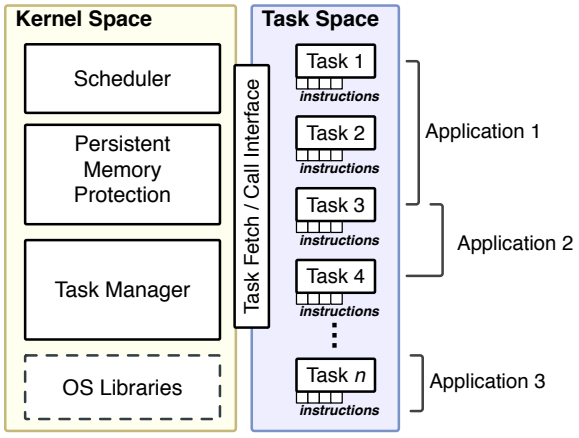

Figure 1: High level task based system architecture and memory space supporting multi-tenancy. Applications are made up of tasks. Tasks use OS level libraries and can be shared among applications.

node is starving while it could utilize that wasted computation to complete its task. Additionally, priority or event driven systems potentially can hog the processing unit if the energy availability coincides with a top priority task or event. Thus we need to be able to schedule applications under frequent power failures in way that will allow for the optimal best effort performance according to the design requirements.

2.3 Performance Metrics

For examining the implications of power failures in scheduling algorithms we advocate for considering the following performance metrics:

- *Completion time*: the time it takes for an application to finish from its initial start;
- *Response Time*: the time it takes for an application to finish from its release;
- *Throughput*: the amount of successful completions per unit of time;
- *Wasted Time*: the amount of time spent on computation that is lost due to power failures.

3 AN INTERMITTENT MICRO-KERNEL

To demonstrate the feasibility of multi-tenancy in intermittently-powered systems, we have developed a specialized software system that is composed of two distinct layers: (i) the micro-kernel layer and (ii) the application layer. The layered design enables a degree of task isolation and protection as in modern operating systems and provides design space for future features such as application update, task graph update and remote kernel instruction.

3.1 Design

The micro-kernel proposed is a system developed solely for the purpose of testing scheduling algorithms while providing some extra features (memory protection, task sharing) that can provide key advantages when operating under frequent power failures. The micro-kernel is the backbone of the designed system and it encompassed all the abstractions and infrastructure needed to host and execute multiple third party applications on intermittent power, in the same compute unit. The kernel is responsible for ensuring forward progress of computation and guaranteeing memory consistency despite power failures. The micro-kernel is composed of a **scheduler** that operates on a ready queue that is the main data structure used to implement several scheduling policies. This queue holds the tasks which are ready to be scheduled and run. The current scheduler implements several scheduling policies for enabling comparisons which will be described later in Section 4.2.

A number of unique challenges were encountered while developing the kernel as forward progress of computation and memory consistency issues have to be ensured not only for the applications but also for the kernel itself. Using double buffering and two-phase commit operations as well as a systems state machine, we were able to maintain the consistency of internal systems data structures; e.g. ready queue, and the correctness of the scheduler.

3.2 Programming Model

Programming the micro-kernel is fairly simple as the programmer just needs to write a function in plain C and declare it as a task using the relevant notation. The function-task accepts no input variables and returns no output. On the contrary data that are crucial to the programmer and need to be protected through power-failures or communicated between tasks are explicitly declared on compile time (using the relevant abstractions) and accesses are handled by a virtual memory manager. The order of execution and data accesses are addressed in the following Sections.

3.3 Virtual Memory Manager

The Virtual Memory Management (VMM) system handles protected variables and data sharing between tasks. The VMM abstracts the physical address space of the non-volatile memory into two regions: 'safe' and 'volatile':

- *Safe*: holds the data for which the system can guarantee their consistency;
- *Volatile*: acts as a buffer where an image of the safe data is manipulated during execution;

During the execution of a task the volatile address space is populated with new data and upon successful termination the Volatile buffer is committed to the Safe area where they can be accessed by the next task. In that fashion a task is unable to directly access the data protection the memory space from unwanted accesses. When a task is activated the VMM maps the variables into the correct address space taking also application context into account. For example if Application 1 and Application 2 are sharing SharedTask when SharedTask attempts to access protected data under the context of Application 1 the VMM is responsible for addressing the memory space referring to that application, thus providing not only memory protection under power failures but also some memory isolation between applications.

3.4 Structure of the Applications

Applications in the context of our intermittent micro-kernel are defined as a graph of independent tasks. More specifically the application is a data structure that holds information for its execution (priority, release time, executing order) and pointers to tasks. The order of execution is defined on compile time and dictates the order in which tasks will be executed as to complete an application.

Tasks are the fundamental unit of execution in our micro-kernel and are considered atomic in the context of intermittency. Tasks contain user code and explicitly declared protected variables protecting against write after read dependencies. Moreover, each task is executed in isolation regarding the memory space of the microcontroller making them reusable by many applications (see Figure 1).

Task reuse is important for memory constrained devices. For example a commonly used filtering task can be used by multiple applications in the same node, reducing the code size footprint, reducing developing cost and allowing reconfigurable versatile applications to run on remote transient nodes. Tasks operate on data in the context of the application they currently execute for, thus application execution can be either non-preemptive (execute the task graph of an application from start to end) or preemptive (execute interleaving applications while switching tasks depending on the current context). Preemption in the context of our system works on a task level. Since tasks are fairly small and the frequency of death is usually high, context switching while ensuring the validity of the preempted stack is an expensive operation which will increase computational burden and memory access cost. This is why when an application is preempted the scheduler waits for the currently running task to complete or a death to occur in which case it switches to the next application of the schedule.

4 IMPLEMENTATION

The system under test in comprised of a TI MSP430FR5994 MCU [6] which hosts the scheduling micro-kernel and an auxiliary MCU (TI MSP430FR5994) which is used to signal each process release in a periodic manner. The micro-kernel experiences frequent power interrupts invoked from a software library developed specifically for that purpose. The power failures are caused after a time interval is elapsed causing the MCU to reset at its initial state. The time intervals are randomly generated and provided to the library on compile time which allows us to use the same random interval pattern under different scheduling policies effectively controlling for the environment energy availability. Since we are targeting small platforms (e.g. computational RFIDs) we are interested in frequent power failures and small (less than 100 ms) execution times we bound our experiment to those constraints.

4.1 Applications Under Test

Although the applications for such system presented in introduction present a compelling use-case for such multi-tenant systems for evaluation we selected three generic applications (see Table 1) which were designed to resemble software commonly found in wireless sensor networks. This choice was made in an attempt to generalize the results of this work, for scheduling, without being bound to the specifics (hardware design, timing constraints, deployment environment) that each application would entail.

Table 1: Description of applications used

	Application 1	**Application 2**	**Application 3**
Task 1	Sample	Sample	Sample
Task 2	Average Filter	Low-Pass Filter	High-Pass Filter
Task 3	Selection Sort	Quick Sort	Bubble Sort
Running Time	16.15 ms	12.84 ms	10.26 ms
Period	65 ms	40 ms	50 ms

The total running time for each application was profiled so we can guarantee the successful completion, avoiding possible deadlocks in the system from tasks that cannot complete due to their relative size in comparison to available execution time.

4.2 Selected Scheduling Algorithms

Current available intermittent runtimes support some variant of the first come first serve principle. The logic being that an available job should be serviced as close to its release as possible opportunistically scavenging all the available current energy. In reality, when a number of different applications need to be serviced, starvation or resource monopoly is a highly probable scenario which is intensified when we consider energy or computational time as a scarce resource.

Moreover lacking reliable timer support in intermittent systems does not allow us to experiment with more time critical algorithms such as Earliest Deadline First or Shortest Remaining Time. In such cases current solutions (e.g. external RTC or persistent timekeeping) does not provide enough accuracy or time granularity at the task level to serve our needs. Therefore we consider three algorithms:

- *First Come First Serve (FCFS):* Non-preemptive scheduling of the applications as soon as they are released.
- *Round Robin (RR):* Preemptive scheduling circularly switching among application when their available time quantum expires.
- *Predictive Algorithm (PERC):* Non-preemptive scheduling based on perceptron prediction, which tries to schedule the application most probable to be completed without a power failure based on previous history.

The predictive algorithm was developed as an adaptation of the perceptron based branch prediction [7] algorithm were the probability that a task will be successfully completed is calculated as the sum of past histories multiplied by their weighting factors (derived from training on past result comparison). We chose this algorithm as it can be easily materialized using binary logic since we wanted to keep scheduling overhead relatively low to other predictive methods but high enough to observe its performance in comparison to other simpler methods.

4.3 Auxiliary Microcontroller

The auxiliary MCU is utilized as a semaphore to signal process released to the MCU under test. Since we are testing three applications, we setup three timers to run periodically according to the period (see Table 1) of each application and raise the corresponding pin to high whenever a timer expires. The pin raise invokes

an interrupt in the MCU under test which signifies the release of an application that needs to be processed. Processes are released by an external source as we wanted to keep process release as a controlled parameter not being affected by power interrupts, so it will be consistent throughout the experimentation phase regardless of the state of the MCU under test.

4.4 Testing Library

Testing the current setup required a library that could provide the environment under which we can evaluate the scheduling algorithms. In reality harvesting environments can provide a huge range of cases under which the system can be evaluated. More specifically the amount of execution time available, time spent while dead, recharge time, energy availability are variables that will change depending on hardware design, deployed environment, time of the day and a number of unaccounted factors. Trying to provide a more general case we developed a small library that acts as a random reseter for the processor under test. The library accepts as an input a number of values which act as noise to be added to a central timing value. This value in addition to the supplied noise creates a pseudo-random executing time available after each reset. The noise values are produced by a python script for random number generation[12] in the [0 us,10000 us] range. For each iteration of the experiment we change the central timing value and supply the same noise pattern as we want to compare the results under the same pseudo-random conditions. As an example if we pick the central value to be 20 ms we expect our execution times to range from 20 ms to 30 ms. The interrupt that occurs in that time incurs a brownout reset (BOR) which leads to a power on reset (POR) and power up clear (PUC) which puts the microcontroller to its initial power-on state with the contents of its registers cleared [6]. After the reset the library sets the new power interrupt to occur by picking a new noise value added to the central timing value.

5 EVALUATION

We use our preliminary intermittent micro-kernel, and the discussed experimental setup to emulate application and energy scenarios to investigate the proposed scheduling protocols. This section presents results of the experiments and our interpretation of them. All the applications are released based on an interrupt invoked by the auxiliary microcontroller on periodic manner. The applications are then inserted to the ready queue and serviced based on the current scheduling algorithm. Figure 2 demonstrates the data collected for each application and scheduling algorithm, with the microcontroller running on transient power. Completion time, response time and throughput are normalized over executing on stable power while computation time wasted (the time a task has spent executing before being intercepted by a power failure) is presented in seconds.

5.1 Interpretation of Results

Examining the results we can see that running on FCFS provides no immediate benefits as completion times and response times are kept in high levels without significant gains in throughput, although FCFS has the least wasted time among any other policy. PERC scheduling seems to have similar performance with FCFS with significantly more time wasted on unfinished tasks.

Interestingly RR scheduling appears to have relatively better performance for Applications 1, 3 and relatively similar performance for Application 2. Gains in completion times are significant in both cases while response times remain relatively small. Interestingly these results are occurring while wasted computation is higher than any other algorithm. This result comes opposite to our intuition that the first come first serve algorithm would yield better response times as we speculated that opportunistically executing a task as fast as possible consuming the latest available energy would benefit task responsiveness. Wasted computation in RR was expected to be high as switching between applications frequently was expected to produce that behavior, but this wasted computing time does not seem to impede the overall performance of the system. On the contrary, it seems that fairness allows the system to perform more balanced, ensuring that all applications have (nearly) the same access to the scarce resource. Moreover observing a relative better performance using round robin scheduling while having 40% more wasted computation provides an indication that optimization could yield even better results. Namely a more flexible and fitting time slice allocation for our selected applications would significantly reduce the penalty we pay for frequently switching among them giving the opportunity to optimize also for wasted computation. Although the result does not present us with a clear top performer it reinforces our belief that traditional scheduling concepts and expectations do not necessarily hold when considering intermittent systems. As a result we have to accurately dimension our system before deployment if we want to be able to guarantee performance especially in multi-tenant systems where energy and computation time has to be used by different applications with diverse specifications.

6 FUTURE WORK

This work provides an initial set of results on the performance of traditional embedded system scheduling techniques translated for the transiently-powered domain. This is also just a first step in enabling multi-tenant systems for intermittently-powered devices. Here we summarize the list of further action items and associated research challenges.

Identifying Properties of Deployed Embedded Software. To be able to exploit multi-tenancy at intermittently-powered devices to the fullest, we need to know more about the characteristics of the *real* applications running on intermittently-powered devices. The information of the underlying software will provide input to the design of new schedulers. Points of interest include:

- Which functions are the most widespread among existing deployed code—is it e.g. averaging, FFT, sorting, or yet another application?
- What are the properties of variables passed by the functions (their size, type)?
- How large are the functions that can be shared by the multi-tenant system (in terms of memory footprint)?
- How similar are the same function implemented by different developers?

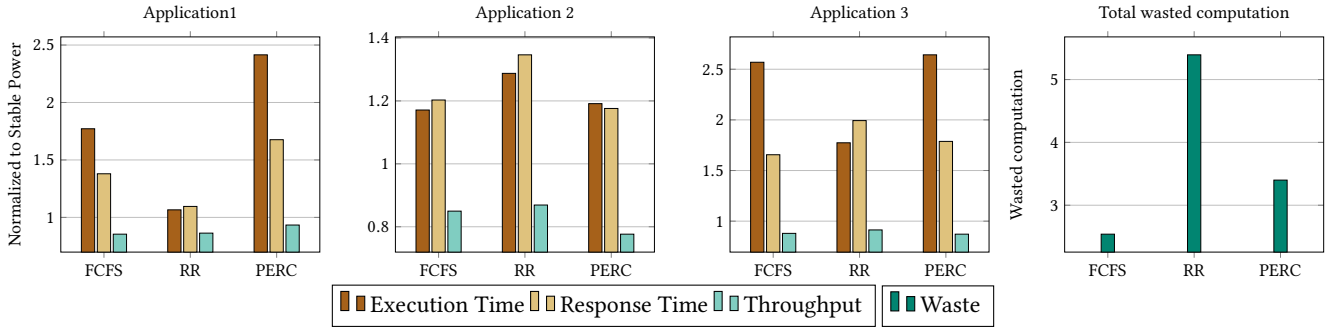

Figure 2: Scheduling performance for FCFS, RR, PERC.

One approach to answer these questions is to mine software repositories of open-sourced intermittent-applications.

Bringing Time to Scheduling. Any scheduling algorithm would benefit from notion of time. This is however difficult in the context of intermittently-powered domain, where energy required for keeping time is unavailable at times of device "death". One idea is to measure time with remanence timekeepers, such as [4]. More specifically, utilizing timekeeping enables scheduling deadlines and periodic tasks which we are currently unable to cover sufficiently in this work.

Definition of Realistic Energy Availability Models. A rigid classification of common energy availability environments and harvesting techniques is required to design realistic micro-kernels. This will provide an opportunity of better framing the problem and more adequately providing possible solutions for scheduling and multi-tenancy. Utilizing and improving tools like Ekho [3] for emulating realistic harvesting scenarios and applying possible results to real-life scenarios is fundamental for this task. We note that definitions of realistic energy availability models is a more broad problem applied to any sub-domains of intermittent computing.

7 CONCLUSIONS

Processing over uncertain amounts of time without having data either for the energy availability of the environment nor for the available computation time poses a challenge for battery-less systems. Although transiently powered computers by default operate in a best-effort basis, some performance must be guaranteed if we want to be able to build systems that meet developer's requirements. As shown by our data, traditional scheduling algorithms are not performing as expected under power failures. Moreover available energy and processing power is a scarce and unpredictable resource that is expended more efficiently when distributed amongst all available applications rather than opportunistically attempting to execute them to completion. Optimizing resource sharing and minimizing scheduling overheads as well as wasted computation might have significant positive impact application performance is battery-free systems.

REFERENCES

[1] Domenico Balsamo, Alex Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining Computation during Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters* 7, 1 (Mar. 2015), 15–18.

[2] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. *SIGPLAN Not.* 51, 10 (Oct 2016), 514–530.

[3] Matthew Furlong, Josiah Hester, Kevin Storer, and Jacob Sorber. 2016. Realistic Simulation for Tiny Batteryless Sensors. In *Proc. International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys 2016)*. 23–26.

[4] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proc. 15th ACM Conference on Embedded Network Sensor Systems (SenSys 2017)*. 17:1–17:13.

[5] Jason Hsu, Sadaf Zahedi, Aman Kansal, Mani Srivastava, and Vijay Raghunathan. 2006. Adaptive Duty Cycling for Energy Harvesting Systems. In *Proc. International Symposium on Low Power Electronics and Design*. Tegernsee, Bavaria, Germany.

[6] Texas Instruments. 2017. User's Guide, MSP430FR58xx,MSP430FR59xx,MSP430FR6xx Family. http://www.ti.com/lit/ug/slau367o/slau367o.pdf

[7] Daniel A Jiménez and Calvin Lin. 2001. Dynamic branch prediction with perceptrons. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. 197–206.

[8] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proc. OOPSLA* 1, OOPSLA (Oct. 2017), 96:1–96:30.

[9] Geoff V. Merrett. 2016. Invited: Energy harvesting and transient computing: A paradigm shift for embedded systems?. In *Proc. ACM/EDAC/IEEE Design Automation Conference*.

[10] Clemens Moser, Davide Brunelli, Lothar Thiele, and Luca Benini. 2007. Real-time scheduling for energy harvesting sensor nodes. *Real-Time Systems* 37, 3 (Dec. 2007), 233–260.

[11] Joseph A. Paradiso and Thad Starner. 2005. Energy Scavenging for Mobile and Wireless Electronics. *IEEE Pervasive Computing* 4, 1 (Jan.-Mar. 2005), 18–27.

[12] randit documentation 2018. The Python Standard Library » 9. Numeric and Mathematical Modules ». Retrieved July 30, 2018 from https://docs.python.org/2/library/random.html

[13] Sujesha Sudevalayam and Purushottam Kulkarni. 2011. Energy Harvesting Sensor Nodes: Survey and Implications. *IEEE Communications Surveys and Tutorials* 13, 3 (Third Quarter 2011), 443–461.